Tencalc Users' Guide

Release 0.1a

Joao Pedro Hespanha

Mar 11, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

TensCalc is a MATLAB© toolbox that generates optimized code to perform computations involving *tensors*, i.e., multi-dimensional arrays. TensCalc was specifically designed to solve nonlinear constrained optimizations very efficiently, but its code generation engine can be used to generate code to perform much more general computations.

TensCalc is aimed at scenarios where one needs to perform very fast a large number of computations (e.g., optimizations) that depend on parameters that change from one instance of the computation to the next instance, but the overall structure of the computation remains the same. This is common in applications where the computations/optimizations depend on measured data and one wants to compute optima for large or evolving datasets, e.g., in robust estimation and classification, maximum likelihood estimation, model predictive control (MPC), moving horizon estimation (MHE), and combined MPC-MHE (which requires the computation of a saddle-point equilibrium).

1.1 What type of code is generated by TensCalc?

TensCalc can either produce optimized MATLAB© or C code. The former is only preferable for very large problems with mild speed requirements, whereas the latter is aimed at small to medium-size problems that need to be solved in just a few milliseconds.

The C code is completely self-contained and library free (aside from the C standard library), making it extremely portable.

For ease of use within MATLAB©, both the C and MATLAB© code are encapsulated into MATLAB© classes that appear indistinguishable to the MATLAB© user, aside from the speed of execution. Within the class wrapper, the C code is called from MATLAB© using the *cmex* interface and dynamic libraries. **Typically, C code is 10-100 times faster.**

1.2 Which optimizations is TensCalc best at?

In the context of optimizations, 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 variables to be optimized can be multi-dimensional arrays of any dimension (tensors) and the cost functions and inequality constraints are specified using MATLAB©-like formulas.

TensCalc's optimization solvers uses primal-dual interior point methods and uses formulas for the gradient and the hessian matrix that are computed symbolically in an automated fashion.

1.3 Why is TensCalc-generated code fast?

The speed achieved by code generated by TensCalc arises from a combination of features: reuse of intermediate computations across and within iterations of the solver, detection and exploitation of matrix sparsity, avoidance of run-time memory allocation and garbage collection, and reliance on flat code that improves the efficiency of the micro-processor pipelining and caching. All these features have been automated and embedded into the code generation process.

A price to pay for TensCalc's speed is that the structure of the computation/optimization is fixed at code-generation time. This has two important consequences:

- the sizes of all variables must be specified at code-generation time, and
- the sparsity structure of all computations is automatically determined at code-generation time and assume that all external parameters will generally be nonzero.

1.4 How does TensCalc syntax compare to MATLAB©?

TensCalc computations are defined using operations on symbolic variables that very much resemble MATLAB©'s functions and operators. This means that MATLAB© user's will easily understand TensCalc's syntax. However, there are a few key issues that must be kept in mind:

- TensCalc is very picky about the sizes of tensors and several conversions of sizes that MATLAB© performs automatically must be done manually in TensCalc. For example, TensCalc distinguishes between 2-by-1 matrices and 2-vector (see *What are Tensors?*), which cannot be added together in TensCalc without an explicit `reshape()` operation.

  > **Warning:** This is likely the case for most syntax errors when seasoned MATLAB© users first start to work with TensCalc.

- A few TensCalc functions have slightly different behavior (and sometimes syntax) than the corresponding MATLAB© function with the same name. Examples of this include the matrix inverse `inv` and determinant `det` that in TensCalc can only be applied to matrices that have been factorized using `lu` or `ldl`.

- TensCalc has a few functions that do not exist in MATLAB©, but can be used to generate more efficient code; most notably the function *tprod()* that provides a very general multiplication operation between tensors (see *Tensor product*). This operation includes the usual matrix multiplication *() as a special case, the summation over rows and/or columns *sum()*, computing the trace of a matrix *trace()*, extracting a matrix main diagonal *diag()*, computing the euclidean norm of a vectors *norm()*; as well as generalizations of all these operations to n-dimensional tensors.

CHAPTER

TWO

INSTALLATION

TensCalc is available on GitHub and requires the installation of 3 toolboxes, which must be installed in the following order:

1. FunParTools: https://github.com/hespanha/funpartools
2. CmexTools: https://github.com/hespanha/cmextools
3. TensCalc: https://github.com/hespanha/tenscalc

Installations instructions for all these tools is available at GitHub.

2.1 Issues

While most MATLAB© scripts are agnostic to the underlying operating systems (OSs), the use of mex functions depends heavily on the operating system.

Our goal is to build a toolbox that works across multiple OSs; at least under OSX, linux, and Microsoft Windows. However, most of our testing was done under OSX so one should expect some bugs under the other OSs. Currently, it is fair to say that TensCalc has been tested

- fairly extensively under OSX
- lightly under linux
- very lightly under Microsoft Windows

Any help in fixing bugs is greatly appreciated.

CHAPTER

THREE

BASICS

TensCalc computations and optimizations are defined using symbolic expressions supported by the class `Tcalculus`, named after *tensor calculus*.

3.1 What are Tensors?

In TensCalc, tensors are multi-dimensional real-valued arrays over which one performs numerical operations. The `size()` of a tensor is a vector that specifies how many entries the tensor has in each dimension. The following table shows common tensors and their sizes:

Table 1: Common tensor types

name	number of dimensions	TensCalc's size	mathematical set
scalar	0	`[]`	$\mathbb{R}$
n-vector	1	`[n]`	$\mathbb{R}^{n}$
n-by-m matrix	2	`[n,m]`	$\mathbb{R}^{n \times m}$
tensor with d of dimensions	d	`[n1,n2,...,nd]`	$\mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$

Warning: It is common (at least within MATLAB©) to make no distinction between a scalar, a 1-vector, and a 1-by-1 matrix. In MATLAB© it is also common not to distinguish between an n-vector and an n-by-1 matrix. However, TensCalc does make a distinction between all these and does not allow, e.g., adding a 2-vector with a 2-by-1 matrix, without an explicit `reshape()` operation.

TensCalc is slightly more permissive with operations between scalars and tensors of larger sizes in that it does allow summations between a scalar and a tensor of any size, with the understanding that the scalar is added to *every entry* of the tensor, which is consistent with MATLAB©'s syntax.

See *Using MATLAB© matrices in TensCalc symbolic expressions* regarding the rules for automatic conversion of sizes between MATLAB© matrices and TensCalc tensors.

3.2 Building `Tcalculus` symbolic expressions

TensCalc symbolic expressions are supported by the class

`class Tcalculus`

but you never really need to call this class directly. Instead, symbolic expressions are built like regular MATLAB© expressions using many of the usual MATLAB© operators and functions, which have been overloaded to operate on TensCalc symbolic expressions.

