Guide to the MLR Code

M Toussaint

September 2, 2017

Abstract

Code needs mathematical grounding.

Contents

1 Linear Algebra

I spare the description of the `mlr::Array` class – it is a standard tensor storage. It's implementation and interface is very similar to Octave's Array class.

Instead I just provide my recommentation for C++ operator overloading for easy linear algebra syntax.

	Tensor/Matlab notation	C++
"inner" product[1]	$C_{ijl} = \sum_k A_{ijk}B_{kl}$	`A * B`
index-wise product[2]	$c_i = a_i b_i$ or $c = a \circ b$	`a % b`
	$C_{ijkl} = A_{ijk}B_{kl}$	`A % B`
diag	$\mathrm{diag}(a)$	`diag(a)`
	$\mathrm{diag}(a)B$ or $c_i = a_i B_{ij}$	`a % B` or `diag(a) * B`
element-wise product	$c_i = a_i b_i$ or $c = a \circ b$	`a % b`
	$C_{ij} = A_{ij}B_{ij}$ or $C = A \circ B$	no operator-overload![3]
		`elemWiseProduct(A,B)`
outer product	$C_{ijklm} = A_{ijk}B_{lm}$	`A ^ B`
	$ab^\top$ (vectors)	`a ^ b`
transpose	$A_{ij} = B_{ji}$	`A = ~B`
inverse	$AB^{-1}C$	`A*(1/B)*C`
	$A^{-1}b$ (or `A\b` in Matlab)	`A|b`
element *reference*[5]	A_{103}	`A(1,0,3)`
	$A_{(n-2)03}$	`A(-2,0,3)`
sub-*references!*[6]	$x_i = A_{2i}$	`A(2,{})` or `A[2]`
	$C_i = A_{20i}$ or `C=A[2,0,:]`	`A(2,0,{})` ([7])
	$C_{ijk} = A_{20ijk}$	`A(2,0,{})` (trailing `{},{}..` are implicit)
sub-refercing ranges[8]	`C=A[2:4,:,:]`	`A({2,4})` (trailing `{},{}..` are implicit)
	`A[2,1:3,:]`	`A(2,{1,3})`
sub-*copies*[9]	`A(1:3, :, 5:)`	`A.sub(1,3, 0,-1, 5,-1)`
sub-selected-copies	`A[{1,3,4},:,{2,3},2:5]`	`A.sub({1,3,4}, 0,-1, {2,3}, 2,5)`
sub-assignment[10]	`A[4:6, 2:5] = B` ($B \in \mathbb{R}^{3\times4}$)	`A.setBlock(B, 4, 2)`
	`x[4:6] = b` ($b \in \mathbb{R}^3$)	`x.setBlock(b, 4)` or `x({4,6}) = b`
initialization	`A=[1 2 3]'`	`arr A={1.,2,3}`
		or `arr A(3, {1.,2,3})`
	`A=[1 2 3]`	`arr A=~arr({1.,2,3})`
		or `arr A(1, 3, {1.,2,3})`
	`A=[1 2; 3 4]`	`arr A(2,2, {1.,2,3,4})`
concatenation	$(x^\top, y^\top, z^\top)^\top$ (stacked vectors)	`(x,y,z)`
	`cat(1,A,B)` (stacked matrices)	`(A,B)` (memory serial)

[1] The word "inner" product should, strictly, be used to refer to a general 2-form $\langle \cdot, \cdot \rangle$ which, depending on coordinates, may have a non-Euclidean metric tensor. However, here we use it in the sense of "assuming Euclidean metric".

[2] For matrices or tensors, this is *not* the element-wise (Hadamard) product!

[3] The elem-wize product for matricies/tensors is much less used within equations than what I call 'index-wise product'.

[5] Negative indices are always interpreted as $n - index$. As we start indexing from 0, the index n is already out of range. $n - 1$ is the last entry. An index of -1 therefore means 'last'.

[6] In C++, assuming the last index to be memory aligned, sub-referencing is only efficient w.r.t. major indices: the reference then points to the same memory as the parent tensor.

[7] As a counter example, `A[:,0,2]` could be referenced in a memory aligned manner. It can only be copied with `A.sub(0,-1,0,0,2,2)`

[8] again, this can only be ranges w.r.t. the major index, to ensure memory alignment

[9] this is a $\mathbb{R}^{3\times\cdots}$ matrix: The '3' is *included* in the range.

