TensCalc

A MATLAB® Toolbox for Nonlinear Optimization Using Symbolic Tensor Calculus

João P. Hespanha

July 5, 2020

Abstract

This tool provides an environment for performing nonlinear constrained optimization. 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. Interior point methods are used for the numerical optimization, which uses formulas for the gradient and the hessian matrix that are computed symbolically in an automated fashion. The package can either produce optimized MATLAB® code or C code. The former is preferable for very large problems, whereas the latter for small to mid-size problems that need to be solved in just a few milliseconds. The C code can be used from inside MATLAB® using an (automatically generated) cmex interface or in standalone applications. No libraries are required for the standalone code.

Contents

1 Quick Start

What are tensors? Tensors are essentially multi-dimensional arrays, but one needs to keep in mind that in MATLAB® every variable is an array of dimension 2 or larger. Unfortunately, this is not suitable for `TensCalc`, which also needs arrays of dimension 0 (i.e., scalars) and 1 (i.e., vectors). This can create confusion because MATLAB® automatically "upgrades" scalars and vectors to matrices (by adding singleton dimensions), but this is not done for `TensCalc` expressions. More on this later, but for now let us keep going with the quick start.

STVEs? The basic objects in `TensCalc` are symbolic tensor-valued expressions (STVEs). These expressions typically involve symbolic variables that can be manipulated symbolically, evaluated for specific values of its variables, and optimized.

Need speed? Prior to numerical optimization, STVEs must be "compiled" for efficient computation. This compilation can take a few seconds or even minutes but results in highly efficient MATLAB® or C code. Big payoffs arise when you need to evaluate or optimize an expression multiple time, for different values of input variables. `TensCalc`'s compilation functions thus always ask you to specify input parameters. Much more on `TensCalc`'s compilations tools can be found in `CSparse`'s documentation.

1.1 The examples you've been waiting for...

Creating STVEs. The following sequence of `TensCalc` command can be used to declare an STVE to be used in a simple least-squares optimization problem

```
N=100;
n=8;

Tvariable A [N,n];
Tvariable b N;
Tvariable x n;

y=A*x-b;
J=norm2(y);
```

Optimizing STVEs. To perform an optimization also need to create an appropriate specialized MATLAB® class, say called `minslsu`, using the following command:

```
class2optimizeCS('classname','minslsu',...
                 'objective',J,...
                 'optimizationVariables',{x},...
                 'outputExpressions',{J,x},...
                 'parameters',{A,b},...
                 'solverVerboseLevel',3);
```

The goal of this class is to minimize the symbolic expression `J` with respect to the variable `x`. The symbolic variables `A` and `b` were declared as parameters that can be changed from optimization to optimization. Setting the `solverVerboseLevel` to 3, asks for a moderate amount of debugging information to be printed while the command is executed (one line per iteration of the solver). More details on the `class2optimizeCS` function can be found in Section 3.1.

Once can see the methods available for the class `minslsu` generated by `class2optimizeCS` using the usual `help` command, which produces:

```
>> help minslsu
  % Create object
  obj=minslsu();
  % Set parameters
  setP_A(obj,{[10000,800] matrix});
  setP_b(obj,{[10000,1] matrix});
  % Initialize primal variables
  setV_x(obj,{[800,1] matrix});
  % Solve optimization
  [status,iter,time]=solve(obj,mu0,int32(maxIter),int32(saveIter));
  % Get outputs
  [y1,y2]=getOutputs(obj);
```

The following commands creates an instance of the class and preforms the optimization for specific parameter values:

```
thisA=rand(N,n);
thisb=rand(N,1);
x0=.02*rand(n,1);

obj=minslsu();
setP_A(obj,thisA);
setP_b(obj,thisb);
setV_x(obj,x0);
mu0=1;
maxIter=20;
[status,iter,time]=solve(obj,mu0,int32(maxIter),int32(-1));
[Justar,xustar]=getOutputs(obj);
```

The parameters `mu0` and `maxIter` passed to the solver are the initial value of the barrier variable and the maximum number of Newton iterations, respectively. On a 2012 MacBook Pro, the solve command above takes about 43ms. More details on the inputs and outputs to the `solve` method can be found in Section 3.1.

If we want instead to perform a constrained optimization, we can use the following command to create the appropriate optimization class:

```
class2optimizeCS('classname','minslsc',...
                 'objective',J,...
                 'optimizationVariables',{x},...
                 'constraints',{x>=0,x<=.05},...
                 'outputExpressions',{J,x},...
                 'parameters',{A,b},...
                 'solverVerboseLevel',3);
```

To perform this optimization one would follow similar steps for the new class:

```
obj=minslsc();
setP_A(obj,thisA);
setP_b(obj,thisb);
setV_x(obj,x0);
[status,iter,time]=solve(obj,mu0,int32(maxIter),int32(-1));
[Justar,xustar]=getOutputs(obj);
```

On a 2012 MacBook Pro, this solve command takes about 195ms.

Turn on the turbo. For really fast results the optimization needs to be done entirely on C, which can be interfaced with MATLAB® using cmex functions. All this is hidden from the user, which simply needs to replace `class2optimizeCS` by `cmex2optimizeCS` to generate the optimization class:

```
cmex2optimizeCS('classname','Cminslsc',...
                'method','primalDual',...
                'objective',J,...
                'optimizationVariables',{x},...
                'constraints',{x>=Tzeros([n]),x<=.05*Tones([n])},...
                'outputExpressions',{J,x},...
                'parameters',{A,b},...
                'solverVerboseLevel',2);
```

The command follows the same syntax, but we decreased the `solverVerboseLevel` to 2, which ask for a small amount of debugging information to be printed while the command is executed (one status line when the solver terminates). More details on the `cmex2optimizeCS` function can be found in Section 3.2.

```
obj=Cminslsc();
setP_A(obj,thisA);
setP_b(obj,thisb);
setV_x(obj,x0);
[status,iter,time]=solve(obj,mu0,int32(maxIter),int32(-1));
[Jcstar,xcstar]=getOutputs(obj);
```

The optimization now takes only 6ms on the same 2012 MacBook Pro.

Need more examples... This and many other examples can be found in `TensCalc`'s `examples` folder.

2 Constructing tensor-valued expressions in `TensCalc`

In `TensCalc`, an *α-index tensor* is an array in $\mathbb{R}^{n_1\times n_2\times\cdots\times n_\alpha}$ where α is an integer in $\mathbb{Z}_{\geqslant 0}$. By convention, the case $\alpha = 0$ corresponds to a *scalar* in $\mathbb{R}$. We use the terminology *vector* and *matrix* for the cases $\alpha = 1$ and $\alpha = 2$, respectively. The integar α is called the *index* of the tensor and the vector of integers $[n_1, n_2, \ldots, n_\alpha]$ (possibly empty for $\alpha = 0$) is called the *dimension* of the tensor.

It needs to be emphasizes that from `TensCalc`'s perspective there is a difference between a scalar in $\mathbb{R}$ (which is a 0-index tensor), a vector with dimension one also in $\mathbb{R}^1$ (which is a 1-index tensor), and a 1×1 matrix in $\mathbb{R}^{1\times 1}$ (which is a 1-index tensor). This distinction is ignored by MATLAB®, which represents scalars and vectors as 2-index matrices, but is important for several tensor operations.

`TensCalc`expressions are constructed using commands similar to the ones used in MATLAB®'s to perform numerical computation.

2.1 `TensCalc` building blocks

The building blocks of STVEs are

symbolic variables that can eventually be replaced by specific numerical values both prior to expression evaluations and prior to expression optimization. A scalar-valued parameter (0-index tensor) is declared using

```
Tvariable parameter_name []
```

and an a-index tensor-valued parameter is declared using

```
Tvariable parameter_name [n1,n2,...,na]
```

for vectors `a=1`, for matrices `a=2`, and for higher-index tensors `na>2`. The integers `n1,n2,...,na` specify the dimension of the tensor. These commands create an STVE with the given name.

special-structure numeric-valued tensors that are recognize by `TensCalc`'s symbolic processing engine, such as tensors with zeros, tensors with ones, or identify tensors. A special-structure tensor is created using any of the following commands

```
var = Tzeros([])
var = Tzeros([n1,n2,...,na])
var = Tones([])
var = Tones([n1,n2,...,na])
var = Teye([])
var = Teye([n1,n2,...,na,n1,n2,...,na])
```

The integers `n1,n2,...,na` specify the dimension of each index of the tensor. Note that an identity tensor is expected to have the first half of the dimensions equal to the second half.

unstructured numeric-valued tensors for which `TensCalc`'s symbolic processing engine only takes into account their sparcity pattern. Unstructured tensors are created using

```
var = Tconstant(numeric_expression)
```

where `numeric_expression` is any valid numeric matlab expression. The above command will actually attempt to find any special structure in `numeric_expression` and returns a special-structure tensor if it detects a tensor with zeros, a tensor with ones, or an identify matrix (2-index).

Because of the different approach used by MATLAB® and `TensCalc` to represent scalars and vectors, the following rules are used to determine the size of the tensor returned:

1. 1×1 MATLAB® matrices are converted to scalars (0-index tensors);
2. $n \times 1$ with $n > 1$ MATLAB® matrices are converted to vectors (1-index tensors);
3. all other MATLAB® matrices (including row-vectors) are kept with the same dimension.

When these rules need to be overwritten one can use

```
var = Tconstant(numeric_expression,[n1,n2,...,na])
```

where the integers `n1,n2,...,na` specify the dimension of each index of the tensor.

When numeric MATLAB® expressions appear within STVEs, they are automatically converted into special structure or unstructured numeric-valued tensors using `Tconstant` and the rules above to determine the tensor size. An exception to this rule arises when numeric MATLAB® expressions appear within the STVE functions `vertcat` and `horzcat`, in which case the values are kept as matrices (2-indexes).

2.2 STVE operators

The building blocks described above can be combined into complex mathematical expressions using the following operators:

(.)/subsref The following command returns a selected set of entries of `stve`:

```
subsref(stve,vec1,vec2,...,vecn)
stve(vec1,vec2,...,vecn)
```

Either of these commands return a subtensor of `stve` consisting of the entries specified by the vectors of indices `vec1,vec2,...,vecn`. The subtensor returned has the same index as `stve`, even if the vectors of indices are singletons, e.g., `stve(1,1)` is still a 2-index tensor.

As in regular matlab, one can use the keyword `end` to construct any of the vectors `vec1,vec2,...,vecn`, e.g., as in `stve(3,2:end-1)`.

uplus The following command returns `stve`:

```
uplus(stve)
+stve
```

uminus The following command returns `-stve`:

```
uminus(stve)
-stve
```

ctranspose/transpose Any of the following command returns the transpose of `stve`:

```
ctranspose(stve)
tranpose(stve)
stve'
stve.'
```

Note that all `TensCalc` tensors are real-values so `transpose` and `ctranspose` return the same object.