Warning: To speed up operations, the class *`Tcalculus`* keeps some information in a few auxiliary global variables. Before defining a new computation or optimization it is a good idea to clear these variables with

```
clear all
```

If this is not possible, one can use the following method to just clear *`Tcalculus`*'s auxiliary global variables

`Tcalculus.`**`clear`**`()`

However, after using this method, any instances of the *`Tcalculus`* class that remains in memory becomes invalid and will lead to errors if used in a subsequent computations or optimizations **without necessarily resulting in a syntax error**. It is thus strongly advised to use `clear all` rather than `Tcalculus.clear`.

3.2.1 Symbolic variables

Most symbolic expressions start with symbolic variables that represent tensors that will only be assigned numeric values at code-execution time. Symbolic variables are created using

`Tvariable name size`

Parameters

- **`name`** (`character string`) – Name of the variable to be created
- **`size`** (`vector of integers`) – Size of the tensor as a vector of integers. When omitted, the empty size `[]` is assumed, which corresponds to a scalar.

The variable is created in the caller's workspace:

```
>> clear all
>> Tvariable a [3,4];
>> whos
   Name      Size            Bytes  Class        Attributes
   a         3x4                39  Tcalculus
   ans       3x4                39  Tcalculus
```

`Tvariable()` can also be used as a regular MATLAB© function using the syntax:

Tvariable(*name*, *size*)

Returns
Tcalculus symbolic expression holding the symbolic variable

In this case, *Tvariable()* returns a symbolic expression that contains the variable. Note that the name of the symbolic variable is still given by `name` regardless of the name of the variable that you use to store it in:

```
>> clear all
>> y=Tvariable('a',[3,4]);
>> whos
   Name      Size            Bytes  Class        Attributes
   y         3x4                39  Tcalculus
>> disp(y)
variable1   = variable('a',) : [3,4]
```

3.2.2 Using MATLAB© matrices in TensCalc symbolic expressions

Often symbolic expressions involve regular MATLAB© matrices, as in adding a symbolic variable created with *Tvariable()* with a regular MATLAB© matrix:

```
>> Tvariable a [2,2]
>> b=a+[2,1;3,4]
```

The following rules are used to convert MATLAB© matrices to TensCalc symbolic expressions. In essence, the conversion is completely transparent *unless the matrix is 1-by-1 or n-by-1.*

Table 2: Rules for automatic convertion between MATLAB© matrices and TensCalc tensors

MATLAB© variable	MATLAB©'s size	TensCalc variable	TensCalc's size
1-by-1 matrix	`[1,1]`	scalar	`[]`
n-by-1 column matrix	`[n,1]`	n-vector	`[n]`
1-by-n row matrix	`[1,n]`	1-by-n tensor	`[1,n]`
m-by-n matrix	`[m,n]`	m-by-n tensor	`[m,n]`
n1-by-n2- ... -by-nd matrix	`[n1,n2,...,nd]`	n1-by-n2- ... -by-nd tensor	`[n1,n2,...,nd]`

TensCalc also provides a few functions that can be used to directly create symbolic tensors that can be useful when either the rules above do not have the desired effect (e.g., in the somewhat unlikely case wants to create a `1-by-1` tensor, rather than a scalar) or when it is more efficient to directly create the symbolic tensor.

Table 3: Functions to create (constant) tensors

Usage	Description	Notes
Tzeros([*n1*, ..., *nd*]) **Tzeros**(*n1*, ..., *nd*)	Returns a tensor with all entries equal to 0. The tensor size can be provided as a single vector with the number of entries in all directions, or as multiple input parameters, one per dimension.	This function can be used interchangeably with the regular MATLAB© function `zeros`, as long as the the size conversion rules in *Rules for automatic convertion between MATLAB© matrices and TensCalc tensors* are appropriate.
Tones([*n1*, ..., *nd*]) **Tones**(*n1*, ..., *nd*)	Returns a tensor with all entries equal to 1. The tensor size is specified as in *Tzeros()*.	This function can be used interchangeably with the regular MATLAB© function `ones`, as long as the the size conversion rules in *Rules for automatic convertion between MATLAB© matrices and TensCalc tensors* are appropriate.
Teye([*n1*, ..., *nk*, *n1*, ..., *nk*]) **Teye**(*n1*, ..., *nk*, *n1*, ..., *nk*)	Returns an "identity" tensor with all entries equal to zero, except for the entries with the indice 1 equal to the indice k+1, the indice 2 equal to the indice k+2, ..., as in `(i1,i2,...,ik,i1,i2,...,ik)` These entries are all equal to 1.	For `k=1`, this function can be used interchangeably with the regular MATLAB© function `eye`, but is convenient to generate "identity" tensors with a larger number of dimensions.
Tconstant(*mat*, *size*)	Returns a tensor with entries given by the matrix `mat` and size specified by `size` as in *Tzeros()*.	This function is typically used to override the size conversion rules in *Rules for automatic convertion between MATLAB© matrices and TensCalc tensors*.

Warning: When called with a single argument *Tzeros()*, *Tones()*, and *Teye()* differ from their MATLAB© counterparts: *Tzeros()* and *Tones()* return vectors (i.e., tensors with 1 dimension), rather than square matrices, and *Teye()* generates an error since it has no counter-part for vectors.

3.3 Accessing information about the size of symbolic variables

Table 4: Functions to access information about symbolic expressions

Usage	Description	Notes
`size`(*X*) `[n1,n2,...,nd]=size(X)` `size`(*X*, *dim*)	Size of a tensor	Similar syntax to MATLAB©
`ndims`(*X*)	Number of dimensions in a tensor	Similar syntax to MATLAB©. Note that a scalar (which has empty `size = []`) always has 0 dimensions.
`numel`(*X*)	Number of elements in a tensor	Similar syntax to MATLAB©. Note that a scalar (which has empty `size = []`) always has 1 elements.
`length`(*X*)	Length of a tensor	Similar syntax to MATLAB©. Note that a scalar (which has empty `size = []`) always has 1 elements.
`isempty`(*X*)	True for an empty tensor	Similar syntax to MATLAB©. Note that a scalar (which has empty `size = []`) always has 1 elements so it is never empty.

3.4 Indexing and resizing symbolic variables

TensCalc uses the same syntax as MATLAB© to index tensors:

- `X(i,j,k,...)` returns a subtensor formed by the elements of `X` with subscripts vectors `i,j,k,...`

  The resulting tensor has the same number of dimensions as `X`, with lengths along each dimension given by `length(i), length(j), length(k),...`

- A colon `:` can be used as a subscript to indicate all subscripts on that particular dimension.

- The keyword `end` can be used within an indexing expression to denote the last index. Specifically, `end = size(X,k)` when used as part of the `k`th index.

- `subsref(X,S)` with a subscript reference structure `S` behaves as in regular MATLAB©, with he caveat that entries of the tensor must always be indexed using subscripts the type `()`. However, this does not preclude the construction of cells of *`Tcalculus`* objects.