[10] As the assignments are not memory-aligned, they can't be done with returned references.

2 Graph

Our graph syntax is a bit different to standard conventions. Actually, our graph could be called a *key-value hierarchical hyper graph*: nodes can play the role of normal nodes, or hypernodes (=edges or factors/cliques) that connect other nodes. Every node also has a set of keys (or tags, to retrieve the node by name) and a typed value (every node can be of a different type). This value can also be a graph, allowing to represent hierarchies of graphs and subgraphs.

- A graph is a set of nodes
- Every node has three properties:
  - A tuple of **keys** (=strings)
  - A tuple of **parents** (=references to other nodes)
  - A typed **value** (the type may differ for every node)

Therefore, depending on the use case, such a graph could represent just a key-value list, an 'any-type' container (container of things of varying types), a normal graph, a hierarchical graph, or an xml data structure.

We use the graph in particular also to define a generic file format, which we use for configuration (parameter) files, files that define robot kinematic and geometry, or any other structured data. This ascii file format of a graph helps to also understand the data structure. Here is the `example.g` from `test/Core/graph`:

```
## a trivial graph
x            # a 'vertex': key=x, value=true, parents=none
y            # another 'vertex': key=y, value=true, parents=none
(x y)        # an 'edge': key=none, value=true, parents=x y
x            # key=x, value=true, parents=none
y            # key=y, value=true, parents=none
(x y)        # key=none, value=true, parents=x y
(-1 -2)      # key=none, value=true, parents=the previous and the y-node

## a bit more verbose graph
node A{ color=blue }         # keys=node A, value=<Graph>, parents=none
node B{ color=red, value=5 } # keys=node B, value=<Graph>, parents=none
edge C(A,B){ width=2 }       # keys=edge C, value=<Graph>, parents=A B
hyperedge(A B C) = 5         # keys=hyperedge, value=5, parents=A B C

## standard value types
a=string      # MT::String (except for keywords 'true' and 'false' and 'Mod' and 'Include')
b="STRING"    # MT::String (does not require a '=')
c='file.txt'  # MT::FileToken (does not require a '=')
d=-0.1234     # double
e=[1 2 3 0.5] # MT::arr (does not require a '=')
#f=(c d e)    # DEPRECATED!! MT::Array<*Node> (list of other nodes in the Graph)
g!            # bool (default: true, !means false)
h=true        # bool
i=false       # bool
j={ a=0 }     # sub-Graph (special: does not require a '=')

## parsing: = {..} (..) , and \n are separators for parsing key-value-pairs
b0=false b1 b2, b3() b4   # 4 boolean nodes with keys 'b0', 'b1 b2', 'b3', 'b4'
k={ a, b=0.2 x="hallo"     # sub-Graph with 6 nodes
  y
  z()=filename.org x }

## special Node Keys

## editing: after reading all nodes, the Graph takes all Edit nodes, deletes the Edit tag, and calls a edit()
## this example will modify/append the respective attributes of k
Edit k { y=false, z=otherString, b=7, c=newAttribute }

## including
Include = 'example_include.g'   # first creates a normal FileToken node then opens and includes the file directly

## any types
#trans=<T t(10 0 0)>  # 'T' is the tag for an arbitrary type (here an ors::Transformation)
                      # which was registered somewhere in the code using the registry()
                      # (does not require a '=')

## strange notations
a()       # key=a, value=true, parents=none
()        # key=none, value=true, parents=none
[1 2 3 4] # key=none, value=MT::arr, parents=none
```

A special case is when a node has a Graph-typed value. This is considered a **subgraph**. Subgraphs are sometimes handled special: their nodes can have parents from the containing graph, or from other subgraphs of the containing graph. Some methods of the `Graph` class (to find nodes by key or type) allow to specify whether also nodes in subgraphs or parentgraphs are to be searched.