Attention! Transposes only make sense for 2-index tensors (matrices) and the flexibility provided by the tensor product makes the use of transposes unnecessary so *this operator should not be used*. In fact, `TensCalc` is unable to compute gradients of expressions involving transposes[1].

plus The following command returns the sum of `stve1` and `stve2`:

```
stve1+stve2
plus(stve1,stve2)
```

The two STVEs must have the same dimension or one of them must be a scalar (0-index tensor). In the latter case, the scalar is added to each entry of the other tensor[2].

minus The following command returns the difference between `stve1` and `stve2`:

```
stve1-stve2
minus(stve1,stve2)
```

[1] Draft note: One could compute the gradient precisely by converting `ctranspose` to `tprod`

[2] In practice, the scalar is multiplied by a tensor of ones of appropriate size.

The two STVEs must have the same dimension or one of them must be a scalar (0-index tensor). In the latter case, the scalar is added to each entry of the other tensor[3].

ge, >= The following command returns 1 (true) is `stve1>=stve2` and 0 (false) otherwise:

```
stve1>=stve2
ge(stve1,stve2)
```

> The following command returns 1 (true) is `stve1>stve2` and 0 (false) otherwise:

```
stve1>stve2
```

le, <= The following command returns 1 (true) is `stve1<=stve2` and 0 (false) otherwise:

```
stve1<=stve2
le(stve1,stve2)
```

< The following command returns 1 (true) is `stve1<stve2` and 0 (false) otherwise:

```
stve1<stve2
```

tprod The following command returns the `(p1,p2,p3,...)`-product of `stve1`, `stve2`, `stve3`, ...

```
tprod(stve1,p1,stve2,p2,stve3,p3,...)
```

times The following command returns the entry-wise multiplication of `stve1` and `stve2`:

```
stve1.*stve2
times(stve1,stve2)
```

This operation is automatically converted to an equivalent `tprod`.

mtimes The following command returns the matrix/vector multiplication of `stve1` and `stve2`:

```
stve1*stve2
mtimes(stve1,stve2)
```

The dimensions of `stve1` and `stve2` must be one of the following

1. both matrices (2-index tensors), returning the usual matrix product;
2. `stve1` a matrix (2-index tensor) and `stve2` a vector (1-index tensor), returning the usual matrix by column-vector product;
3. `stve1` a vector (1-index tensor) and `stve2` a matrix (2-index tensor), returning the usual row-vector by matrix product;
4. both vectors (1-index tensors), returns the inner product of the two vectors.
5. any of them a scalar (0-index) and the other any tensor, returning the usual scalar-by-matrix product.

This operation is automatically converted to an equivalent `tprod`.

rdivide The following command returns the entry-wise right division of `stve1` and `stve2`:

[3]In practice, the scalar is multiplied by a tensor of ones of appropriate size.

```
stve1./stve2
rdivide(stve1,stve2)
```

Attention! Currently, `TensCalc` is unable to compute gradients of expressions involving `rdivide`.

mldivide The following commands are inspired by MATLAB®'s `mldivide` command and returns the left matrix division of `stve1` and `stve2`:

```
stve1\stve2
mldivide(stve1,stve2)
```

Attention! Currently, `TensCalc` is unable to compute gradients of expressions involving `mldivide`.

Attention! The following notable operators are currently not implemented `rdivide/./`, `ldivide/.\`, `mrdivide//`, `mldivide/\`, `power/.^`. `mpower/^`.

2.3 STVE functions

The building blocks described above can be combined into complex mathematical expressions using the following functions:

reshape The following commands are inspired by MATLAB®'s `reshape` command:

```
reshape(stve,[n1 n2 ... na])
reshape(stve,n1,n2,...,na)
```

These commands does not alter the entries of `stve`, but changes its dimension to `[n1 n2 ... na]`. The total number of entries of `stve` cannot change.

repmat The following command is inspired by MATLAB®'s `repmat` command:

```
repmat(stve,[m1 m2 ... ma])
```

This command tiles `stve` to produce a tensor formed by taking multiple copies of `stve`. When the dimension of `stve` is equal to `[n1 n2 ... na]`, the size of the tensor returned is equal to

```
[n1*m1 n2*m2 ... na*ma]
```

Attention! This command can be emulated through appropriate tensor multiplications by tensors with ones. While this is computationally more expensive, `TensCalc`'s symbolic engine understands the equivalence between these two approaches to tiling and actually uses `repmat` to replace some multiplications by tensors with ones. Currently, `TensCalc` is unable to compute gradients of expressions involving this command[4].

cat The following commands are inspired by MATLAB®'s `cat` command.

```
cat(dim,stve1,stve2,...,stven)
```

This command concatenates multiple arrays along the dimension `dim`. All the arrays to be concatenated must have matching sizes along all dimensions other than the one that along which they are being concatenated.

[4]Draft note: One could compute the gradients precisely by converting `repmat` to `tprod`

vertcat The following commands are inspired by MATLAB®'s `vertcat` command.

```
[stve1;stve2;...;stven]
vertcat(stve1,stve2,...,stven)
cat(1,stve1,stve2,...,stven)
```

This command concatenates multiple arrays along the 1st dimension. All the arrays to be concatenated must have matching sizes along all dimensions other than the one that along which they are being concatenated.

horzcat The following commands are inspired by MATLAB®'s `horzcat` command.

```
[stve1,stve2,...,stven]
horzcat(stve1,stve2,...,stven)
cat(2,stve1,stve2,...,stven)
```

This command concatenates multiple arrays along the 2nd dimension. All the arrays to be concatenated must have matching sizes along all dimensions other than the one that along which they are being concatenated.

sum The following command is inspired by MATLAB®'s `sum` command:

```
sum(stve,dim)
```

This command produces a tensor with one index less than `stve` by adding the entries of the tensor along that index:

Note: This command is emulated through an appropriate tensor multiplication.

diag The following command is inspired by MATLAB®'s `diag` command:

```
diag(stve)
```

When `stve` is a vector (1-index tensor), a square matrix (2-index tensor) is returned with `stve` in the main diagonal. When `stve` is a square matrix, a vector is returned containing the main diagonal of `stve`.

Attention! This command only makes sense for 1- and 2-index tensors and the flexibility provided by the tensor product makes the use of this command unnecessary so this *function should not be used.* In fact, `TensCalc` is unable to compute gradients of expressions involving this command[5].

norm2 The following command computes the squared Frobenius norm of a tensor (i.e., the sum of the squares of all its entries):

```
norm2(stve)
```

chol The following command computes the lower-triangular Cholesky factorization of a symmetric positive define matrix (2-index tensor):

```
chol(A)
```

[5]Draft note: One could compute the gradient precisely by converting `diag` to `tprod`

This command returns a lower-triangular matrix `L` so that `A=L'*L`, much like MATLAB®'s `chol(A,'lower')`.

In C-compiled code, the matrix first undergoes a row/column permutation computed using MATLAB®'s command `symamd` to maximize the sparsity of the Cholesky factor. This permutation is taken into account by any `pptrs` function that uses a matrix computed by `chol`.

Attention! `TensCalc` is unable to compute gradients of expressions involving this function.

pptrs The following command computes solution to a lower-triangular system of linear equations `A x=b`:

```
pptrs(L,b)
```

where `L` is the lower triangular Cholesky factor of `A`, as computed by `chol(A)`, i.e., `A=L'*L`.

In C-compile code, `pptrs` is aware that `chol(A)` may permute rows/columns to maximize sparsity and "undoes" the permutation to produce the solution to the original system of equations

Attention! `TensCalc` is unable to compute gradients of expressions involving this function.

splineDeriv/splineDDeriv The following commands returns estimates of the 1st and 2nd derivative of the time series `stve`, computed based on a 2nd order spline.

```
dstve=splineDeriv(stve)
ddstve=splineDDeriv(stve)
```

The input `stve` should be a vector (1-index tensor) with dimension n, representing a time series at times 1 through n, and the output is a vector (1-index tensor) with dimension $n-2$, representing a time series of 1st/2nd derivatives at times 2 through $n-1$.

Note: A 2nd order spline

$$x(t-k)=\frac{a}{2}(t-k)^2+v(t-k)+x(k),\quad \forall t\geqslant 0$$

passing through the three points $x(k-1), x(k), x(k+1)$, satisfies

$$\begin{bmatrix}\frac{1}{2} & -1\\ \frac{1}{2} & +1\end{bmatrix}\begin{bmatrix}a\\ v\end{bmatrix}=\begin{bmatrix}x(k-1)-x(k)\\ x(k+1)-x(k)\end{bmatrix}\quad\Leftrightarrow\quad\begin{bmatrix}a\\ v\end{bmatrix}=\begin{bmatrix}x(k-1)-2x(k)+x(k+1)\\ \frac{x(k+1)-x(k-1)}{2}\end{bmatrix}$$

Therefore

```
splineDeriv(stve)=.5*(stve(3:end)-stve(1:end-2));
splineDDeriv(stve)=stve(3:end)-2*stve(2:end-1)+stve(1:end-2);
```

compose The following command returns an STVE that results from applying the scalar function `f` to every entry of `stve`, returning an STVE of the same size[6]:

```
compose(stve,f,df,ddf, ...)
```

`f` is a MATLAB® handle to a function that maps scalars to scalars, `df` a handle to its 1st derivative, `ddf` a handle its 2nd derivative, and so on. The handles to the derivatives are optional, but are typically needed to compute gradients and hessian matrices needed to perform optimizations.

[6] `compose` actually allows the function to be tensor valued, in which case the dimensions of `f` are appended to the dimensions of `stve`.

All the functions referenced must be vectorizable, i.e., they must be able to take tensors as inputs and return a tensor of the same size by applying the function entry by entry.

The following matlab functions have been redefined to perform composition in a transparent way: `exp`, `log`, `square`, `sqrt`, `cos`, `sin`, `tan`, `normpdf`[7]. For example `exp(stve)` has been redefined as `compose(stve,@(x)exp(x),@(x)exp(x),@(x)exp(x))`.

gradient The following command computes the gradient of `stve` with respect to the `TensCalc` variable `var`:

```
gradient(stve,var)
```

Attention! `TensCalc` is not able to compute gradients of `TensCalc` expression involving: `ctranspose`, `transpose`, `rdivide`, `mldivide`, `repmat`, `diag`, `chol`, `pptrs`, `clp`.

hessian The following command computes the hessian of `stve` with respect to the `TensCalc` variables `var1,var2`:

```
hessian(stve,var1,var2)
```

substitute The following command replaces the variable `var` by the expression `stve2`, in the expression `stve1`:

```
substitute(stve1,var,stve2)
```

clp The following command solves the following canonical linear program $\max\{\alpha > 0 : \alpha \text{ stve1} + \text{stve2} \geqslant 0\}$, where `stve1` and `stve2` are 2 tensors with the same dimension:

```
clp(stve1,stve2)
```

interpolate The following command returns the value of a function defined through an interpolation table:

```
interpolate(X,Xi,Yi,S,method)
```

For a function $Y = F(X)$ that maps α-tensors X with dimension $[n_1, n_2, \dots, n_\alpha]$ into β-tensors Y with dimension $[m_1, m_2, \dots, m_\beta]$, an interpolation table with K entries is represented by the

1. the $\alpha + 1$-tensor with dimension $[n_1, n_2, \dots, n_\alpha, K]$, and
2. the $\beta + 1$-tensor with dimension $[m_1, m_2, \dots, m_\beta, K]$,

with the understanding that, for each $k \in \{1, 2, \dots, K\}$,

```
F( Xi(:,:,\dots,:,k) ) = Yi(:,:,\dots,:,k).
```

The interpolation method is specified by the string in the 5th parameter and can take the following values:

1. When `method='ugaussian'`, we have

$$F(X) = \sum_{k=1}^{K} Yi(:, \dots, :, k) e^{-\frac{1}{2S^2}\|Xi(:,\dots,:,k) - X\|^2}$$

[7] $\texttt{Tnormpdf}(x) := \frac{e^{-x^2/2}}{\sqrt{2\pi}}$.