Table 5: Functions reshape and resize symbolic variables

Usage	Description	Notes
reshape(*X*, *size*) **reshape**(*X*, *n1*, ..., *nd*)	Reshape array	Similar syntax to MATLAB©, except that it does not support using `[]` as one of the dimensions.

repmat(*X*, *size*)
repmat(*X*, *n1*, ..., *nd*)
	Replicate and tile tensor	Similar syntax to MATLAB©, with the understanding that the number of dimensions of `X` must be strictly preserved. Often *repmat()* needs to be combined with *reshape()* to obtain the desired effect. For example to replicate twice a 3-vector `X` to create a 3-by-2 matrix by placing the copies of `X` side by side, one needs:
		`Y=repmat(reshape(X,3,1),1,2);`
		or, to create a 2-by-3 matrix by placing the copies of `X` one on top of the other, one needs:
		`Y=repmat(reshape(X,1,3),2,1);`

cat(*dim*, *A*, *B*, ...)	Concatenate tensors along the dimension `dim`	Similar syntax to MATLAB©, with the understanding that the number of dimensions of the input tensors `A,B,...` must match for all but dimensions but `dim`. TensCalc will try to add singleton dimensions to make *cat()* succeed as best as it can. For example in the following code, the scalar variable `a` with size `[]` is augmented to the size `[1,1]` so that concatenation to a 2-by-3 matrix is possible:
		`>> Tvariable a [];`
		`>> b=[a,a,a;a,a,a];`
		`>> size(b)`
		`ans =`
		`2 3`
		and in the following code, the vector variable `b` with size `[3]` is augmented to the size `[3,1]` so that concatenation to a 3-by-2 matrix is possible:
		`>> Tvariable b [3];`
		`>> c=[b,b];`
		`>> size(c)`
		`ans =`
		`3 2`

vertcat(*A*, *B*, ...)	Concatenate tensors along the 1st dimension, which is equivalent to `cat(1,A,B,...)` and also to ``[A;B;...]`	Similar syntax to MATLAB©, with the same caveats as *cat()*.
horzcat(*dim*, *A*, *B*, ...)	Concatenate tensors along the 2nd dimension, which is	Similar syntax to MATLAB©, with the same caveats as *cat()*.

Warning: Unlike MATLAB©, TensCalc does *not* allow for subscripted assignments, such as

```
A(1,2)=5
```

This functionality needs to be achieved with the concatenation operations *cat()*, *vertcat()*, *horzcat()*.

3.4.1 Creating structured matrices

The function *vec2tensor()* is very useful to create structured matrices from vectors, to be used as optimization variables

- Diagonal matrix:

```
% Creates an NxN diagonal matrix
Tvariable v [N];
A=vec2tensor(v,[N,N],[1:N;1:N]');
```

- Lower triangular matrix:

```
% Creates an NxN lower triangular matrix
[i,j]=find(ones(N));
k=find(i>=j);
Tvariable v length(k);
A=vec2tensor(v,[N,N],[i(k),j(k)]);
```

- Symmetric matrix:

```
% Creates an NxN symmetric matrix
[i,j]=find(ones(N));
kl=find(i>j);
k0=find(i==j);
Tvariable v length(k0)+length(kl);
A=vec2tensor([v;v(1:length(kl))],[N,N],[i(kl),j(kl);i(k0),j(k0);j(kl),
↪i(kl)]);
```

- Matrix with the same sparsity structure as a known matrix:

```
% Creates a matrix with the sparsity structure of S
[i,j]=find(S);
Tvariable v length(i);
A=vec2tensor(v,size(S),[i,j]);
```

In all these examples, one would set v to be an optimization variable, that implicitly represents the structured matrix.

CHAPTER

FOUR

OPERATIONS ON SYMBOLIC EXPRESSIONS

4.1 Arithmetic operations

TensCalc supports most of the basic arithmetic operations in MATLAB©, in most cases using the same or a very similar syntax.

Table 1: Basic arithmetic operations

Usage	Description	Notes
`X + Y` `plus`(*X*, *Y*)	Entry-by-entry addition of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `plus()`, expansion upon singleton dimensions is not performed automatically to match the tensors' sizes.
`X - Y` `minus`(*X*, *Y*)	Entry-by-entry subtraction of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `minus()`, expansion upon singleton dimensions is not performed automatically to match the tensors' sizes.
`X .* Y` `times`(*X*, *Y*)	Entry-by-entry multiplication of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `times()`, expansion upon singleton dimensions is not performed automatically to match the tensors' sizes.
`X * Y` `mtimes`(*X*, *Y*)	Matrix multiplication of tensors that adapts to the size of the operands as follows: • regular matrix multiplication when `X` and `Y` are both matrices (tensors with 2 dimensions) with `size(X,2)==size(Y,1)` • matrix by column-vector multiplication when `X` is a matrix (tensor with 2 dimensions) and `Y` a vector (tensor with 1 dimension) with `size(X,2)==size(Y,1)` • row vector by matrix multiplication when `X` is a vector (tensor with 1 dimension) and `Y` a matrix (tensors with 2 dimensions) with `size(X,1)==size(Y,1)` • inner product when `X` and `Y` are both vectors (tensors with 1 dimension) with `size(X,1)==size(Y,1)` • entry-by-entry multiplication with either `X` or `Y` are scalars (tensor with 0 dimensions).	Depending on the sizes of the parameters, this operation may behave quite differently from MATLAB©'s matrix multiplication `mtimes()`.
`X ./ Y` `rdivide`(*X*, *Y*)	Entry-by-entry right division of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `rdivide()`, expansion upon singleton dimensions is not performed automatically to match matrix sizes.
`X .\\ Y` `ldivide`(*X*, *Y*)	Entry-by-entry left division of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `ldivide()`, expansion upon singleton dimensions is not performed automatically to match matrix sizes.
`sum`(*X*, *vecdim*) `sum`(*X*, *'all'*)	Sum of entries of the tensor `X` along the directions specified by the vector `vecdim`, or over all dimensions. Resulting in a vector with the same size as `X`,	Similar syntax to MATLAB©

4.1.1 Tensor product

The function *tprod()* provides a very general and flexible multiplication operation between tensors, which includes the usual matrix multiplication * as a special case, the summation over rows and/or columns *sum()*, computing the trace of a matrix *trace()*, extracting a matrix main diagonal *diag()*, computing the Euclidean norm of a vectors *norm()*; as well as generalizations of all these operations to n-dimensional tensors:

tprod(*A*, *a*, *B*, *b*, *C*, *c*, ...)