3 Logic – A graph implementation of First Order Logic

We represent everything, a full knowledge base (KB), as a graph:

- Symbols (both, constants and predicate symbols) are nil-valued nodes. We assume that they are declared in the root scope of the graph
- A grounded literal is a tuple of symbols, for instance `(on box1 box2)`. Note that we can equally write this as `(box1 on box2)`. There is no need to have the 'predicate' first. In fact, the basic methods do not distinguish between predicate and contant symbols.
- A universal quantification $\forall X$ is represented as a scope (=subgraph) which first declares the logic variables as nil-valued nodes as the subgraph, then the rest. The rest is typically an implication, i.e., a rule. For instance

  $$\forall XY\ p(X,Y)q(Y) \Rightarrow q(X)$$

  is represented as `{X, Y, { (p X Y) (q Y) } { (q X) }` where the precondition and postconditions are subgraphs of the rule-subgraph.

Here is how the standard FOL example from Stuart Russell's lecture is represented:

```
Constant M1
Constant M2
Constant Nono
Constant America
Constant West

American
Weapon
Sells
Hostile
Criminal
Missile
Owns
Enemy

STATE {
(Owns Nono M1),
(Missile M1),
(Missile M2),
(American West),
(Enemy Nono America)
}

Query { (Criminal West) }

Rule {
x, y, z,
{ (American x) (Weapon y) (Sells x y z) (Hostile z) }
{ (Criminal x) }
}

Rule {
x
{ (Missile x) (Owns Nono x) }
{ (Sells West x Nono) }
}

Rule {
x
{ (Missile x) }
{ (Weapon x) }
}
```

```
Rule {
x
{ (Enemy x America) }
{ (Hostile x) }
}
```

By default all tuples in the graph are boolean-valued with default value true. In the above example all literals are actually true-valued. A rule `{X, Y, { (p X Y) (q Y) } { (q X)! }` means $\forall XY\ p(X,Y)q(Y) \Rightarrow \neg q(X)$. If in the KB we only store true facts, this would 'delete' the fact `(q X)!` from the KB (for some X).

As nodes of our graph can be of any type, we can represent predicates of any type, for instance `{X, Y, { (p X Y) (q Y)=3 } { (q X)=4 }` would let $p(X)$ be double-typed.

3.1 Methods

The most important methods are the following:

- Checking whether **two facts are equal**. Facts are grounded literals. Equality is simply checked by checking if all symbols (predicate or constant) in the tuples are equal. Optionally (by default), it is also checked if the fact values are equal.
- Checking whether **a fact equals a literal+substitution**. The literal is a tuple of symbols, some of which may be first order variables. All variables must be of the same scope (=declared in the same subgraph, in the same rule). The substitution is a mapping of these variables to root-level symbols (predicate of constant symbols). The methods loops through the literal's symbols; whenever a symbol is in the substitution scope it replaces it by the substitution; then compares to the fact symbol. Optionally (by default) also the value is checked for equality.
- Check whether **a fact is directly true in a KB (or scope)** (without inference). This uses the graph connectivity to quickly find any KB-fact that shares the same symbols; then checks these for exact equality.
- Check whether **a literal+substitution is directly true in a KB** (without inference).
- Given a single literal with only ONE logic variable, and a KB of facts, **compute the domain** (=possible values of the variable) for the literal to be true. If the literal is negated the $D \leftarrow D \setminus d$, otherwise $D \leftarrow D \cup d$ if the d is the domain for true facts in the KB. [TODO: do this also for multi-variable literals]
- **Compute the set of possible substitutions for a conjunction of literals** (typically precondition of a rule) to be true in a KB.
- **Apply a set of 'effect literals'** (RHS of a rule): generate facts that are substituted literals

Given these methods, forward chaining, or forward simulation (for MCTS) is straightforward.

4 Optim – data structures to represent non-linear programs, and basic solvers

The optimization code contributes a nice way to represent (structured) optimization problems, and a few basic solvers for constrained and unconstrained, black-box, gradient-, and

hessian-available problems.

4.1 Representing Optimization Problems

The data structures to represent optimization problems are

A **standard unconstrained problem**

$$\min_x f(x) \tag{1}$$

```
typedef std::function<double(arr& df, arr& Hf, const arr& x)> ScalarFunction;
```

The double return value is $f(x)$. The gradient `df` is returned only on request (for `df!=NoArr`). The hessian `Hf` is returned only on request. If the caller requests `df` or `Hf` but the implementation cannot compute gradient or hessian, the implementation should `HALT`.