2. When `method='ngaussian'`, we have

$$F(X)=\frac{\sum_{k=1}^{K} Yi(:,\dots,:,k)e^{-\frac{1}{2S^2}\|Xi(:,\dots,:,k)-X\|^2}}{\sum_{k=1}^{K} e^{-\frac{1}{2S^2}\|Xi(:,\dots,:,k)-X\|^2}}$$

Ginterpolate The following command returns the gradient of the function `interpolate(X,Xi,Yi,S,method)` with respect to `X`:

```
Ginterpolate(X,Xi,Yi,S,method)
```

Ginterpolate The following command returns the hessian of the function `interpolate(X,Xi,Yi,S,method)` with respect to `X`:

```
Hinterpolate(X,Xi,Yi,S,method)
```

Attention! The following notable functions are currently not implemented `inv`, `pinv`, `det`, `expm`, `logm`, `sqrtm`, `logdet`.

2.4 Examples

Example 1. Suppose that we have defined 4 vectors $x,z\in\mathbb{R}^N$, $a,b\in\mathbb{R}^n$ and two scalars functions $f,g:\mathbb{R}\to\mathbb{R}$ as STVE objects.

1. To compute $y\in\mathbb{R}^N$ defined by

$$y_k=\sum_{i=1}^{n}a_i f(b_i+x_k)$$

one could use

```
Xx=tprod(x,1,Tones(n),2); % expands x into an N x n matrix
Xb=tprod(Tones(N),1,b,2); % expands b into an N x n matrix
fxb=compose(f,Xx+Xb);
y=tprod(fxb,[1,-1],a,-1);
```

2. To compute $y\in\mathbb{R}^N$ defined by

$$y_k=\sum_{i=1}^{n}a_i f(b_i+c_i x_k)$$

one could use

```
Xb=tprod(Tones(N),1,b,2); % expands b into an N x n matrix
Xcx=tprod(x,1,c,2);       % expands c*x into an N x n matrix
fbcx=compose(f,Xb+Xcx);
y=tprod(fbcx,[1,-1],a,-1);
```

3. To compute $y\in\mathbb{R}^N$ defined by

$$y_k=\sum_{i=1}^{n}a_i f(b_i+x_k)g(c_i+z_k)$$

one could use

```
Xx=tprod(x,1,Tones(n),2); % expands x into an N x n matrix
Xz=tprod(z,1,Tones(n),2); % expands z into an N x n matrix
Xb=tprod(Tones(N),1,b,2); % expands b into an N x n matrix
Xc=tprod(Tones(N),1,c,2); % expands c into an N x n matrix
fbx=compose(f,Xb+Xx);
gcz=compose(f,Xc+Xz);
y=tprod(fbx,[1,-1],gcz,[1,-1],a,-1);
```

3 Constrained optimization of tensor-valued functions

The functions `class2optimize` and `cmex2optimize` are both used to solve optimization problems of the form

$$J(x^*; p) = \quad \text{minimum} \quad J(x; p) \tag{1}$$
$$\text{subject to} \quad F(x; p) \geqslant 0, \;\; G(x; p) = 0 \tag{2}$$

where $J(\cdot)$ is a scalar-valued variable, $F(\cdot)$ and $G(\cdot)$ are tensor-valued variables, x represents the variables to be optimized, p denote fixed parameters, and $F(x; p) \geqslant 0$ should be understood as an constraint on every entry of $F(x; p)$. The two functions `class2optimize` and `cmex2optimize` have the same syntax, but differ in several key aspects:

solver speed The solver produced by `cmex2optimize` is implemented in C and is typically much faster than the MATLAB® solver produced by `cmex2optimize`.

code size The MATLAB® code produced by `class2optimize` is mostly independent of the size of the problem, whereas the C code produced by `cmex2optimize` grows with the size of the problem (often linearly, but many times worst than that).

compilation speed The generation of MATLAB® code by `class2optimize` is typically fast, whereas `cmex2optimize` may take a while to generate code for large problems. Note that `cmex2optimize` not only generates the optimization code, but also generate cmex wrappers and compiles the whole thing using some form of compiler optimization (typically `-O1`). Often the slowest part is the cmex compilation with optimization.

3.1 Using a MATLAB® class

The function `class2optimizeCS` creates a MATLAB®class to solve optimizations of the form (1):

```
[...]=class2optimizeCS('parameter name 1',value,'parameter name 2',value,...);
```

creates a matlab class for solving optimization problems of the form:

```
 objective(optimizationVariables*,parameters) =
      = minimum     objective(optimizationVariables,parameters)
        w.r.t.      optimizationVariables
        subject to  constraints(optimizationVariables,parameters)
```
and returns
```
 outputExpressions(optimizationVariables extasciicircum*,parameters)
```
The solver is accessed through a matlab class. See `ipm.pdf` for details of the optimization engine.

Input parameters:

- `verboseLevel` [default 0]

  Level of verbose for debug outputs (0 - for no debug output)

- `parametersStructure` [default '']

  Structure whose fields are used to initialize parameters not present in the list of parameters passed to the function. This structure should contains fields with names that match the name of the parameters to be initialized.

- `pedigreeClass` [default '']

  When nonempty, the function outputs are saved to a file set. All files in the set will be characterized by a 'pedigree', which decribes all the input parameters that were used in the script. This variable contains the name of the file class and may include a path. See also createPedigree

- `executeScript` [default `'yes'`] taking values in [`'yes'`,`'no'`,`'asneeded'`]

  Determines whether or not the body of the function should be executed:

  - `yes` - the function body should always be executed.
  - `no` - the function body should never be executed and therefore the function returns after processing all the input parameters.
  - `asneeded` - if a pedigree file exists that match all the input parameters (as well as all the parameters of all 'upstream' functions) the function body is not executed, otherwise it is execute.

- `classname` [default `<to be determined from the pedigree >`]

  Name of the class to be created. A matlab class will be created with this name plus a `.m` extension. The class will have the following methods:

  - `obj=classname()` - creates class
  - `delete(obj)` - deletes the class
  - `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
  - `[status,iter,time]=solve(obj,mu0,int32(maxIter),int32(saveIter))`

  where

  - `mu0` - initial value for the barrier variable
  - `maxIter` - maximum number of Newton iterations
  - `saveIter` - Parameter not used (included for compatibility with `cmex2optimizeCS`).
  - `status` - solver exist status
    * 0 = success
    * -1 = maximum # of iterations reached
    * -2 = failed to invert hessian
    * -3 = (primal) variables violate constraints

        * -4 = negative value for dual variables
    - `iter` - number of iterations
    - `time` - solver's compute time (in secs).

  One can look "inside" this class to find the name of the cmex functions.

- `folder` [default `'.'`]

  Path to the folder where the files will be created. Needs to be in the Matlab path.

- `objective`

  Scalar TC symbolic object to be optimized.

- `optimizationVariables`

  Cell-array of TC symbolic objects representing the variables to be optimized.

- `constraints` [default `{}`]

  Cell-array of TC symbolic objects representing the constraints. Both equality and inequality constraints are allowed.

- `parameters` [default `{}`]

  Cell-array of TC symbolic objects representing the parameters (must be given, not optimized).

- `outputExpressions` [default `{}`]

  Cell-array of TC symbolic objects representing the variables to be returned.

  The following TC symbolic variables are assigned special values and can be using in outputExpressions

    - `lambda0_`,`_lambda1_`,... - Lagrangian multipliers associated with the inequalities constraints (in the order that they appear and with the same size as the corresponding constraints)
    - `nu0_`,`nu1_`,... - 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`).

      ATTENTION: To be able to include these variables as input parameters, they will have to be created outside this function **with the appropriate sizes**. Eventually, their values will be overridden by the solver to reflect the values above.

- `method` [default `'primalDual'`] taking values in [`'primalDual'`]

  Variable that specifies which method should be used:

    - `primalDual` - interior point primal-dual method
    - `barrier` - interior point barrier method (not yet implemented)

- `alphaMin` [default `1e-07`]

  Minimum value for the scalar gain in the line search below which a search direction is declared to have failed.

- `alphaMax` [default `1`]

  Maximum value for the scalar gain in the line search. Should only be set lower to 1 for very poorly scaled problems.

- `skipAffine` [default `false`] taking values in [`false`,`true`]

  When `true` the affine search direction step is omitted.

- `delta` [default `3`] taking values in [2,3]

  Delta parameter used to determine mu based on the affine direction. Set `Delta=3` for well behaved problems (for an aggressive convergence) and `Delta=2` in poorly conditioned problems (for a more robust behavior). This parameter is only used when `skipAffine=false`.

- `muFactorAggressive` [default `0.333333`]

  Multiplicative factor used to update the barrier parameter (must be smaller than 1). This value is used when there is good progress along the Newton direction. Nice convex problems can take as low as 1/100, but poorly conditioned problems may require as high as 1/3. This parameter is only used when `skipAffine=true`.

- `muFactorConservative` [default `0.75`]

  Multiplicative factor used to update the barrier parameter (must be smaller than 1). This value is used when there is poor or no progress along the Newton direction. A value not much smaller than one is preferable.

- `gradTolerance` [default `0.0001`]

  Maximum norm for the gradient below which the first order optimality conditions assumed to by met.

- `equalTolerance` [default `0.0001`]

  Maximum norm for the vector of equality constraints below which the equalities are assumed to hold.

- `desiredDualityGap` [default `1e-05`]

  Value for the duality gap that triggers the end of the constrained optimization. The overall optimization terminates at the end of the first Newton step for which the duality gap becomes smaller than this value.

- `addEye2Hessian` [default `0`]

  Add to the Hessian matrix appropriate identity matrices scaled by this constant.

- `scratchbookType` [default `'double'`] taking values in [`'double'`]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `codeType` [default `'C'`] taking values in [`'C'`,`'C+asmSB'`,`'C+asmLB'`]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `compilerOptimization` [default `'-O1'`] taking values in [`'-O0'`,`'-O1'`,`'-O2'`,`'-O3'`,`'-Ofast'`]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `callType` [default `'dynamicLibrary'`] taking values in [`'dynamicLibrary'`,`'client-server'`]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `serverProgramName` [default ''"]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `serverAddress` [default `'localhost'`]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `port` [default 1968]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `targetComputer` [default `'maci64'`] taking values in [`'maci64'`,`'glnxa64'`]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `solverVerboseLevel` [default 1]

  Level of verbose for the solver outputs:

  - 0 - the solver does not produce any output
  - 1 - the solver only report a summary of the solution status when the optimization terminates
  - 2 - the solver reports a summary of the solution status at each iteration
  - $>$2 - the solver produces several (somewhat unreadable) outputs at each iteration step

- `debugConvergence` [default `false`] taking values in [`true`,`false`]

  Includes additional output to help debug failed convergence.