Parameters

- **A** (*Tcalculus* symbolic tensor) – 1st tensor
- **a** (vector of integers) – indices for A with length ndims(A)
- **B** (*Tcalculus* symbolic tensor) – 2nd tensor
- **b** (vector of integers) – indices for B with length ndims(B)
- **C** (*Tcalculus* symbolic tensor) – 3rd tensor
- **c** (vector of integers) – indices for C with length ndims(C)

tprod() returns a tensor Y obtained using a summation-product operation of the form

$$Y(y_1, y_2, ...) = \sum_{s_1} \sum_{s_2} \cdots A(a_1, a_2, ...) B(b_1, b_2, ...) C(c_1, c_2, ...) \cdots$$

with the matching between the result tensor indices $y_1, y_2, ...$, the summation indices $s_1, s_2, ...$, and the input tensors indices $a_1, a_2, ..., b_1, b_2, ..., c_1, c_2, ...$ is determines as follows:

- A negative values of -1 in one or several of the input tensor indices $a_1, a_2, ..., b_1, b_2, ..., c_1, c_2, ...$ means those particular indices should be summed under the 1st summation operation.
- A negative values of -2 in one or several of the input tensor indices $a_1, a_2, ..., b_1, b_2, ..., c_1, c_2, ...$ means those particular indices should be summed under the 2nd summation operation.

  ...
- The absence of a negative index means that there are no summations.
- A positive value of +1 in one or several of the input tensor indices means those indices should match the 1st index y_1 of the result tensor Y.
- A positive value of +2 in one or several of the input tensor indices means those indices should match the 2nd index y_1 of the result tensor Y.

  ...
- The absence of a positive index means that the result has an empty size (i.e., it is a scalar).

A few examples are helpful to clarify the syntax and highlight the flexibility of *tprod()*:

```
Tvariable A [m,n];
Tvariable x [n]
```

(continues on next page)

(continued from previous page)

```
Tvariable B [n,k]

tprod(A,[1,-1],x,[-1]);    % same as the matrix-vector product A*x
tprod(A,[1,-1],B,[-1,2]);  % same as the matrix-matrix product A*B

tprod(A,[2,1]);            % same as the transpose A'

tprod(A,[1,-1]);           % same as sum(A,2)

tprod(x,[-1],x,[-1])       % same as the square of the Euclidean norm x'*x
tprod(A,[-1,-2],A,[-1,-2]) % same as the square of the Frobenius norm

Tvariable C [n,n]
tprod(C,[1,1])             % vector with the main diagonal of A
tprod(C,[-1,-1])           % same as trace(A)
```

4.2 Logical-valued operations

TensCalc uses logical-valued operations mostly to specify optimization constraints.

Table 2: Logical-valued operations

Usage	Description	Notes
X==Y **eq**(*X*, *Y*)	Entry-by-entry equality comparison of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `eq()`, expansion upon singleton dimensions is not performed automatically to match the tensors' sizes.
X>=Y **ge**(*X*, *Y*)	Entry-by-entry greater than or equal to comparison of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `ge()`, expansion upon singleton dimensions is not performed automatically to match the tensors' sizes. From the perspective of a constrained optimization numerical solver, due to finite numerical precision, X>=Y and X>Y represent the same constraint.
X>Y **gt**(*X*, *Y*)	Entry-by-entry greater than comparison of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `gt()`, expansion upon singleton dimensions is not performed automatically to match the tensors' sizes.
X<=Y **le**(*X*, *Y*)	Entry-by-entry smaller than or equal to comparison of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `le()`, expansion upon singleton dimensions is not performed automatically to match the tensors' sizes. From the perspective of a constrained optimization numerical solver, due to finite numerical precision, X<=Y and X<Y represent the same constraint.
X<Y **lt**(*X*, *Y*)	Entry-by-entry smaller than comparison of tensors of the same size or of a scalar with a tensor of arbitrary size.	Unlike MATLAB©'s regular `lt()`, expansion upon singleton dimensions is not performed automatically to match the tensors' sizes.
all(*X*, *dim*) **all**(*X*, *'all'*)	Checks if the entries of the tensor `X` are nonzero and performs the Boolean operation `and` along the dimensions specified in `vecdim` (1st form) or along every dimension (2nd form), producing the logical value `true` if all entries are nonzero. The result is a tensor with the same size as X, but with the dimensions in `vecdim` removed.	Similar syntax to MATLAB©.
any(*X*, *dim*) **any**(*X*, *'all'*)	Checks if the entries of the tensor `X` are nonzero and performs the Boolean operation `or` along the dimensions specified in `vecdim` (1st form) or along every dimension (2nd form), producing the logical value `true` if at least one entry is nonzero. The result is a tensor with the same size as X, but with the dimensions in `vecdim` removed.	Similar syntax to MATLAB©.

4.3 Entry-wise operations

The following functions are applied to every entry of a tensor.

Table 3: Entry-wise operations

Usage	Description	Notes
`exp`(*X*)	Exponential of tensor entries.	Similar syntax to MATLAB©.
`log`(*X*)	Natural logarithm of tensor entries.	Similar syntax to MATLAB©.
`sin`(*X*)	Sine of tensor entries in radians.	Similar syntax to MATLAB©.
`cos`(*X*)	Cosine of tensor entries in radians.	Similar syntax to MATLAB©.
`tan`(*X*)	Tangent of tensor entries in radians.	Similar syntax to MATLAB©.
`atan`(*X*)	Inverse tangent in radians of tensor entries.	Similar syntax to MATLAB©.
`sqr`(*X*)	Square of tensor entries.	Similar to `X.^2` or `X .* X`.
`cube`(*X*)	Cube of tensor entries.	Similar to `X.^3` or `X .* X .* X`.
`X.^Y` `power`(*X*, *Y*)	Element-wise `X` raised to the power `Y`.	Similar syntax to MATLAB©, but TensCalc requires the power `Y` to be a regular numeric scalar, *not* a matrix/vector nor Tcalculus symbolic expression.
`sqrt`(*X*)	Square root of tensor entries.	Similar syntax to MATLAB©.
`round`(*X*)	Round to nearest integer	Similar syntax to MATLAB©, except that it does **not** support a second argument specifying a desired number of digits for rounding to a decimal.
`ceil`(*X*)	Round to nearest integer towards +infinity.	Similar syntax to MATLAB©.
`floor`(*X*)	Round to nearest integer towards -infinity.	Similar syntax to MATLAB©.
`sign`(*X*)	Signum function applied to the entries `X`, equal to 1, 0, or -1, depending on whether the corresponding extry of `X` is positive, zero, or negative.	Similar syntax to MATLAB©,
`heaviside`(*X*)	Step or heaviside function applied to the entries `X`, equal to 1, 0.5, or 0, depending on whether the corresponding entry of `X` is positive, zero, or	Similar syntax to MATLAB©.

4.4 Linear algebra

TensCalc supports several basic linear algebra operations, but for operations that require some form of factorization to be performed efficiently, such as:

```
det()    logdet()    inv()    mldivide()    \    traceinv()
```

TensCalc requires the user to explicitly select the factorization desired:

```
ldl()    lu()
```

This is accomplished by passing to the function the factorized matrix, as in:

```
det(lu(A))            det(ldl(A))
logdet(lu(A))         logdet(ldl(A))
inv(lu(A))            inv(ldl(A))
traceinv(lu(A))       traceinv(ldl(A))
mldivide(lu(A),Y)     mldivide(ldl(A),Y)
lu(A)\B               ldl(A)\B
```

This syntax works because in TensCalc the factorization functions *lu()* and *ldl()* return the *whole factorization as a single entity*, which can then be passed to any function that take a factorization as input (such as the functions listed above). This behavior is distinct from regular MATLAB© for which *lu()* and *ldl()* return factorizations through multiple outputs.