These can be implemented using a lambda expression or setting it equal to a C-function. See the examples.

A **sum-of-squares problem** (Gauss-Newton type)

$$\min_x y(x)^\top y(x) \tag{2}$$

```
typedef std::function<void(arr& y, arr& Jy, const arr& x)> VectorFunction;
```

where the vector `y` must always be returned. The Jacobian `Jy` is returned on request (`Jy!=NoArr`).

A **constrained problem** (for vector valued functions f, y, g, h)

$$\min_x \sum_i f_i(x) + y(x)^\top y(x) \quad \text{s.t.} \quad g(x) \le 0\,, \quad h(x) = 0 \tag{3}$$

Note that we can rewrite this as

$$\min_x \sum_{t\in F} \phi_t(x) + \sum_{t\in Y} \phi_t(x)^\top \phi_t(x) \quad \text{s.t.} \quad \forall_{t\in G}:\ \phi_t(x) \le 0\,, \quad \forall_{t\in H}:\ \phi_t(x) = 0\,, \tag{4}$$

where the vector-valued *feature* function ϕ contains all f_i, y_i, g_i, h_i, and the disjoint partition $F \cup Y \cup G \cup H = \{1, .., T\}$ indicates whether the t-th feature contributes a scalar objective, sum-of-square objective, inequality constraint or equality constraint. We represent this as

```
enum TermType { noTT=0, fTT, sumOfSqrTT, ineqTT, eqTT };
typedef mlr::Array<TermType> TermTypeA;
struct ConstrainedProblem{
  virtual ~ConstrainedProblem() = default;
  virtual void phi(arr& phi, arr& J, arr& H, TermTypeA& tt, const arr& x) = 0;
};
```

Here, the returned `phi` is the feature vector and the returned `tt` indicates for every `phi`-entry its type (`fTT, sumOfSqrTT, ineqTT, eqTT`). The (on request) returned `J` is the Jacobian of `phi`. The (on request) returned `H` is the Hessian *of the scalar features only*, that is, the Hessian of $\sum_i f_i(x)$.

A **structured constrained problem**: Assume we have N decision variables $x_i \in \mathbb{R}^{d_i}$, each with its own dimensionality d_i. Assume we have J features $\phi_{j=1,..,J}$, but each feature ϕ_j depends on only a tuples $X_j \subseteq \{x_1, .., x_N\}$ of variables. We minimize

$$\min_{x_{1:N}} \sum_{j\in F} \phi_j(X_j) + \sum_{j\in Y} \phi_j(X_j)^\top \phi_j(X_j) \quad \text{s.t.} \quad \forall_{j\in G}:\ \phi_j(X_j) \le 0\,, \quad \forall_{j\in H}:\ \phi_j(X_j) = 0\,. \tag{5}$$

```
struct GraphProblem {
  virtual void getStructure(uintA& variableDimensions, uintAA& featureVariables, TermTypeA& feature
  virtual void phi(arr& phi, arrA& J, arrA& H, const arr& x) = 0;
};
```

Here we decided to provide a method that first allows the optimizer to query the structure of the problem: return N, $d_{i=1,..,N}$, J, $X_{t=1,..,J}$, and $\texttt{tt}_{i=1,..,J}$. This allows the optimizer to setup its own data structures or so. Then, in each iteration the optimizer only queries `phi(...)`. This always returns the J-dimensional feature vector `phi`, which contains an f_i, y_i, g_i or h_i-value, depending on `tt(j)`. This `phi(j)` may only depend on the decision variables X_j. On request it returns the gradient `J(j)` of `phi(j)` w.r.t. X_j. Note that the dimensionality of X_j may vary—therefore we return an array of gradients instead of a Jacobian. On request also a hessian `H(j)` is returned for the scalar objectives (when `tt(j)==fTT`).