- `debugConvergenceThreshold` [default 100000]

  Threshold above which solves warns about large values. Only used when `debugConvergence=true`

- `allowSave` [default `false`] taking values in [`true`,`false`]

  Parameter not used (included for compatibility with `cmex2optimizeCS`).

- `profiling` [default `false`] taking values in [`true`,`false`]

  When nonzero, adds profiling to the C code.

Outputs:

- `classname`

  Name of the class created.

3.2 Using C code

The function `cmex2optimizeCS` creates C code to solve optimizations of the form (1), using a syntax similar to `class2optimizeCS`:

```
[...]=cmex2optimizeCS('parameter name 1',value,'parameter name 2',value,...);
```

creates a matlab class for solving optimization problems of the form:

```
objective(optimizationVariables*,parameters) =
     = minimum     objective(optimizationVariables,parameters)
       w.r.t.      optimizationVariables
       subject to  constraints(optimizationVariables,parameters)
```

and returns

```
outputExpressions(optimizationVariables extasciicircum*,parameters)
```

The solver is accessed through several cmex functions that can be accessed directly or through a matlab class. See `ipm.pdf` for details of the optimization engine.

Input parameters:

- `verboseLevel` [default 0]

  Level of verbose for debug outputs (0 - for no debug output)

- `parametersStructure` [default '']

  Structure whose fields are used to initialize parameters not present in the list of parameters passed to the function. This structure should contains fields with names that match the name of the parameters to be initialized.

- `pedigreeClass` [default '']

  When nonempty, the function outputs are saved to a file set. All files in the set will be characterized by a 'pedigree', which decribes all the input parameters that were used in the script. This variable contains the name of the file class and may include a path. See also createPedigree

- `executeScript` [default `'yes'`] taking values in [`'yes'`,`'no'`,`'asneeded'`]

  Determines whether or not the body of the function should be executed:

  – `yes` - the function body should always be executed.

  – `no` - the function body should never be executed and therefore the function returns after processing all the input parameters.

  – `asneeded` - if a pedigree file exists that match all the input parameters (as well as all the parameters of all 'upstream' functions) the function body is not executed, otherwise it is execute.

- `classname` [default `<to be determined from the pedigree >`]

  Name of the class to be created. A matlab class will be created with this name plus a `.m` extension. The class will have the following methods:

  – `obj=classname()` - creates class and loads the dynamic library containing the C code

  - `delete(obj)` - deletes the class and unload 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
  - `[status,iter,time]=solve(obj,mu0,int32(maxIter),int32(saveIter))`

  where

  - `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=-2`
  - `status` - solver exist status
    * 0 = success
    * -1 = maximum # of iterations reached
    * -2 = failed to invert hessian
    * -3 = (primal) variables violate constraints
    * -4 = negative value for dual variables
  - `iter` - number of iterations
  - `time` - solver's compute time (in secs).

  One can look "inside" this class to find the name of the cmex functions.

- `folder` [default `'.'`]

  Path to the folder where the class and cmex files will be created. The folder will be created if it does not exist and it will be added to the begining of the path if not there already.

- `objective`

  Scalar TC symbolic object to be optimized.

- `optimizationVariables`

  Cell-array of TC symbolic objects representing the variables to be optimized.

- `constraints` [default `{}`]

  Cell-array of TC symbolic objects representing the constraints. Both equality and inequality constraints are allowed.

- `parameters` [default `{}`]

  Cell-array of TC symbolic objects representing the parameters (must be given, not optimized).

- `outputExpressions` [default `{}`]

  Cell-array of TC symbolic objects representing the variables to be returned.

  The following TC symbolic variables are assigned special values and can be using in outputExpressions

  - `lambda0_`,`_lambda1_`,... - Lagrangian multipliers associated with the inequalities constraints (in the order that they appear and with the same size as the corresponding constraints)
  - `nu0_`,`nu1_`,... - 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`).

    ATTENTION: To be able to include these variables as input parameters, they will have to be created outside this function **with the appropriate sizes**. Eventually, their values will be overridden by the solver to reflect the values above.

- `method` [default `'primalDual'`] taking values in [`'primalDual'`]

  Variable that specifies which method should be used:

  - `primalDual` - interior point primal-dual method
  - `barrier` - interior point barrier method (not yet implemented)

- `alphaMin` [default `1e-07`]

  Minimum value for the scalar gain in the line search below which a search direction is declared to have failed.

- `alphaMax` [default `1`]

  Maximum value for the scalar gain in the line search. Should only be set lower to 1 for very poorly scaled problems.

- `skipAffine` [default `false`] taking values in [`false`,`true`]

  When `true` the affine search direction step is omitted.

- `delta` [default 3] taking values in [2,3]

  Delta parameter used to determine mu based on the affine direction. Set `Delta=3` for well behaved problems (for an aggressive convergence) and `Delta=2` in poorly conditioned problems (for a more robust behavior). This parameter is only used when `skipAffine=false`.

- `muFactorAggressive` [default `0.333333`]

  Multiplicative factor used to update the barrier parameter (must be smaller than 1). This value is used when there is good progress along the Newton direction. Nice convex problems can take as low as 1/100, but poorly conditioned problems may require as high as 1/3. This parameter is only used when `skipAffine=true`.

- `muFactorConservative` [default `0.75`]

  Multiplicative factor used to update the barrier parameter (must be smaller than 1). This value is used when there is poor or no progress along the Newton direction. A value not much smaller than one is preferable.

- `gradTolerance` [default `0.0001`]

  Maximum norm for the gradient below which the first order optimality conditions assumed to by met.

- `equalTolerance` [default `0.0001`]

  Maximum norm for the vector of equality constraints below which the equalities are assumed to hold.

- `desiredDualityGap` [default `1e-05`]

  Value for the duality gap that triggers the end of the constrained optimization. The overall optimization terminates at the end of the first Newton step for which the duality gap becomes smaller than this value.

- `addEye2Hessian` [default `0`]

  Add to the Hessian matrix appropriate identity matrices scaled by this constant.

- `scratchbookType` [default `'double'`] taking values in [`'double'`,`'float'`]

  C variable type used for the scratchbook.

- `codeType` [default `'C'`] taking values in [`'C'`,`'C+asmSB'`,`'C+asmLB'`]

  Type of code produced:

  - `C` - all computations done in pure C code. Ideal for final code.

    Impact on non-optimized compilation:

    * medium compilation times
    * largest code size
    * slowest run times

    Impact on optimized code: Gives the most freedom to the compiler for optimization

    * slowest compile optimization times
    * fastest run times
    * smallest code sizes

  - `C+asmLB` - little C code, with most of the computations done by large blocks of inlined assembly code. Ideal for testing.

    Impact on non-optimized compilation:

    * fastest compilation times
    * smallest code size
    * fastest run times (for non-optimized code)

    Impact on optimized compilation: Most of the compiler optimization is restricted to re-ordering and/or inlining the large blocks of asm code

    * fastest compile optimization times
    * slowest run times
    * largest optimized code sizes (due to inlining large blocks)

    NOT FULLY IMPLEMENTED.

  - C+asmSB – little C code, with most of the computations done by small blocks of inlined assembly code

    Impact on non-optimized compilation:

    * medium compilation times
    * medium code size
    * medium run times

    Impact on optimized code: Most of the compiler optimization is restricted to re-ordering and/or inlining the small blocks of asm code

    * medium compile optimization times,
    * medium run times
    * medium code sizes

    NOT FULLY IMPLEMENTED NOR TESTED.

- `compilerOptimization` [default `'-O1'`] taking values in [`'-O0'`,`'-O1'`,`'-O2'`,`'-O3'`,`'-Ofast'`]

  Optimization parameters passed to the C compiler.

  - `-O1` often generates the fastest code, whereas
  - `-O0` compiles the fastest

- `callType` [default `'dynamicLibrary'`] taking values in [`'dynamicLibrary'`,`'client-server'`]

  Method used to interact with the solver:

  - `dynamicLibrary` – the solver is linked with matlab using a dynamic library
  - `client-server` – the solver runs as a server in independent process, and a socket is used to exchange data.

- `serverProgramName` [default '']

  Name of the executable file for the server executable. This parameter is used only when `callType='client-server'`.

- `serverAddress` [default `'localhost'`]

  IP address (or name) of the server. This parameter is used only when `callType='client-server'`.

- `port` [default 1968]

  Port number for the socket that connects client and server. This parameter is used only when `callType='client-server'`.

- `compileGateways` [default `true`] taking values in [`true`,`false`]

  When `true` the gateway functions are compiled using `cmex`.

- `compileLibrary` [default `true`] taking values in [`true`,`false`]

  When `true` the dynamicLibrary is compiled using gcc. This parameter is used only when `callType='dynamicLibrary`

- `compileStandalones` [default `true`] taking values in [`true`,`false`]

  When `true` the standalone/server executable is compiled using gcc. This parameter is used only when `callType` has one of the values: `'standalone'` or `'client-server'`

- `compilerOptimization` [default `'-Ofast'`] taking values in [`'-O0'`,`'-O1'`,`'-O2'`,`'-O3'`,`'-Ofast'`]

  Optimization flag used for compilation. Only used when either `compileGateways`, `compileLibrary`, or `compileStandalones` are set to `true`.

- `targetComputer` [default `'maci64'`] taking values in [`'maci64'`,`'glnxa64'`]

  OS where the mex files will be compiled.

- `serverComputer` [default `'maci64'`] taking values in [`'maci64'`,`'glnxa64'`]

  OS where the server will be compiled. This parameter is used only when `callType='client-server'`.

- `solverVerboseLevel` [default 1]

  Level of verbose for the solver outputs:

  - 0 - the solver does not produce any output
  - 1 - the solver only report a summary of the solution status when the optimization terminates
  - 2 - the solver reports a summary of the solution status at each iteration
  - >2 - the solver produces several (somewhat unreadable) outputs at each iteration step

- `debugConvergence` [default `false`] taking values in [`true`,`false`]

  Includes additional output to help debug failed convergence.

- `debugConvergenceThreshold` [default 100000]

  Threshold above which solves warns about large values. Only used when `debugConvergence=true`

- `allowSave` [default `false`] taking values in [`true`,`false`]

  Generates code that permit saving the "hessian" matrix, for subsequent optimization of pivoting, row/-column permutations, and scaling for the hessian's LU factorization. The hessian in saved in two files named

  `{classname}_WW.subscripts and {classname}_WW.values`

  that store the sparsity structure and the actual values, respectively, at some desired iteration (see output parameter `solver`).

- `profiling` [default `false`] taking values in [`true`,`false`]

  When nonzero, adds profiling to the C code.

Outputs:

- `classname`

  Name of the class created.

4 Model Predictive Control

`TensCalc` provides a class to facilitate the simulation of MPC state-feedback controllers, MHE state estimators, and MPC-MHE output-feedback controllers.