TensCalc's code generation makes sure that redundant computations are not executed, therefore the following two snippets of code result in the same computation:

```
Tvariable A [10,10]
Tvariable b [10]
facA=lu(A);
y=det(facA);
x=facA\b;
```

or:

```
Tvariable A [10,10]
Tvariable b [10]
y=det(lu(A));
x=lu(A)\b;
```

Specifically note that, even though `lu(A)` appears twice in the bottom snippet, the matrix `A` is only factored once.

4.4.1 Which factorization to use?

- The LDL factorization is faster and requires less memory. However, it has some limitations:
  - LDL should only be used for symmetric matrices. When used on a matrix that is not symmetric, all the entries above the main diagonal are ignored.
  - TensCalc's LDL factorization only works for matrices that do not have zeros in the main diagonal. Structural zeros in the main diagonal will result in an error at code generation time. Non-structural zeros (i.e., zeros that cannot be determined at code generation time) will lead to divisions by zero at run time.
- The LU factorization is a little slower and requires twice as much memory to store both the L and U factors. However, it can be applied to non-symmetric matrices and matrices with zeros in the main diagonal.

Both the LU and the LDL factorizations, use "pychologically lower/upper-triangular matrices", i.e., matrices that are triangular up to a permutation, with permutations selected to minimize the fill-in for sparse matrices and reduce computation time (see MATLAB©'s documentation for `lu()` and `ldl()` with sparse matrices).

Table 4: Linear Algebra operations

Usage	Description	Notes
`diag`(*v*, *k*) `diag`(*v*) `diag`(*A*)	When `v` is an `n`-vector, returns a square matrix with `n+abs(k)` rows/columns, with the `k`-th diagonal equal to `v`. `k=0` corresponds to the main diagonal, `k>0` above the main diagonal and `k<0` below. When `k` ommited, it is assumed equal to (main diagonal). When `A` is an `n`-by-`n` a matrix, returns an `n`-vector with the main diagonal of `A`.	Similar syntax to MATLAB©, except that it does **not** support a second argument specifying a diagonal other than the main diagonal, when `A` is a matrix.
`trace`(*A*)	Trace of a matrix, i.e., sum of the diagonal elements of `A`, which is also the sum of the eigenvalues of `A`.	Similar syntax to MATLAB©.

`A.'`
`A'`
`transpose`(*A*)
`ctranspose`(*A*)
	Transpose of a real-valued matrix.	Similar syntax to MATLAB©, except that TensCalc does not support complex-valued variables and therefore `transpose` and `ctranspose` return the same values.

`lu`(*A*)	LU factorization using "pychologically lower/upper-triangular matrices", i.e., matrices that are triangular up to a permutation, with permutations selected to minimize the fill-in for sparse matrices and reduce computation time (see MATLAB©'s documentation for lu with sparse matrices).	The output of this function includes the whole factorization as a single entity, in a format that can be passed to functions that require factorizations (such as mldivide, inv, det, logdet, traceinv), but should *not* be used by functions that are not expecting a factorized matrix as an input.
`ldl`(*A*)	LDL factorization using "pychologically lower-triangular matrices", i.e., matrices that are triangular up to a permutation, with permutations selected to minimize the fill-in for sparse matrices and reduce computation time (see MATLAB©'s documentation for lu with sparse matrices). All entries of `A` above the main diagonal are ignored and assumed to be equal to the one below the main diagonal, *without performing any test regarding of whether or not this is true*.	The output of this function includes the whole factorization as a single entity, in a format that can be passed to functions that require factorizations (such as mldivide, inv, det, logdet, traceinv), but should *not* be used by functions that are not expecting a factorized matrix as an input.

`lu(A) \\ B`
`mldivide`(*lu(A)*, *B*)
`ldl(A) \\ B`
`mldivide`(*ldl(A)*, *B*)
	Left matrix division, which is the solution to the system of equations `A*X=B` where `A` must be a nonsingular square matrix (tensor with 2 dimensions) and `B` a tensor	MATLAB© allows for non-square and possibly singular matrices `A`, in which case the least-squares solution to `A*X=B` is returned. Currently, TensCalc requires `A` to be square and

Warning: Calling \, *mldivide()*, *inv()*, *det()*, *logdet()*, *traceinv()*, *ldl_d()*, or *lu_d()* with a matrix that has not been factorize will lead to syntax errors, often not easy to directly relate to the missing factorization.

4.4.2 Avoiding lack of smoothness

In general norm are not smooth:

- The 2-norm is not smooth at the origin because of the *sqrt()*
- The 1-norm is not smooth at any point where one coordinate is equal to zero because of the *abs()*
- The infinity-norm is not smooth at any point where two or more entries have the equal largest absolute values because of the *max()*

For optimizations, this is particular problematic if the optimum occurs precisely at points where the norm is not differentiable, *which is almost always the case for the 1-norm and the infinity norm.*

However, it is generally possible to avoid this problem by introducing auxiliary slack variables and constraints that make the optimization smooth, without losing convexity.

- A minimization involving a 2-norm of the form:

  $$\min\left\{\text{norm}(x,2) + f(x) : x \in \mathbb{R}^n, F(x) \geq 0, G(x) = 0\right\}$$

  can be reformulated as

  $$\min\left\{v + f(x) : v \in \mathbb{R}, x \in \mathbb{R}^n, v > 0, v^2 \geq \text{norm2}(x), F(x) \geq 0, G(x) = 0\right\}$$

- A minimization involving a 1-norm of the form:

  $$\min\left\{\text{norm}(x,1) + f(x) : x \in \mathbb{R}^n, F(x) \geq 0, G(x) = 0\right\}$$

  can be reformulated as

  $$\min\left\{\text{sum}(v) + f(x) : x, v \in \mathbb{R}^n, -v \leq x \leq v, F(x) \geq 0, G(x) = 0\right\}$$

- A minimization involving an infinity-norm of the form:

  $$\min\left\{\text{norm}(x,\text{inf}) + f(x) : x \in \mathbb{R}^n, F(x) \geq 0, G(x) = 0\right\}$$

  can be reformulated as

  $$\min\left\{v + f(x) : v \in \mathbb{R},\ x \in \mathbb{R}^n, -v \leq x \leq v, F(x) \geq 0, G(x) = 0\right\}$$

4.5 Calculus - differentiation

The following function computes the partial derivatives of a symbolic expression with respect to the entries of a tensor-valued symbolic variable:

gradient(*f*, *x*)

Parameters

- **f** (*Tcalculus* symbolic tensor) – tensor-valued expression to be differentiated
- **x** (*Tcalculus* tensor-values symbolic variable) – variable (created with *Tvariable()*) with respect to which the derivatives will be taken