A **k-order Markov Optimization** problem. We have T decision variables $x_{1,..,T}$, each with potentially different dimensionality $d_{1,..,T}$. We have J features $\phi_{1,..,J}$, each of which may only depend on $k+1$ consecutive variables $X_j = (x_{t_j-k}, .., x_{t_j})$, where t_j tells us which $k+1$-tuple feature ϕ_j depends on. We minimize again (5). For easier readibility, this is equivalent to a problem of the form:

$$\min_{x_{1:T}} \sum_{t=1}^{T} y_t(x_{t-k:t})^\top y_t(x_{t-k:t}) \quad \text{s.t.} \quad \forall_t:\ g_t(x_{t-k:t}) \le 0\,, \quad h_t(x_{t-k:t}) = 0\,, \tag{6}$$

where the feature with same $t_k = t$ have been collected in different vector-value functions y_t, g_t, h_t.

```
struct KOMO_Problem {
  virtual uint get_k() = 0;
  virtual void getStructure(uintA& variableDimensions, uintA& featureTimes, TermTypeA& featureTypes
  virtual void phi(arr& phi, arrA& J, arrA& H, TermTypeA& tt, const arr& x) = 0;
};
```

Here, the structure function returns N, $d_{1,..,N}$, J, t_j, `tt(j)`.

5 Kinematics – data structures to represent kinematic configurations, interface models/optimizers/simulators, represent task spaces

5.1 Task Spaces

5.1.1 Purpose

Task spaces are defined by a mapping $\phi : q \to y$ from the joint state $q \in \mathbb{R}^n$ to a task space $y \in \mathbb{R}^m$. They are central in designing motion and manipulation, both, in the context of trajectory optimization as well as in specifying position/force/impedance controllers:

For **trajectory optimization**, cost functions are defined by costs or constraints in task spaces. Given a single task space ϕ, we may define

- costs $\|\phi(q)\|^2$,
- an inequality constraint $\phi(q) \leq 0$ (element-wise inequality),
- an equality constraint $\phi(q) = 0$.

All three assume that the 'target' is at zero. For convenience, the code allows to specify an additional linear transform $\tilde{\phi}(q) \leftarrow \rho(\phi(q) - y_{\text{ref}})$, defined by a target reference y_{ref} and a scaling ρ. In KOMO, costs and constraints can also be defined on $k+1$-tuples of consecutive states in task space, allowing to have cost and constraints on task space velocities or accelerations. Trajectory optimization problems are defined by many such costs/constraints in various task spaces at various time steps.

For simple **feedback control**, in each task space we may have

- a desired linear acceleration behavior in the task space
- a desired force or force constraint (upper bound) in the task space
- a desired impedance around a reference

All of these can be fused to a joint-level force-feedback controller (details, see controller docu). On the higher level, the control mode is specified by defining multiple task spaces and the desired behaviors in these. (The activity of such tasks (on the symbolic level) is controlled by the RelationalMachine, see its docu.)

In both cases, defining task spaces is the core.

5.1.2 Basic notation

We follow the notation in the robotics lecture slides. We enumerate all bodies by $i \in \mathcal{B}$. We typically use $v, w \in \mathbb{R}^3$ to denote vectors attached to bodies. $T_{W\to i}$ is the 4-by-4 homogeneous transform from world frame to the frame of shape i, and $R_{W\to i}$ is its rotation matrix only. In the text (not in equations) we sometimes write

$$(i + v)$$

where i denotes a body and $v \in \mathbb{R}^3$ a relative 3D-vector. The rigorous notation for this would be $T_{W\to i}v$, which is the position of i plus the relative displacement v.

To numerically evaluate kinematics we assume that, for a certain joint configuration q, the positions p_k and axes a_k of all joints $k \in JJ$ have been precomputed. The boolean expression

$$[k \prec i]$$

iff joint k is "below" body i in the kinematic tree, that is, joint k is between root and body i and therefore moves it.

Sometimes we write $J_{\cdot k} = ...$, which means that the kth column of J is defined as given. Let's first define

$$(A_i)_{\cdot k} = [k \prec i][i \text{ rotational}]\ a_k \qquad \text{axes matrix below } i \tag{7}$$

This matrix contains all rotational axes below i as columns and will turn out convenient, because it captures all axes that make i move. Many Jacobians can easily be described using A_i. Analogously we define

$$(T_i)_{\cdot k} = [k \prec i][i \text{ prismatic}]\ a_k \qquad \text{prism matrix below } i \tag{8}$$

This captures all prismatic joints. Note the following relation to Featherstone's notation: In his notation, $h_k \in \mathbb{R}^6$ denotes, for very joint k, how much the joints contributes to rotation and translation, expressed in the link frame. The two matrices A_i and T_i together express the same, but in world coordinates. While typically axes have unit length (and entries of h are zeros or ones), this is not necessary in general, allowing for arbitrary scaling of joint configurations q with these axis (e.g., using degree instead of radial units).