4.1 MPC control

The following code shows an example of how to use the `Tmpc` class to generate a solver for a state-feedback MPC controller and to simulate its operation in feedback.

```
% create symbolic optimization
Tvariable Ts [];
Tvariable x [nx,T];  % [x(t+Ts),...,x(t+T*Ts)]
Tvariable u [nu,T];  % [u(t),...,u(t+(T-1)*Ts)]
Tvariable A [nx,nx];
Tvariable B [nx,nu];

dxFun=@(x,u,A,B,C,D)A*x+B*u;

J=norm2(x)+norm2(u);

% create mpc object
mpc=Tmpc('sampleTime',Ts,...
         'inputVariable,u,...
␣␣␣␣␣␣␣␣␣␣␣'stateVariable,x,...
         'stateDerivative',dx,...
         'objective',J,...
         'constraints',{u<=1, u>=-1},...
         'outputExpressions',{J,x,u},
         'parameters',{A,B},...
         'classname','tmp1');

% set parameter values
setParameter(mpc,'A',A);
setParameter(mpc,'B',B);

% set process initial condition
setInitialState(mpc,t0,x0);

u0=zeros(nu,T);               % cold start
for i=1:100
    % move warm-start away from constraints
    u0=min(u0,.95);
    u0=max(u0,-.95);
    setSolverWarmStart(mpc,u0);

    [solution,J,x,u]=solve(mpc,mu0,maxIter,saveIter);

    % apply 3 controls and get time and warm start for next iteration
    ufinal=0;
    [t,u0]=applyControls(mpc,solution,3,ufinal);
end

history=getHistory(mpc);
plot(history.t,history.x,'.-',history.t,history.u,'.-');grid on;
```

4.2 MPC-MHE control

The following code shows an example of how to use the `Tmpc` class to generate a solver for an output-feedback MPC-MHE controller and to simulate its operation in feedback.

```
% create symbolic optimization
Tvariable Ts [];
Tvariable x      [nx,L+T+1];    % [x(t-L*Ts),...,x(t),...,x(t+T*Ts)]
Tvariable y_past [ny,L+1];      % [y(t-L*Ts),...,y(t)]
Tvariable u_past [nu,L+1+delay] % [u(t-L*Ts),...,u(t+(delay-1)*Ts)]
Tvariable u      [nu,T-delay];  % [u(t+delay*Ts), ...,u(t+(T-1)*Ts)]
Tvariable d      [nd,L+T];      % [d(t-(L-1)*Ts,...,x(t),...,x(t+(T-1)*Ts)]
Tvariable A [nx,nx];
Tvariable B [nx,nu];
Tvariable C [ny,nx];
Tvaribale D [ny,nu];

dxFun=@(x,u,A,B,C,D)A*x+B*u;
yFun=@(x,u,A,B,C,D)C*x+D*u;

J=norm2(x(:,L+1:end))+norm2(u)-norm2(d)-norm2(y_past-y(x(:,1:L+1),u_past(:,1:L+1)));

% create mpc-mhe object
mpcmhe=Tmpcmhe('sampleTime',Ts,...
               'stateVariable',x,...
               'pastInputVariable',u_past,...
               'pastOutputVariable',y_past,...
               'futureControlVariable',u,...
               'disturbanceVariable',d,...
               'stateDerivativeFunction',dxFun,...
               'outputFunction',yFun,...
               'objective',J,...
               'inputConstraints',{u<=1, u>=-1},...
               'disturbanceConstraints',{d<=1, d>=-1},...
               'outputExpressions',{J,x,u,d},
               'parameters',{A,B,C,D},...
               'classname','tmp1');

% set parameter values
setParameter(mpcmhe,'A',A);
setParameter(mpcmhe,'B',B);
setParameter(mpcmhe,'C',C);
setParameter(mpcmhe,'D',D);

% set process initial condition, inputs, disturbances, and
% noise and get the corresponding measurements
[t,y_past,u_past]=setInitialState(mpcmhe,t0,x0,u0,d0,n0);

% cold start
x_warm=zeros(nx,1);
u_warm=zeros(nu,T-delay);
d_warm=zeros(nd,L+T);

for i=1:100
    % move warm-start away from constraints
    u_warm=min(u_warm,.95);
    u_warm=max(u_warm,-.95);
    d_warm=min(d_warm,.95);
    d_warm=max(d_warm,-.95);
    setSolverWarmStart(mpcmhe,x_warm,u_warm,d_warm);
    setSolverMeasurements(mpcmhe,y_past,u_past);
    [solution,J,x,u,d]=solve(mpcmhe,mu0,maxIter,saveIter);

    % apply 3 optimal controls/disturbances and get time,
    % (noiseless) measurements, and warm start for the next iteration
    ufinal=zeros(nu,3);
    dfinal=zeros(nd,3);
    [t,y_past,u_past,x_warm,u_warm,d_warm]=updateWarmStart(mpcmhe,solution,3,ufinal,dfinal);
end

history=getHistory(mpcmhe);
plot(history.t,history.x,'.-',history.t,history.u,'.-');grid on;
```

A Tensor calculus

In TensCalc, an *α-index tensor* is an array in $\mathbb{R}^{n_1\times n_2\times\cdots\times n_\alpha}$ where α is an integer in $\mathbb{Z}_{\geqslant 0}$. By convention, the case $\alpha = 0$ corresponds to a *scalar* in $\mathbb{R}$. We use the terminology *vector* and *matrix* for the cases $\alpha = 1$ and $\alpha = 2$, respectively. The integar α is called the *index* of the tensor and the tuple of integers $(n_1, n_2, \dots, n_\alpha)$ (possibly empty for $\alpha = 0$) is called the *dimension* of the tensor.

Given a tensor $A \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_\alpha}$, $n_A \coloneqq (n_1, n_2, \dots, n_A)$ denotes the dimension of the tensor; $\mathbb{R}^{(n_A)}$ the linear space $\mathbb{R}^{n_1\times n_2\times\cdots\times n_\alpha}$;

$$\mathcal{I}_A \coloneqq \Big\{(i_1, i_2, \dots, i_\alpha) : i_1 \in \{1, \dots, n_1\}, i_2 \in \{1, \dots, n_2\}, \dots, i_\alpha \in \{1, \dots, n_\alpha\}\Big\}$$

the set of all tuples of indices into the entries of A. The entry of A corresponding to the indices in the tuple $i \in \mathcal{I}_A$ is denoted by A_i and the whole tensor in $\mathbb{R}^{n_A}$ with entries given by A_i, $i \in \mathcal{I}_A$ is denoted by

$$\big[A_i\big]_{i\in\mathcal{A}}.$$

Given two tensors $A \in \mathbb{R}^{(n_A)}$, $B \in \mathbb{R}^{(n_B)}$ with dimensions

$$n_A \coloneqq (n_1, n_2, \dots, n_\alpha), \qquad\qquad n_B \coloneqq (m_1, m_2, \dots, m_\beta),$$

we define

$$n_A \times n_B \coloneqq (n_1, n_2, \dots, n_\alpha, m_1, m_2, \dots, m_\beta),$$

which allow us to write

$$\mathbb{R}^{(n_A)} \times \mathbb{R}^{(n_B)} = \mathbb{R}^{(n_A\times n_B)}.$$

Given indices $i \coloneqq (i_1, i_2, \dots, i_\alpha) \in \mathcal{I}_A$, $j = (j_1, j_2, \dots, j_\beta) \in \mathcal{I}_B$, we denote by

$$(i, j) \coloneqq (i_1, i_2, \dots, i_\alpha, j_1, j_2, \dots, j_\beta)$$

the concatenation of all the indices and by

$$\mathcal{I}_A \times \mathcal{I}_B \coloneqq \big\{(i, j) : i \in \mathcal{I}_A, j \in \mathcal{I}_B\big\}$$

the set of all corresponding indices.

A.1 Tensor Addition

Tensors with the same dimension can be added/subtracted entry by entry. Specifically, given two tensors A, B with the same dimension $n_A = n_B$, we define the tensor $A \pm B$ with the same dimension $n_{A+B} = n_A = n_B$ with entries defined by

$$A \pm B = \big[A_i \pm B_i\big]_{i\in\mathcal{I}_A=\mathcal{I}_B=\mathcal{I}_{A+B}}.$$

Technical note 1. Tensor addition has all the properties of an Abelian group (commutative, associative, zero element, inverse element). TensCalc is aware of these properties and applies then extensively to carry out symbolic simplifications, but this is transparent to the user. □

A.2 Tensor Multiplication

Given an α-index tensor $A \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_\alpha}$, a β-index tensor $B \in \mathbb{R}^{m_1 \times m_2 \times \cdots \times m_\beta}$, and a δ-index tensor $C \in \mathbb{R}^{o_1 \times o_2 \times \cdots \times o_\delta}$, we say that the integer-valued vectors $p \in \mathbb{Z}^\alpha$, $q \in \mathbb{Z}^\beta$, $r \in \mathbb{Z}^\delta$ are *product compatible for* $\mathbb{R}^{n_1 \times n_2 \times \cdots \times n_\alpha}$*,* $\mathbb{R}^{m_1 \times m_2 \times \cdots \times m_\beta}$*, and* $\mathbb{R}^{o_1 \times o_2 \times \cdots \times o_\delta}$ if the following conditions hold;

1. p, q, and r dot not have repeated entries;
2. the union of the positive entries of p, q, and r form an empty set or a set of the form $\{1, 2, \dots, \gamma\}$;
3. the set of negative entries of p, q, and r form an empty set or a set of the form $\{-1, -2, \dots, -\sigma\}$;
4. if $p_i = q_j = r_l$, then $n_i = m_j = o_l$.

For a product compatible pair, we define the (p, q, r)*-product of A, B, and C* to be a γ-index tensor defined by

$$(A^{(p)} * B^{(q)} * C^{(r)})_{k_1,\dots,k_\gamma} = \sum_{\ell_1,\dots,\ell_\sigma} A_{i_1,\cdots,i_\alpha} B_{j_1,\dots,j_\beta} C_{l_1,\dots,l_\delta} \tag{3}$$

where

1. the summation indices $\ell_1, \dots \ell_\sigma$ are matched to the A-indices i_x, the B-indices j_y, and the C-indices l_z based on the negative entries of p, q, and r as follows
$$p_x = -w \;\Rightarrow\; \ell_w = i_x, \quad q_y = -w \;\Rightarrow\; \ell_w = j_y, \quad r_z = -w \;\Rightarrow\; \ell_w = l_z, \quad \forall w \in \{1, \dots, \sigma\}.$$
2. the product indices $k_1, \dots, k_\gamma$ are matched to the A-indices i_x, the B-indices j_y , and the C-indices l_z based on the positive entries of p, q, and r as follows
$$p_x = w \;\Rightarrow\; k_w = i_x, \quad q_y = w \;\Rightarrow\; k_w = j_y, \quad r_z = w \;\Rightarrow\; k_w = l_z, \quad \forall w \in \{1, \dots, \gamma\}.$$

These rules also apply for 0-index tensors (i.e., scalars), in which case the scalar always appear in the product inside the summation (without an index). *The rules above generalize trivially to any number of factors, including a single factor*, which would be of the form

$$(A^{(p)} *)_{k_1,\dots,k_\gamma} = \sum_{\ell_1,\dots,\ell_\sigma} A_{i_1,\cdots,i_\alpha}$$

for indices selected using the rules above.