When

- f is a tensor with size [n1,n2,...,nN]
- x is a tensor-valued variable (created with *Tvariable()*) with size [m1,m2,...,mM]

then g=gradient(f,x) results in a tensor with size [n1,n2,...,nN,m1,m2,...,mM] with

$$g(i1, i2, ..., iN, j1, j2, ..., jM) = \frac{d\, f(i1, i2, ..., iN)}{dx(j1, j2, ..., jM)}$$

For example, if f is a scalar (with size []) and x an n-vector (with size [n]), then g=gradient(f,x) results in an n-vector (with size [n]) with the usual gradient:

$$g(i) = \frac{d\, f}{dx(i)}$$

If we then compute h=gradient(g,x), we obtain an n-by-n matrix (with size [n,n]) with the Hessian matrix:

$$h(i, j) = \frac{d\, g(i)}{dx(j)} = \frac{d^2\, f}{dx(i)\, dx(j)}$$

The computation of first and second derivatives can be streamlined using the following function:

hessian(*f*, *x*[, *y*])

Parameters

- **f** (*Tcalculus* symbolic tensor) – tensor-valued expression to be differentiated
- **x** (*Tcalculus* tensor-values symbolic variable) – variable (created with *Tvariable()*) with respect to which the 1st derivatives will be taken
- **y** (*Tcalculus* tensor-values symbolic variable) – variable (created with *Tvariable()*) with respect to which the 2nd derivatives will be taken (optinal, when omitted the 2nd derivatives will also be taken with respect to x)

When

- f is a tensor with size [n1,n2,...,nN]
- x a tensor-valued variable (created with *Tvariable()*) with size [m1,m2,...,mM]

- `y` a tensor-valued variable (created with *Tvariable()*) with size `[l1,l2,...,lL]`

then `[h,g]=hessian(f,x,y)` results in

- a tensor `g` with size `[n1,n2,...,nN,m1,m2,...,mM]` with

$$g(i1,i2,...,iN,j1,j2,...,jM) = \frac{d\,f(i1,i2,...,iN)}{dx(j1,j2,...,jM)}$$

- a tensor `h` with size `[n1,n2,...,nN,m1,m2,...,mM,l1,l2,...,lL]` with

$$h(i1,i2,...,iN,j1,j2,...,jM,k1,k2,...,kL) = \frac{d\,f(i1,i2,...,iN)}{dx(j1,j2,...,jM)dy(j1,j2,...,jM)}$$

In practice, `[h,g]=hessian(f,x,y)` is equivalent to:

```
g=gradient(f,x);
h=gradient(g,y);
```

CHAPTER

FIVE

CONSTRAINED OPTIMIZATION

TensCalc can be used to generate code to solve optimizations or Nash equilibria. All solvers use a primal-dual interior-point method and share several common parameters.

5.1 Types of code

C code

> C code is self-contained and library free (aside from the C standard library). The C code is encapsulated into a dynamic library that can be linked to MATLAB© or used independently.
>
> This type of code is extremely efficient and fast for small to medium-size problems; typically up to a few thousands of variables and constraints. In this domain C code is typically **10-100 times faster.**
>
> In addition to the C code, a MATLAB© class is also generated to set parameter values and call the solver from within MATLAB©.
>
> The scripts that generate C code start with the prefix `cmex2`.

MATLAB©

> MATLAB© code is integrated into a MATLAB© class that is used to set parameter values and call the solver.
>
> The scripts that generate MATLAB© code start with the prefix `class2`.

Note: The MATLAB© classes that call the C and the MATLAB© solvers appear indistinguishable to the user, aside from the speed of execution.

5.2 Minimization

TensCalc can generate optimized code to solve constrained optimizations of the form

$$x^* \in \arg\min_x \left\{ f(x) : F(x) \geq 0, G(x) = 0 \right\}$$

where the variable **x** can include multiple tensors and the equality and inequality constraints can be expressed by equalities and inequalities involving multiple tensors.

The following two scripts are used to generate code to solve this type of constrained minimization

cmex2optimizeCS(*parameter1*, *value1*, *parameter2*, *values2*, ...)

class2optimizeCS(*parameter1*, *value1*, *parameter2*, *values2*, ...)

Parameters

- **parameter1** (`string`) – parameter to set
- **value1** (`type depends on the parameter`) – parameter to set
- **parameter2** (`string`) – parameter to set
- **value2** (`type depends on the parameter`) – parameter to set, ...

Returns
name of the MATLAB© class created

Return type
string

Both scripts take several parameters (many of them optional) in the form of pairs, consisting of a parameter name and its value, as in:

```
classname=cmex2optimizeCS('classname','myoptimization', ...
                          'objective',norm2(x-y), ...
                          'optimizationVariables', { x, y }, ...
                          'constraints', { x>=-1, x<=1, y>=10 }, ...
                          'outputExpressions', { x, y });
```

Alternative, parameters to the scripts can be passed using a structure, as in:

```
opt.classname='myoptimization';
opt.objective=norm2(x-y);
opt.optimizationVariables={ x, y };
opt.constraints={ x>=-1, x<=1, y>=10 };
opt.outputExpressions={ x, y };
classname=cmex2optimizeCS(opt);
```

Note: This type of parameter passing and validation is enabled by the FunParTools toolbox, which also enables several other advanced features. See *Pedigrees*.

The function *cmex2optimizeCS()* generates C code, whereas *class2optimizeCS()* generates MATLAB© code, but both functions take the same set of parameters and generate MATLAB© classes that are indistinguishable to the user.

The following table list the most commonly used parameters for *cmex2optimizeCS()* and *class2optimizeCS()*. For the full set of parameters, use:

```
cmex2optimizeCS help
class2optimizeCS help
```

Table 1: Selected parameters for `cmex2optimizeCS()` and `class2optimizeCS()`. For the full set of parameters use `cmex2optimizeCS help` or `class2optimizeCS help`

Parameter	Allowed values	Description
'optimizationVariables'	cell-array of *Tcalculus* tensor variables created using *Tvariable()*	Variables to be optimized.
'objective'	scalar *Tcalculus* tensor	Criterion to optimize.
'constraints'	cell-array of *Tcalculus* tensors, each involving one of the following operations ==, >=, <=, >, <.	Optimization constraints. **Warning:** For constraint satisfaction, there is no difference between <= and < or between >= and >.
'outputExpressions'	cell-array of *Tcalculus* tensors	Expressions (typically involving the optimization variables) that the solver should return upon termination. **Warning:** MATLAB© does not support sparse arrays with more than 2 dimensions, therefore to include in `outputExpressions` tensors with more than 2 dimensions one must make them *full()*, as in: `cmex2optimizeCS( ↪'classname ↪', ↪'myoptimization`

Warning: *cmex2optimizeCS()* and *class2optimizeCS()* only solve for first-order optimality conditions so they may produce either a minimium or a maximum (or a saddle-point). It is up to the user to restrict the search domain to make sure that the desired optimum was found, or test if that was the case after the fact. The `Hess_` output can help in that regard.