For convenience, for a matrix of 3D columns $A \in \mathbb{R}^{n\times 3}$, we write

$$B = A \times p \iff B_{\cdot k} = A_{\cdot k} \times p$$

which is the column-wise cross product. Also, we define

$$(\hat{A}_i)_{\cdot k} = [k \prec i][i \text{ rotational}]\ a_k \times p_k \qquad \text{axes position matrix below } i \tag{9}$$

which could also be written as $\hat{A}_i = A_i \times P$ if P contains all axes positions.

5.1.3 Task Spaces

Position

$$\phi^{\mathsf{p}}_{iv}(q) = T_{W\to i}\ v \qquad \text{position of } (i+v) \tag{10}$$
$$J^{\mathsf{p}}_{iv}(q)_{\cdot k} = [k \prec i]\ a_k \times (\phi^{\mathsf{p}}_{iv}(q) - p_k) \tag{11}$$
$$J^{\mathsf{p}}_{iv}(q) = A_i \times \phi^{\mathsf{p}}_{iv}(q) - \hat{A}_i \tag{12}$$
$$\phi^{\mathsf{p}}_{iv-jw}(q) = \phi^{\mathsf{p}}_{iv} - \phi^{\mathsf{p}}_{jw} \qquad \text{position difference} \tag{13}$$
$$J^{\mathsf{p}}_{iv-jw}(q) = J^{\mathsf{p}}_{iv} - J^{\mathsf{p}}_{jw} \tag{14}$$
$$\phi^{\mathsf{p}}_{iv|jw}(q) = R_j^{-1}(\phi^{\mathsf{p}}_{iv} - \phi^{\mathsf{p}}_{jw}) \qquad \text{relative position} \tag{15}$$
$$J^{\mathsf{p}}_{iv|jw}(q) = R_j^{-1}[J^{\mathsf{p}}_{iv} - J^{\mathsf{p}}_{jw} - A_j \times (\phi^{\mathsf{p}}_{iv} - \phi^{\mathsf{p}}_{jw})] \tag{16}$$

Derivation: For $y = Rp$ the derivative w.r.t. a rotation around axis a is $y' = Rp' + R'p = Rp' + a \times Rp$. For $y = R^{-1}p$ the derivative is $y' = R^{-1}p' - R^{-1}(R')R^{-1}p = R^{-1}(p' - a \times p)$. (For details see http://ipvs.informatik.uni-stuttgart.de/mlr/marc/notes/3d-geometry.pdf)

Vector

$$\phi^{\mathsf{v}}_{iv}(q) = R_{W\to i}\ v \qquad \text{vector} \tag{17}$$
$$J^{\mathsf{v}}_{iv}(q) = A_i \times \phi^{\mathsf{v}}_{iv}(q) \tag{18}$$
$$\phi^{\mathsf{v}}_{iv-jw}(q) = \phi^{\mathsf{v}}_{iv} - \phi^{\mathsf{v}}_{jw} \qquad \text{vector difference} \tag{19}$$
$$J^{\mathsf{v}}_{iv-jw}(q) = J^{\mathsf{v}}_{iv} - J^{\mathsf{v}}_{jw} \tag{20}$$
$$\phi^{\mathsf{v}}_{iv|j}(q) = R_j^{-1}\phi^{\mathsf{v}}_{iv} \qquad \text{relative vector} \tag{21}$$
$$J^{\mathsf{v}}_{iv|j}(q) = R_j^{-1}[J^{\mathsf{v}}_{iv} - A_j \times \phi^{\mathsf{v}}_{iv}] \tag{22}$$

Quaternion See equation (15) in the geometry notes for explaining the jacobian.