For several linear algebra operations, it will be useful to consider the special case of tensor multiplication that extends the usual matrix product when the first tensor is a square matrix. Specifically given a square matrix $A \in \mathbb{R}^{n_1 \times n_1}$ and another tensor $B \in \mathbb{R}^{n_1 \times n_2 \times \cdot \times n_\sigma}$ whose first dimension matches that of A, we extend the usual matrix multiplication as follows:

$$AB = A^{(1,-1)} * B^{(-1,2,\cdots,\sigma)} \in \mathbb{R}^{n_1 \times n_2 \times \cdot \times n_\sigma}, \tag{4}$$

and will continue to simply use AB to denote this product. This extended multiplication continues to enjoy the associative rule in that given and additional square matrix $C \in \mathbb{R}^{n_1 \times n_1}$, we have that

$$(C * A) * B = C * (A * B).$$

In addition, when the matrix A in nonsingular, we further introduce the notation

$$A \backslash B := A^{-1} B$$

where we can also see $A \backslash B$ as the solution $Y \in \mathbb{R}^{n_1 \times n_2 \times \cdot \times n_\sigma}$ to the system of equations

$$AY = B.$$

Examples Suppose that $a \in \mathbb{R}$ (scalar), $x \in \mathbb{R}^2$, $y \in \mathbb{R}^2$, $A \in \mathbb{R}^{3\times 2}$, $B \in \mathbb{R}^{2\times 2}$

$$\begin{aligned}
x^{(-1)} &= x_1 + x_2 \in \mathbb{R} && \text{(scalar)}\\
a^{()} * x^{(1)} &= x^{(1)} * a^{()} = (ax_1, ax_2) \in \mathbb{R}^2 && \text{(vector)}\\
a^{()} * x^{(-1)} &= x^{(-1)} * a^{()} = a(x_1 + x_2) \in \mathbb{R} && \text{(scalar)}\\
x^{(1)} * y^{(-1)} &= y^{(-1)} * x^{(1)} = \big(x_1(y_1 + y_2), x_2(y_1 + y_2)\big) \in \mathbb{R}^2 && \text{(vector)}\\
x^{(1)} * y^{(1)} &= y^{(1)} * x^{(1)} = (x_1 y_1, x_2 y_2) \in \mathbb{R}^2 && \text{(vector)}\\
x^{(1)} * y^{(2)} &= y^{(2)} * x^{(1)} = \begin{bmatrix} x_1 y_1 & x_1 y_2 \\ x_2 y_1 & x_2 y_2 \end{bmatrix} \in \mathbb{R}^2 && \text{(matrix)}\\
x^{(2)} * y^{(1)} &= y^{(1)} * x^{(2)} = \begin{bmatrix} x_1 y_1 & x_2 y_1 \\ x_1 y_2 & x_2 y_2 \end{bmatrix} \in \mathbb{R}^2 && \text{(matrix)}\\
A^{(1,-1)} * x^{(-1)} &= \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \\ a_{31}x_1 + a_{32}x_2 \end{bmatrix} \in \mathbb{R}^2 && \text{(vector)}\\
A^{(1,2)} * x^{(-1)} &= \begin{bmatrix} a_{11}(x_1 + x_2) & a_{12}(x_1 + x_2) \\ a_{21}(x_1 + x_2) & a_{22}(x_1 + x_2) \\ a_{31}(x_1 + x_2) & a_{32}(x_1 + x_2) \end{bmatrix} \in \mathbb{R}^2 && \text{(matrix)}\\
A^{(2,1)} * &= A' = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{23} \end{bmatrix} && \text{(matrix)}\\
B^{(1,2)} * I_{2\times 2}^{(1,2)} &= \begin{bmatrix} b_{11} & 0 \\ 0 & b_{22} \end{bmatrix} && \text{(matrix)}\\
B^{(-1,-2)} * I_{2\times 2}^{(-1,-2)} &= b_{11} + b_{22} && \text{(scalar)}\\
B^{(1,1)} * &= (b_{11}, b_{22}) && \text{(vector)}\\
B^{(1,-1)} * I_{2\times 2}^{(1,-1)} &= (b_{11}, b_{22}) && \text{(vector)}
\end{aligned}$$

Technical note 2. The tensor product enjoys a special form of *commutative property*, in the sense that

$$\begin{aligned}
A^{(p)} * B^{(q)} * C^{(r)} = A^{(p)} * C^{(r)} * B^{(q)} &= B^{(q)} * A^{(p)} * C^{(r)}\\
&= B^{(q)} * C^{(r)} * A^{(p)} = C^{(r)} * A^{(p)} * B^{(q)} = C^{(r)} * B^{(q)} * A^{(p)}.
\end{aligned}$$

An *associative-like property* also holds but it is more complicated, for example

$$(A^{(p)} *)^{(q)} * C^{(r)} = A^{(\bar{p})} * C^{(r)}$$

with

$$p_x = w > 0 \quad \Rightarrow \quad \bar{p}_x = q_w, \qquad\qquad p_x = -w < 0 \quad \Rightarrow \quad \bar{p}_x = -w - \sigma_C,$$

where σ_C denotes the number of summations needed for the product defined by q and r.

The 2α-index *identity matrix* I is defined by

$$I_{i_1,\dots,i_\alpha,j_1,\dots,j_\alpha} = \begin{cases} 1 & i_1 = j_1, \dots, i_\alpha = j_\alpha \\ 0 & \text{otherwise.} \end{cases}$$

For such matrices, we have that

$$A^{(p)} * I^{(q)} = I^{(q)} * A^{(p)} = A$$

when q has as many positive as negative entries and if we obtain the vector $(1, 2, \dots, \alpha)$ when we replace each negative entry $p_x < 0$ of p by the entry of $q_y > 0$ of q such that $q_z = p_x$ with the indices z and y such that $|y - z| = \alpha$.

`TensCalc` is aware of these rules and applies the extensively to carry out symbolic simplifications, but this is transparent to the user. $\square$

A.3 Composition

Given two functions $F : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_B)}$ and $G : \mathbb{R}^{(n_B)} \to \mathbb{R}^{(n_C)}$, the *(usual) composition of F with G* is defined by

$$\begin{aligned} G \circ F : \mathbb{R}^{(n_A)} &\to \mathbb{R}^{(n_C)} \\ x &\mapsto G\big(F(x)\big) \end{aligned}$$

and given a third function $g : \mathbb{R} \to \mathbb{R}$, the *component-wise composition of F with g* is defined by

$$\begin{aligned} g \circ F : \mathbb{R}^{(n_A)} &\to \mathbb{R}^{(n_B)} \\ x &\mapsto \big[g\big(F_i(x)\big)\big]_{i \in \mathcal{I}_B}. \end{aligned}$$

More generally, given a function $g : \mathbb{R} \to \mathbb{R}^{(n_C)}$, the *component-wise composition of F with g* is defined by

$$\begin{aligned} g \circ F : \mathbb{R}^{(n_A)} &\to \mathbb{R}^{(n_B \times n_C)} \\ x &\mapsto \big[g_j\big(F_i(x)\big)\big]_{(i,j) \in \mathcal{I}_B \times \mathcal{I}_C}. \end{aligned}$$

A.4 Tensor Gradient

Given a function $F : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_B)}$, the *gradient of $x \mapsto F(x)$* is the function defined by

$$\begin{aligned} \nabla_x F : \mathbb{R}^{(n_A)} &\to \mathbb{R}^{(n_B \times n_A)} \\ x &\mapsto \big[\nabla_x F\big]_{(i,j) \in \mathcal{I}_B \times \mathcal{I}_A} := \left[\frac{\partial F_i(x)}{\partial x_j}\right]_{(i,j) \in \mathcal{I}_B \times \mathcal{I}_A} \end{aligned}$$

The hessian matrix is then defined by

$$H_{xx} F = \nabla_x (\nabla_x F).$$

Sum rule: Given two functions $F, G : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_B)}$, the gradient of their sum is given by the sum of the gradients:

$$\nabla_x\big(F(x) + G(x)\big) = \nabla_x F(x) + \nabla_x G(x).$$

Product rule: Given two function $F : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_B)}$, $G : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_C)}$, $H : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_D)}$ and a triple (p, q, r) that is product compatible for their co-domains, the gradient of their (p, q, r)-product is given by

$$\begin{aligned}\nabla_x\big(F(x)^{(p)} * G(x)^{(q)} * H(x)^{(r)}\big) &= (\nabla_x F(x))^{(\bar{p})} * G(x)^{(q)} * H(x)^{(r)} \\ &+ F(x)^{(p)} * (\nabla_x G(x))^{(\bar{q})} * H(x)^{(r)} + F(x)^{(p)} * G(x)^{(q)} * (\nabla_x H(x))^{(\bar{r})}, \quad (5)\end{aligned}$$

where $n_A \coloneqq (n_1, n_2, \dots, n_\alpha)$, $\eta \coloneqq \max\{0, p, q, r\}$, and

$$\bar{p} \coloneqq (p, \eta+1, \dots, \eta+\alpha), \qquad \bar{q} \coloneqq (q, \eta+1, \dots, \eta+\alpha), \qquad \bar{r} \coloneqq (r, \eta+1, \dots, \eta+\alpha).$$

For the generalized product (4) of a square matrix-valued function $F : \mathbb{R}^{(m_A)} \to \mathbb{R}^{n_1 \times n_1}$ by a tensor-valued function $G : \mathbb{R}^{(m_A)} \to \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_\sigma}$, with $m_A \coloneqq (m_1, m_2, \dots, m_\alpha)$, the product rule in (5) becomes

$$\begin{aligned}\nabla_x\big(F(x)G(x)\big) &= \nabla_x\big(F^{(1,-1)} * G^{(-1,2,\dots,\sigma)}\big) \\ &= (\nabla_x F)^{(1,-1,\eta+1,\dots,\eta+\alpha)} * G^{(-1,2,\dots,\sigma)} + F^{(1,-1)} * (\nabla_x G)^{(-1,2,\dots,\sigma,\eta+1,\dots,\eta+\alpha)} \\ &= (\nabla_x F)^{(1,-1,\eta+1,\dots,\eta+\alpha)} * G^{(-1,2,\dots,\sigma)} + F \nabla_x G\end{aligned}$$

where $\eta \coloneqq \max\{2, \dots, \sigma\}$.