5.2.1 Special variables to include in `'outputExpressions'`

The following *Tcalculus* variables are assigned special values and can be using in `outputExpressions`:

`lambda1_`, `lambda2_`, ...
Lagrangian multipliers associated with the inequalities constraints (in the order that they appear and with the same size as the corresponding constraints)

`nu1_`, `nu2_`, ...
Lagrangian multipliers associated with the equality constraints (in the order that they appear and with the same size as the corresponding constraints)

`Hess_`
Hessian matrix used by the (last) Newton step to update the primal variables (not including addEye2Hessian).

`dHess_`
D factor in the LDL factorization of the Hessian matrix used by the (last) Newton step to update the primal variables (including addEye2Hessian, unlike `Hess_`).

`Grad_`
gradient of Lagrangian at the (last) Newton step.

`mu_`
barrier parameter at the (last) Newton step.

`u_`
vector stacked with all primal variables at the (last) Newton step.

`F_`
vector stacked with all equalities at the (last) Newton step.

`G_`
vector stacked with all inequalities at the (last) Newton step.

`nu_`
vector stacked with all dual equality variables at the (last) Newton step.

`lambda_`
vector stacked with all dual inequality variables at the (last) Newton step.

Warning: To be able to include these variables as input parameters, they have to be previously created using *Tvariable() with the appropriate sizes*. Eventually, their values will be overridden by the solver to reflect the values listed above.

5.2.2 Sensitivity variables

> **Warning:** This section of the documentation is still **incomplete**.

`Tvars2optimizeCS`(*parameter1*, *value1*, *parameter2*, *values2*, ...)

Table 2: Selected parameters for `Tvars2optimizeCS()`. For the full set of parameters use `Tvars2optimizeCS help`

Parameter	Allowed values	Description
'optimizationVariables'	cell-array of *Tcalculus* tensor variables created using *Tvariable()*	Variables to be optimized.
'sensitivityVariables'	cell-array of *Tcalculus* tensor variables created using *Tvariable()*	Optimization variables with respect to which we want to compute cost sensitivity.
'objective'	scalar *Tcalculus* tensor	Criterion to optimize.
'constraints'	cell-array of *Tcalculus* tensors, each involving one of the following operations ==, >=, <=, >, <	Optimization constraints.
'addEye2Hessian'	nonnegative real, default 1e-9	Add to the Hessian matrix appropriate identity matrices scaled by this constant. A larger value for addEye2Hessian has two main effects:
'smallerNewtonMatrix'	[`false`, `true`], default `false`	When `true` the matrix that needs to be inverted to compute a Newton step is reduced by first eliminating the dual variables associated with inequality constraints. However, often the smaller matrix is not as sparse so the computation may actually increase.

5.2.3 Solver Class

The MATLAB© class created to set parameter values and call the solver from within MATLAB© has the following methods:

`obj=classname()`

creates class and, for C code, loads the dynamic library containing the C code

`delete`(*obj*)

deletes the class and, for C code, unloads the dynamic library

`setP_{parameter}(obj,value)`

sets the value of one of the parameters

`setV_{variable}(obj,value)`

sets the value of one of the optimization variables

`[y1,y2, ...]=getOutputs(obj)`

gets the values of the `outputExpressions`

`[status,iter,time]=solve(obj,mu0,int32(maxIter),int32(saveIter))`

Parameters

- **`mu0`** – initial value for the barrier variable
- **`maxIter`** – maximum number of Newton iterations
- **`saveIter`** – iteration # when to save the "hessian" matrix (for subsequent pivoting/permutations/scaling optimization) only saves when allowSave is true.
  - When `saveIter=0`, the hessian matrix is saved at the last iteration; and
  - When `saveIter<0`, the hessian matrix is not saved.

  The "hessian" matrix will be saved regardless of the value of `saveIter`, when the solver exists with `status=4`.

Returns status

solver exist status

- 0 = success
- >0 = solver terminated unexpectedly
- nonzero status indicates the reason for termination in a binary format:
  - bit 0 = 1 - (primal) variables violate constraints
  - bit 1 = 1 - dual variables are negative
  - bit 2 = 1 - failed to invert hessian
  - bit 3 = 1 - maximum # of iterations reached

  when the solver exists because the maximum # of iterations was reached (bit 3 = 1), the remaining bits provide information about the solution returned

- bit 4 = 1 - gradient larger then `gradTolerance`
- bit 5 = 1 - equality constraints violate `equalTolerance`
- bit 6 = 1 - duality gap larger than `desiredDualityGap`
- bit 7 = 1 - barrier variable larger than minimum value
- bit 8 = 1 - scalar gain alpha in Newton direction smaller than `alphaMin`
- bit 9 = 1 - scalar gain alpha in Newton direction smaller than .1
- bit 10 = 1 - scalar gain alpha in Newton direction smaller than .5

Returns iter
number of iterations

Returns time
solver's compute time (in secs).

5.2.4 Pedigrees

Pedigrees can save a lot of time for functions that take some time to execute, like the TensCalc's functions that generate code:

```
|cmex2optimizeCS|      |cmex2equilibriumLatentCS|      |cmex2compute|
|class2optimizeCS|     |class2equilibriumLatentCS|     |class2compute|
```

Essentially, every time a function is executed with pedigrees enables, all its inputs are saved in a *pedigree* file and the function's outputs are also saved. In case the function produces output files (e.g., C or MATLAB© code), the files are stored with unique names for possible subsequent reuse. When the function is called again, it checks whether it has been previously called with *the same exact inputs*:

- If it has, then the previously saved outputs can be retrieved from the appropriate files and the function does not need to be recomputed.
- Otherwise, the function is executed and an additional pedigree and the associated outputs are saved for potential subsequent use.

Pedigrees are enables by specifying the input parameter `'pedigreeClass'`, which is the common prefix used for the names of all the files used to save the pedigree and outputs of the function. Each time the function is called, this prefix is augmented with a unique suffix that reflects the date and time the function was called. To be precise, the *first* date/time the function was called with that particular set of inputs. Any subsequent calls with the same exact inputs reuse the previously saved pedigree.

The following call to *cmex2optimizeCS()* enables the used of pedigrees:

```
classname=cmex2optimizeCS('pedigreeClass','tmp_myopt', ...
                          'executeScript','asneeded',...
                          'objective',norm2(x-y), ...
                          'optimizationVariables', { x, y }, ...
                          'constraints', { x>=-1, x<=1, y>=10 }, ...
                          'outputExpressions', { x, y });
```

and an instance of the solver generated by this call to *cmex2optimizeCS()* can be created using:

```
obj=feval(classname);
```

In the call to *cmex2optimizeCS()*, the parameter pair:

```
'pedigreeClass','tmp_myopt'
```

specifies that pedigrees should be enabled and that all pedigree files should start with the prefix `tmp_myopt`. The *cmex2optimizeCS()* parameter pair:

```
'executeScript','asneeded'
```

further specifies that the *cmex2optimizeCS()* function should only be executed if "it is needed". Specifically, if *cmex2optimizeCS()* has previously been called with the same exact set of inputs, then *cmex2optimizeCS()* should *not* be executed and, instead, the previously generated code should be reused. Alternatively,

```
'executeScript','yes'
```

would specify that the code should be regenerated again *even it cmex2optimizeCS() has been called before with the same exact inputs.*

Warning: When pedigrees are enabled through `'pedigreeClass'` one should **not** specify the class name using the parameter `'classname'`.