$$\phi_i^{\mathsf{q}}(q) = \text{quaternion}(R_{W\to i}) \qquad \text{quaternion} \in \mathbb{R}^4 \tag{23}$$

$$J_i^{\mathsf{q}}(q)_{\cdot k} = \begin{pmatrix} 0 \\ \frac{1}{2}(A_i)_{\cdot k} \end{pmatrix} \circ \phi_i^{\mathsf{q}}(q) \qquad J_i^{\mathsf{q}}(q) \in \mathbb{R}^{4\times n} \tag{24}$$

$$\phi_{i-j}^{\mathsf{q}}(q) = \phi_i^{\mathsf{q}} - \phi_j^{\mathsf{q}} \qquad \text{difference} \in \mathbb{R}^4 \tag{25}$$

$$J_{i-j}^{\mathsf{q}}(q) = J_i^{\mathsf{q}} - J_j^{\mathsf{q}} \tag{26}$$

$$\phi_{i|j}^{\mathsf{q}}(q) = (\phi_j^{\mathsf{q}})^{\text{-1}} \circ \phi_i^{\mathsf{q}} \qquad \text{relative} \tag{27}$$

$$J_{i|j}^{\mathsf{q}}(q) = \text{not implemented} \tag{28}$$

A relative rotation can also be measured in terms of the 3D rotation vector. Lets define

$$w(r) = \frac{2\phi}{\sin(\phi)}\bar{r}\,, \quad \phi = \text{acos}(r_0)$$

as the rotation for a quaternion. We have

$$\phi_{i|j}^{\mathsf{w}}(q) = w(\phi_j^{\mathsf{q}})^{\text{-1}} \circ \phi_i^{\mathsf{q}}) \qquad \text{relative rotation vector} \in \mathbb{R}^3 \tag{29}$$

$$J_{i|j}^{\mathsf{w}}(q) = AJ_i^{\mathsf{q}} - J_j^{\mathsf{q}} \tag{30}$$

$$\tag{31}$$

Alignment parameters: shape indices i, j, attached vectors v, w

$\phi_{iv|jw}^{\text{align}}(q) = (\phi_{jw}^{\mathsf{v}})^\top \phi_{iv}^{\mathsf{v}}$

$J_{iv|jw}^{\text{align}}(q) = (\phi_{jw}^{\mathsf{v}})^\top J_{iv}^{\mathsf{v}} + \phi_{iv}^{\mathsf{v}\,\top} J_{jw}^{\mathsf{v}}$

Note: $\phi^{\text{align}} = 1 \leftrightarrow$ align $\phi^{\text{align}} = -1 \leftrightarrow$ anti-align $\phi^{\text{align}} = 0 \leftrightarrow$ orthog.

Gaze 2D orthogonality measure of object relative to camera plane

parameters: eye index i with offset v; target index j with offset w

$$\phi_{iv,jw}^{\text{gaze}}(q) = \begin{pmatrix} \phi_{i,e_x}^{\mathsf{v}\,\top}(\phi_{jw}^{\mathsf{p}} - \phi_{iv}^{\mathsf{p}}) \\ \phi_{i,e_y}^{\mathsf{v}\,\top}(\phi_{jw}^{\mathsf{p}} - \phi_{iv}^{\mathsf{p}}) \end{pmatrix} \quad \in \mathbb{R}^2 \tag{32}$$

Here $e_x = (1,0,0)$ and $e_y = (0,1,0)$ are the camera plane axes.

Jacobians straight-forward

qItself $\phi_{iv,jw}^{\text{qItself}}(q) = q$

Joint limits measure parameters: joint limits $q_{\text{low}}, q_{\text{hi}}$, margin m

$\phi_{\text{limits}}(q) = \frac{1}{m}\sum_{i=1}^{n}[m - q_i + q_{\text{low}}]^+ + [m + q_i - q_{\text{hi}}]^+$

$J_{\text{limits}}(q)_{1,i} = -\frac{1}{m}[m - q_i + q_{\text{low}} > 0] + \frac{1}{m}[m + q_i - q_{\text{hi}} > 0]$

$[x]^+ = x > 0?x : 0 \qquad [\cdots]$: indicator function

Collision limits measure parameters: margin m

$\phi_{\text{col}}(q) = \frac{1}{m}\sum_{k=1}^{K}[m - |p_k^a - p_k^b|]^+$

$J_{\text{col}}(q) = \frac{1}{m}\sum_{k=1}^{K}[m - |p_k^a - p_k^b| > 0](-J_{p_k^a}^{\mathsf{p}} + J_{p_k^b}^{\mathsf{p}})^\top \frac{p_k^a - p_k^b}{|p_k^a - p_k^b|}$

A collision detection engine returns a set $\{(a, b, p^a, p^b)_{k=1}^K\}$ of potential collisions between shape a_k and b_k, with nearest points p_k^a on a and p_k^b on b.