For the particular case of the Frobenius-norm squared of $F(x)$:

$$\|F(x)\|_F^2 \coloneqq F(x)^{(-1,\dots,-\alpha)} * F(x)^{(-1,\dots,-\alpha)},$$

we get

$$\nabla_x \|F(x)\|_F^2 = 2(\nabla_x F(x))^{(-1,\dots,-\alpha,1,\dots,\alpha)} * F(x)^{(-1,\dots,-\alpha)}.$$

Composition rule: Given two functions $F : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_B)}$ and $G : \mathbb{R}^{(n_B)} \to \mathbb{R}^{(n_C)}$, the gradient of the composition is given by

$$\begin{aligned}\nabla_x(G \circ F) : \mathbb{R}^{(n_A)} &\to \mathbb{R}^{(n_C \times n_A)} \\ x &\mapsto \left[\frac{\partial G_i\big(F(x)\big)}{\partial x_j}\right]_{(i,j) \in \mathcal{I}_C \times \mathcal{I}_A}\end{aligned}$$

where

$$\begin{aligned}\left[\frac{\partial G_i\big(F(x)\big)}{\partial x_j}\right]_{(i,j) \in \mathcal{I}_C \times \mathcal{I}_A} &= \left[\sum_{k \in \mathcal{I}_B} \frac{\partial G_i\big(F(x)\big)}{\partial F_k} \frac{\partial F_k(x)}{\partial x_j}\right]_{(i,j) \in \mathcal{I}_C \times \mathcal{I}_A} \\ &= \nabla_z G(F(x))^{(p)} * \nabla_x F(x)^{(q)}\end{aligned}$$

where $n_A \coloneqq (n_1, n_2, \dots, n_\alpha)$, $n_B \coloneqq (m_1, m_2, \dots, m_\beta)$, $n_C \coloneqq (s_1, s_2, \dots, s_\gamma)$ and

$$p = (1, 2, \dots, \gamma, -1, -2, \dots, -\beta), \qquad q = (-1, -2, \dots, -\beta, 1, 2, \dots, \alpha).$$

Given a third function $g : \mathbb{R} \to \mathbb{R}$, the gradient of the component-wise composition of F with g is given by

$$\begin{aligned}\nabla_x(g \circ F) : \mathbb{R}^{(n_A)} &\to \mathbb{R}^{(n_B \times n_A)}\\ x &\mapsto \Big[\frac{\partial g(F_i(x))}{\partial x_j}\Big]_{(i,j)\in\mathcal{I}_B\times\mathcal{I}_A}\end{aligned}$$

where

$$\begin{aligned}\Big[\frac{\partial g(F_i(x))}{\partial x_j}\Big]_{(i,j)\in\mathcal{I}_B\times\mathcal{I}_A} &= \Big[\frac{\partial g(F_i(x))}{\partial z}\frac{\partial F_i(x)}{\partial x_j}\Big]_{(i,j)\in\mathcal{I}_B\times\mathcal{I}_A}\\ &= \Big(\Big(\frac{\partial g}{\partial z} \circ F\Big)(x)\Big)^{(p)} * \nabla_x F(x)^{(q)}\end{aligned}$$

where

$$p = (1, 2, \dots, \beta), \qquad q = (1, 2, \dots, \beta, \beta + 1, \beta + 2, \dots, \beta + \alpha).$$

More generally, given a third function $g : \mathbb{R} \to \mathbb{R}^{(n_C)}$, the gradient of the component-wise composition of F with g is given by

$$\begin{aligned}\nabla_x(g \circ F) : \mathbb{R}^{(n_A)} &\to \mathbb{R}^{(n_B \times n_C \times n_A)}\\ x &\mapsto \Big[\frac{\partial g_k(F_i(x))}{\partial x_j}\Big]_{(i,k,j)\in\mathcal{I}_B\times\mathcal{I}_C\times\mathcal{I}_A}\end{aligned}$$

where

$$\begin{aligned}\Big[\frac{\partial g_k(F_i(x))}{\partial x_j}\Big]_{(i,k,j)\in\mathcal{I}_B\times\mathcal{I}_C\times\mathcal{I}_A} &= \Big[\frac{\partial g_k(F_i(x))}{\partial z}\frac{\partial F_i(x)}{\partial x_j}\Big]_{(i,k,j)\in\mathcal{I}_B\times\mathcal{I}_C\times\mathcal{I}_A}\\ &= \Big(\Big(\frac{\partial g_k}{\partial z} \circ F\Big)(x)\Big)^{(p)} * \nabla_x F(x)^{(q)}\end{aligned}$$

where

$$p = (1, 2, \dots, \beta + \gamma), \qquad q = (1, 2, \dots, \beta, \beta + \gamma + 1, \beta + \gamma + 2, \dots, \beta + \gamma + \alpha).$$

Solutions to linear equations: Given a square matrix-valued function $F : \mathbb{R}^{(m_A)} \to \mathbb{R}^{n_1\times n_1}$, a tensor-valued function $G : \mathbb{R}^{(m_A)} \to \mathbb{R}^{n_1\times n_2\times\cdots\times n_\sigma}$, with $m_A \coloneqq (m_1, m_2, \dots, m_\alpha)$, and their generalized product $H : \mathbb{R}^{(m_A)} \to \mathbb{R}^{n_1\times n_2\times\cdots\times n_\alpha}$, defined by $H(x) \coloneqq F(x)G(x)$, $\forall x \in \mathbb{R}^{m_1\times m_2\times\cdots\times m_\alpha}$, we conclude from the implicit function theorem and the product rule that

$$\begin{aligned}\nabla_x\big(F(x)G(x)\big) = \nabla_x H(x) \quad &\Leftrightarrow \quad (\nabla_x F)^{(1,-1,\eta+1,\dots,\eta+\alpha)} * G^{(-1,2,\dots,\sigma)} + F\nabla_x G = \nabla_x H\\ &\Leftrightarrow \quad \nabla_x G = F^{-1}\Big(\nabla_x H - (\nabla_x F)^{(1,-1,\eta+1,\dots,\eta+\alpha)} * G^{(-1,2,\dots,\sigma)}\Big)\\ &\Leftrightarrow \quad \nabla_x G = F\backslash\Big(\nabla_x H - (\nabla_x F)^{(1,-1,\eta+1,\dots,\eta+\alpha)} * G^{(-1,2,\dots,\sigma)}\Big),\end{aligned}$$

where $\eta \coloneqq \max\{2, \dots, \sigma\}$.

Matrix rules: Given a function $F : \mathbb{R}^{(n_A)} \to \mathbb{R}^{n_C}$, $n_A := (n_1, n_2, \dots, n_\alpha)$, $n_C := (n, n)$ the gradients

$$\begin{aligned}
&\nabla_x \log\det F : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_A)},\\
&\nabla_x \operatorname{trace} F^{-1} : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_A)},\\
&\nabla_x F^{-1} : \mathbb{R}^{(n_A)} \to \mathbb{R}^{(n_C \times n_A)},
\end{aligned}$$

are given by

$$\begin{aligned}
\nabla_x\big(\log\det F(x)\big) &= \Big[\sum_{ij\in\mathcal{I}_B} \frac{\partial \log\det F}{\partial F_{ij}} \frac{\partial F_{ij}}{\partial x_r}\Big]_{r\in\mathcal{I}_A}\\
&= \Big[\sum_{(i,j)\in\mathcal{I}_C} [F^{-1}]_{ji} \frac{\partial F_{ij}}{\partial x_r}\Big]_{r\in\mathcal{I}_A}\\
&= (F^{-1})^{(p)} * \nabla_x F^{(q)}, \quad p := (-2,-1),\ q := (-1,-2,1,2,\dots,\alpha),\\
\nabla_x\big(\operatorname{trace} F(x)^{-1}\big) &= \Big[\sum_{(i,j)\in\mathcal{I}_C} \frac{\partial \operatorname{trace} F^{-1}}{\partial F_{ij}} \frac{\partial F_{ij}}{\partial x_r}\Big]_{r\in\mathcal{I}_A}\\
&= -\Big[\sum_{(i,j)\in\mathcal{I}_C} [F^{-2}]_{ji} \frac{\partial F_{ij}}{\partial x_r}\Big]_{r\in\mathcal{I}_A}\\
&= -(F^{-2})^{(p)} * \nabla_x F^{(q)}, \quad p := (-2,-1),\ q := (-1,-2,1,2,\dots,\alpha),\\
\nabla_x(F^{-1}(x)) &= \Big[\sum_{(i,j)\in\mathcal{I}_C} \frac{\partial [F^{-1}]_{k\ell}}{\partial F_{ij}} \frac{\partial F_{ij}}{\partial x_r}\Big]_{(k\ell,r)\in\mathcal{I}_C\times\mathcal{I}_A}\\
&= -\Big[\sum_{(i,j)\in\mathcal{I}_C} [A^{-1}]_{ki} \frac{\partial F_{ij}}{\partial x_r} [A^{-1}]_{j\ell}\Big]_{(k\ell,r)\in\mathcal{I}_C\times\mathcal{I}_A}\\
&= -\big(F^{-1}\big)^{(p)} * \nabla_x F^{(q)} * \big(F^{-1}\big)^{(r)},\\
&\qquad\quad p = (1,-1),\ q = (-1,-2,3,4,\dots,2+\alpha),\ r = (-2,2).
\end{aligned}$$

where, for simplicity we omitted the dependence on x from $F(x)$, and used the facts that

$$\frac{\partial \log\det(A)}{\partial A_{ij}} = [A^{-1}]_{ji}, \qquad \frac{\partial \operatorname{trace} A^{-1}}{\partial A_{ij}} = -[A^{-2}]_{ji}, \qquad \frac{\partial [A^{-1}]_{k\ell}}{\partial A_{ij}} = -[A^{-1}]_{ki}[A^{-1}]_{j\ell}.$$

For sparse matrix, it important to implement these rules without ever constructing the inverse matrix. To accomplish this, we use the notation

$$\operatorname{trace} Y = \sum_{i_1} Y_{i_1,i_1,i_3,\dots,i_\alpha} = Y^{(-1,-1,1,2,\dots,n_\sigma-2)}* \in \mathbb{R}^{n_3\times n_4\times\cdots\times n_\sigma}, \quad Y \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_\sigma}$$

which allow us to re-write the gradients above as follows:

$$\begin{aligned}
\nabla_x\big(\log\det F(x)\big) &= (F^{-1})^{(-2,-1)} * (\nabla_x F)^{(-1,-2,1,2,\dots,\alpha)}\\
&= \operatorname{trace}\Big((F^{-1})^{(1,-1)} * (\nabla_x F)^{(-1,2,3,4,\dots,\alpha+2)}\Big)\\
&= \operatorname{trace}(F^{-1}\nabla_x F)
\end{aligned}$$

$$
\begin{aligned}
&= \operatorname{trace}(F\backslash \nabla_x F)\\
\nabla_x\big(\operatorname{trace} F(x)^{-1}\big) &= -(F^{-2})^{(-2,-1)} * (\nabla_x F)^{(-1,-2,1,2,\dots,\alpha)}\\
&= -\operatorname{trace}\Big((F^{-2})^{(1,-1)} * (\nabla_x F)^{(-1,2,3,4,\dots,\alpha+2)}\Big)\\
&= -\operatorname{trace} F^{-2}\nabla_x F\\
&= -\operatorname{trace} F^{-1}F^{-1}\nabla_x F\\
&= -\operatorname{trace} F\backslash\big(F\backslash \nabla_x F\big),
\end{aligned}
$$

which involve solving systems of linear equations, rather than explicitly computing inverses.