The actual classname will be chosen so that it is unique for a specific set of inputs and returned to the user as the output. In this way, regenerating code with a new set of inputs will not overwrite existing code.

Note: Every time a function that uses pedigrees is called with a different set of inputs, a new pedigree is created for potential subsequent reuse. Because of this, one can easily get 100s of pedigree files. The good news is that pedigree files can be safely removed, because they are simply used to save time.

It is a good practice to name all your pedigree files with a unique prefix that marks the file as "safe-to-remove". In all TensCalc examples, we use the prefix `'tmp'` to mark these files, which means that all files started with the 3 letters `'tmp'` are safe to remove.

Note: Pedigrees are stored both as a `.mat` file (for fast retrieval) as well as a human-readable `.html` file. In general, there is little reason to dig into pedigree files, but all the inputs are there in case one is curious about previous calls to code-generation functions.

Pedigrees are enabled by the FunParTools toolbox and are available to any function that uses this toolbox to process input and output parameters.

5.3 Nash equilibrium

TensCalc can generate optimized code to compute Nash equilibrium

$$u^* \in \arg\min_u \left\{ f(u, d^*, x) : F(u, d^*, x) \geq 0, G(u, d^*, x) = 0 \right\}$$
$$d^* \in \arg\min_d \left\{ g(u^*, d, x) : F(u^*, d, x) \geq 0, G(u^*, d, x) = 0 \right\}$$

where the variables `u`, `d`, `x` can include multiple tensors and the equality and inequality constraints can be expressed by equalities and inequalities involving multiple tensors.

The *latent* variable `x` that appears in both minimizations, must be fully determined by the equality constraints and is thus not really a free optimization variable.

The following two scripts are used to generate code to compute this type of Nash equilibrium

cmex2equilibriumLatentCS(*parameter1*, *value1*, *parameter2*, *values2*, ...)

class2equilibriumLatentCS(*parameter1*, *value1*, *parameter2*, *values2*, ...)

Parameters

- **parameter1** (`string`) – parameter to set
- **value1** (`type depends on the parameter`) – parameter to set
- **parameter2** (`string`) – parameter to set
- **value2** (`type depends on the parameter`) – parameter to set, ...

Returns
name of the MATLAB© class created

Return type
string

The function *cmex2optimizeCS()* generates C code, whereas *class2optimizeCS()* generates MATLAB© code, but both functions take the same set of parameters and generate MATLAB© classes that are indistinguishable to the user.

Table 3: Selected parameters for C and `class2equilibriumLatentCS()`. For the full set of parameters use `class2equilibriumLatentCS help` or `class2equilibriumLatentCS help`

Parameter	Allowed values	Description
'P1optimizationVariables' 'P2optimizationVariables'	cell-array of *Tcalculus* tensor variables created using *Tvariable()*	Variables to be optimized by player 1 and player 2
'P1objective' 'P2objective'	scalar *Tcalculus* tensor	Criteria to optimize for player 1 and player 2
'P1constraints' 'P2constraints'	cell-array of *Tcalculus* tensors, each involving one of the following operations ==, >=, <=, >, <	Optimization constraints for player 1 and player 2
'latentVariables'	cell-array of *Tcalculus* tensor variables created using *Tvariable()*	Latent optimization variables common to both players
'latentConstraints'	cell-array of *Tcalculus* tensors, each involving one of the following operations ==, >=, <=, >, <	Optimizations constraints common to both players that implicitely define the latext variables.
'outputExpressions'	cell-array of *Tcalculus* tensors	Expressions (typically involving the optimization variables) that the solver should return upon termination. See *Special variables to include in 'outputExpressions' for Nash solver:*

Warning: For constraint satisfaction, there is no difference between <= and < or between >= and >.

5.3.1 Special variables to include in 'outputExpressions' for Nash solver

The following *Tcalculus* variables are assigned special values and can be using in outputExpressions:

P1lambda1_, P1lambda2_, ... , P2lambda1_, P2lambda2_, ...
: Lagrangian multipliers associated with the inequalities constraints for player 1 and 2 (in the order that they appear and with the same size as the corresponding constraints)

P1nu1_, P1nu2_, ... , P2nu1_, P2nu2_, ... P1xnu1_, P1xnu2_, ... , P2xnu1_, P2xnu2_, ...
: Lagrangian multipliers associated with the equality constraints player 1 and 2 (in the order that they appear and with the same size as the corresponding constraints). The P1x and P2x variables correspond to the latentConstraints.

Hess_
: Hessian matrix used by the (last) Newton step to update the primal variables (not including addEye2Hessian).

Warning: To be able to include these variables as input parameters, they have to be previously created using *Tvariable() with the appropriate sizes*. Eventually, their values will be overridden by the solver to reflect the values listed above.

CHAPTER

SIX

CODE GENERATION FOR COMPUTATIONS

Warning: This section of the documentation is still **incomplete**.

Computation trees

Sets of computations organized as dependency graphs, with multiple roots (input variables) and leaves (output expressions).

6.1 Defining a computation

`class csparse`

`cs=csparse()`

Creates an empty computation tree

Returns

empty *csparse* object

`declareSet`(*cs*, *destination*, *functionName*)

Adds an input variable to the tree

Parameters

- **cs** – *csparse* object
- **destination** – *Tcalculus* tensor variable created using *Tvariable()*
- **functionname** (string) – name of the function to be created

`declareGet`(*cs*, *sources*, *functionName*)

Adds output expressions to the tree

Parameters

- **cs** – *csparse* object
- **sources** – cell array of *Tcalculus* tensor-valued expressions
- **functionname** (string) – name of the function to be created

declareCopy(*cs*, *destinations*, *sources*, *functionName*)

Sets the value of input variables, based on the value of output expressions

Parameters

- **cs** – *csparse* object
- **sources** – cell array of *Tcalculus* tensor-valued expressions
- **destinations** – cell array of *Tcalculus* tensor variables created using *Tvariable()*
- **functionname** (string) – name of the function to be created

6.2 Code generation

cmex2compute(*parameter1*, *value1*, *parameter2*, *values2*, ...)

class2compute(*parameter1*, *value1*, *parameter2*, *values2*, ...)

Generate code to set values of input variables, get values of output expressions, and copy output expressions to input variables.

Parameters

- **parameter1** (string) – parameter to set
- **parameter2** (string) – parameter to set, ...

Returns
name of the MATLAB© class created

Return type
string

- genindex
- search

INDEX

N

P

R

S

T

V