A.5 Table interpolation

Given two sets of tensors $X_i \in \mathbb{R}^{n_1\times\cdots\times n_\alpha}$, $Y_i \in \mathbb{R}^{m_1\times\cdots\times m_\beta}$, $i \in \{1,2,\dots,K\}$ and a scalar $L \in \mathbb{R}$, consider the functions $F : \mathbb{R}^{n_1\times\cdots\times n_\alpha} \to \mathbb{R}^{m_1\times\cdots\times m_\beta}$ defined by

$$
F(x) \coloneqq \sum_{i=1}^{K} e^{-\frac{1}{2S^2}\|X_i-x\|_F^2} Y_i, \qquad \|X_i - X\|_F^2 \coloneqq (X_i - X)^{(-1,-2,\dots,-\alpha)} * (X_i - X)^{(-1,-2,\dots,-\alpha)}
$$

$$
G(x) \coloneqq \frac{F(x)}{f(x)}, \qquad f(x) \coloneqq \sum_{i=1}^{K} e^{-\frac{1}{2S^2}\|X_i-x\|^2},
$$

The gradients and hessian matrix of F and f can be computed using the previous formulas, leading to

$$
\begin{aligned}
\nabla_x F(x) &\coloneqq -\frac{1}{2S^2}\sum_{i=1}^{K} e^{-\frac{1}{2S^2}\|X_i-x\|_F^2} Y_i^{(1,\dots,\beta)} * \nabla_x\big(\|X_i - x\|_F^2\big)^{(\beta+1,\dots,\beta+\alpha)}\\
\nabla_x f(x) &\coloneqq -\frac{1}{2S^2}\sum_{i=1}^{K} e^{-\frac{1}{2S^2}\|X_i-x\|_F^2} \nabla_x\|X_i - x\|_F^2,
\end{aligned}
$$

the gradient of G is given by

$$
\nabla_x G(x) = \frac{\nabla_x F(x)}{f(x)} - \frac{F(x)^{(1,\dots,\beta)} * \nabla_x f(x)^{(\beta+1,\dots,\beta+\alpha)}}{f(x)^2}
$$

and its Hessian matrix is given by

$$
\begin{aligned}
H_{xx}G(x) &= \nabla_x\Big(\frac{\nabla_x F(x)}{f(x)} - \frac{F(x)^{(1,\dots,\beta)} * \nabla_x f(x)^{(\beta+1,\dots,\beta+\alpha)}}{f(x)^2}\Big)\\
&= \frac{H_{xx}F(x)}{f(x)} - \frac{\nabla_x F(x)^{(1,\dots,\beta+\alpha)} * \nabla_x f(x)^{(\beta+\alpha+1,\dots,\beta+2\alpha)}}{f(x)^2}\\
&\quad - \frac{\nabla_x F(x)^{(1,\dots,\beta,\beta+\alpha+1,\dots,\beta+2\alpha)} * \nabla_x f(x)^{(\beta+1,\dots,\beta+\alpha)}}{f(x)^2}\\
&\quad - \frac{F(x)^{(1,\dots,\beta)} * H_{xx} f(x)^{(\beta+1,\dots,\beta+\alpha,\beta+\alpha+1,\dots,\beta+2\alpha)}}{f(x)^2}
\end{aligned}
$$

$$
\begin{aligned}
&+ \frac{\left(F(x)^{(1,\dots,\beta)} * \nabla_x f(x)^{(\beta+1,\dots,\beta+\alpha)}\right)^{(1,\dots,\beta+\alpha)} * \nabla_x (f(x)^2)^{(\beta+\alpha+1,\dots,\beta+2\alpha)}}{f(x)^4} \\
= &\frac{H_{xx}F(x)}{f(x)} - \frac{1}{f(x)^2}\Big(\nabla_x F(x)^{(1,\dots,\beta+\alpha)} * \nabla_x f(x)^{(\beta+\alpha+1,\dots,\beta+2\alpha)} \\
&+ \nabla_x F(x)^{(1,\dots,\beta,\beta+\alpha+1,\dots,\beta+2\alpha)} * \nabla_x f(x)^{(\beta+1,\dots,\beta+\alpha)} + F(x)^{(1,\dots,\beta)} * H_{xx} f(x)^{(\beta+1,\dots,\beta+\alpha,\beta+\alpha+1,\dots,\beta+2\alpha)}\Big) \\
&+ \frac{2}{f(x)^3} F(x)^{(1,\dots,\beta)} * \nabla_x f(x)^{(\beta+1,\dots,\beta+\alpha)} * \nabla_x f(x)^{(\beta+\alpha+1,\dots,\beta+2\alpha)}.
\end{aligned}
$$

These formulas could be obtained from the previous rules together with the division-by-scalar rule

$$
\nabla_x \Big(\frac{F(x)}{f(x)}\Big) = \frac{\nabla_x F(x)}{f(x)} - \frac{F(x)^{(1,\dots,\beta)} * \nabla_x f(x)^{(\beta+1,\dots,\beta+\alpha)}}{f(x)^2}.
$$

However, the above rules provide a small simplification in canceling a few $f(x)$ terms in numerator and denominator.

B Examples to discuss...

B.1 RSS

Suppose that we want to minimize the criteria

$$J(\alpha, \Theta, R) \coloneqq \frac{1}{2}\|B(\Theta, \psi)\alpha + R - Y\|^2,$$

with respect to the antena parameters α and the source location parameters Θ, R, where

$$Y_{T\times 1} = \begin{bmatrix} \log y(1) \\ \log y(2) \\ \vdots \\ \log y(T) \end{bmatrix}, \quad R_{T\times 1} = \begin{bmatrix} -d\log r(1) \\ -d\log r(2) \\ \vdots \\ -d\log r(T) \end{bmatrix}, \quad \theta_{T\times 1} = \begin{bmatrix} \theta(1) \\ \theta(2) \\ \vdots \\ \theta(T) \end{bmatrix}, \quad \psi_{T\times 1} = \begin{bmatrix} \psi(1) \\ \psi(2) \\ \vdots \\ \psi(T) \end{bmatrix},$$

$$B(\Theta, \psi)_{T\times N} = \begin{bmatrix} b(\theta(1)-\psi(1)-\beta_1) & b(\theta(1)-\psi(1)-\beta_2) & \cdots & b(\theta(1)-\psi(1)-\beta_N) \\ b(\theta(2)-\psi(2)-\beta_1) & b(\theta(2)-\psi(2)-\beta_2) & \cdots & b(\theta(2)-\psi(2)-\beta_N) \\ \vdots & \vdots & \ddots & \vdots \\ b(\theta(T)-\psi(T)-\beta_1) & b(\theta(T)-\psi(T)-\beta_2) & \cdots & b(\theta(T)-\psi(T)-\beta_N) \end{bmatrix},$$

where $b : \mathbb{R} \to \mathbb{R}$ corresponds to some given basis function.

B.2 SOM

Given integers $D \ll K, M \ll N$ and vectors $x_1, \dots, x_N \in \mathbb{R}^K$, suppose we want to minimize the criteria

$$J(v_1, \dots, v_M \in \mathbb{R}^K; y_1, \dots, y_N \in \mathbb{R}^D) \coloneqq \sum_{n=1}^{N} \left\| \hat{x}_n - x_n \right\|^2 \tag{6}$$

where

$$\hat{x}_n \coloneqq \sum_{m=1}^{M} v_m b_m(y_n) \in \mathbb{R}^M, \qquad b_m(y) \coloneqq b\big(\|y - c_m\|^2\big) \in \mathbb{R}, \qquad m \in \{1, 2, \dots, M\},$$

for a given "shape" function $b : [0, \infty) \to [0, 1]$ (e.g., $b(s) = e^{-\lambda s}$) and fixed vectors $c_1, \dots, c_M \in \mathbb{R}^D$. We can re-write (7) as

$$\begin{aligned} J(v_1, \dots, v_M \in \mathbb{R}^K; y_1, \dots, y_N \in \mathbb{R}^D) &= \left\| \begin{bmatrix} \hat{x}_1 - x_1 & \cdots & \hat{x}_N - x_N \end{bmatrix} \right\|_F^2 \\ &= \left\| \begin{bmatrix} \sum_{m=1}^{M} v_m b(\|y_1 - c_m\|^2) & \cdots & \sum_{m=1}^{M} v_m b(\|y_n - c_m\|^2) \end{bmatrix} - X \right\|_F^2 = \|VB(Y, C) - X\|_F^2 \end{aligned} \tag{7}$$

where

$$V = \begin{bmatrix} v_1 & \cdots & v_M \end{bmatrix}_{K\times M}, \quad C = \begin{bmatrix} c_1 & \cdots & c_M \end{bmatrix}_{D\times M},$$
$$X \coloneqq \begin{bmatrix} x_1 & \cdots & x_N \end{bmatrix}_{K\times N}, \quad Y = \begin{bmatrix} y_1 & \cdots & y_N \end{bmatrix}_{D\times N},$$

$$B(Y,C) \coloneqq \begin{bmatrix} b(\|y_1 - c_1\|^2) & \cdots & b(\|y_N - c_1\|^2) \\ \vdots & \ddots & \vdots \\ b(\|y_1 - c_M\|^2) & \cdots & b(\|y_N - c_M\|^2) \end{bmatrix}_{M\times N}.$$

Moreover, $B(Y,C)$ is obtained by applying the scalar function b to every entry of the matrix

$$\begin{aligned} N_{M\times N} \coloneqq & \begin{bmatrix} \|y_1 - c_1\|^2 & \cdots & \|y_N - c_1\|^2 \\ \vdots & \ddots & \vdots \\ \|y_1 - c_M\|^2 & \cdots & \|y_N - c_M\|^2 \end{bmatrix} \\ = & \begin{bmatrix} (y_1 - c_1)'(y_1 - c_1) & \cdots & (y_N - c_1)'(y_N - c_1) \\ \vdots & \ddots & \vdots \\ (y_1 - c_M)'(y_1 - c_M) & \cdots & (y_N - c_M)'(y_N - c_M) \end{bmatrix} \\ = & \begin{bmatrix} y_1'y_1 & \cdots & y_N'y_N \\ \vdots & \ddots & \vdots \\ y_1'y_1 & \cdots & y_N'y_N \end{bmatrix} + \begin{bmatrix} c_1'c_1 & \cdots & c_1'c_1 \\ \vdots & \ddots & \vdots \\ c_M'c_M & \cdots & c_M'c_M \end{bmatrix} - \begin{bmatrix} y_1'c_1 & \cdots & y_N'c_1 \\ \vdots & \ddots & \vdots \\ y_1'c_M & \cdots & y_N'c_M \end{bmatrix} - \begin{bmatrix} c_1'y_1 & \cdots & c_1'y_N \\ \vdots & \ddots & \vdots \\ c_M'y_1 & \cdots & c_M'y_N \end{bmatrix} \\ = & Y^{(-1,2)} * Y^{(-1,2)} * 1_M^{(1)} + C^{(-1,1)} * C^{(-1,1)} * 1_N^{(2)} \\ & - Y^{(-1,2)} * C^{(-1,1)} - C^{(-1,1)} * Y^{(-1,2)}. \end{aligned}$$