Shape distance measures (using SWIFT)

- allPTMT, //phi=sum over all proxies (as is standard)
- listedVsListedPTMT, //phi=sum over all proxies between listed shapes
- allVsListedPTMT, //phi=sum over all proxies against listed shapes
- allExceptListedPTMT, //as above, but excluding listed shapes
- bipartitePTMT, //sum over proxies between the two sets of shapes (shapes, shapes2)
- pairsPTMT, //sum over proxies of explicitly listed pairs (shapes is n-times-2)
- allExceptPairsPTMT, //sum excluding these pairs
- vectorPTMT //vector of all pair proxies (this is the only case where dim(phi)¿1)

GJK pairwise shape distance (including negative)

Plane distance

5.1.4 Application

Just get a glimpse on how task space definitions are used to script motions, here is a script of a little PR2 dance. (The 'logic' below the script implements kind of macros – that's part of the RAP.) (`wheels` is the same as qItself, but refers only to the 3 base coordinates)

```
cleanAll #this only declares a novel symbol...

Script {
  (FollowReferenceActivity wheels){ type=wheels, target=[0, .3, .2], PD=[.5, .9, .5, 10.]}
  (MyTask endeffR){ type=pos, ref2=base_footprint, target=[.2, -.5, 1.3], PD=[.5, .9, .5, 10.]}
  (MyTask endeffL){ type=pos, ref2=base_footprint, target=[.2, +.5, 1.3], PD=[.5, .9, .5, 10.]}
  { (conv FollowReferenceActivity wheels)  (conv MyTask endeffR) }  #this waits for convergence of activities
  (cleanAll) #this switches off the current activities
  (cleanAll)! #this switches off the switching-off

  (FollowReferenceActivity wheels){ type=wheels, target=[0, -.3, -.2], PD=[.5, .9, .5, 10.]}
  (MyTask endeffR){ type=pos, ref2=base_footprint, target=[.7, -.2, .7], PD=[.5, .9, .5, 10.]}
  (MyTask endeffL){ type=pos, ref2=base_footprint, target=[.7, +.2, .7], PD=[.5, .9, .5, 10.]}
  { (conv FollowReferenceActivity wheels)  (conv MyTask endeffL) }
  (cleanAll)
  (cleanAll)!

  (FollowReferenceActivity wheels){ type=wheels, target=[0, .3, .2], PD=[.5, .9, .5, 10.]}
  (MyTask endeffR){ type=pos, ref2=base_footprint, target=[.2, -.5, 1.3], PD=[.5, .9, .5, 10.]}
  (MyTask endeffL){ type=pos, ref2=base_footprint, target=[.2, +.5, 1.3], PD=[.5, .9, .5, 10.]}
  { (conv MyTask endeffL) }
  (cleanAll)
  (cleanAll)!

  (FollowReferenceActivity wheels){ type=wheels, target=[0, 0, 0], PD=[.5, .9, .5, 10.]}
  (HomingActivity)
  { (conv HomingActivity) (conv FollowReferenceActivity wheels) }
}


Rule {
    X, Y,
    { (cleanAll) (conv X Y) }
    { (conv X Y)! }
}

Rule {
    X,
    { (cleanAll) (MyTask X) }
    { (MyTask X)! }
}

Rule {
    X,
    { (cleanAll) (FollowReferenceActivity X) }
    { (FollowReferenceActivity X)! }
}
```

6 KOMO

I do not introduce the KOMO concepts here. Read this[1] http://ipvs.informatik.uni-stuttgart.de/mlr/papers/16-toussaint-Newton.pdf !

The goal of the implementation is the separation between the code of optimizers and code to specify motion problems. The problem form (6) provides the abstraction for that interface. The optimization methods all assume the general form

$$\min_x f(x) \quad \text{s.t.} \quad g(x) \le 0\,, \quad h(x) = 0 \tag{33}$$

of a non-linear constrained optimization problem, with the additional assumption that the (approximate) Hessian $\nabla^2 f(x)$ can be provided and is semi-pos-def. Therefore, the KOMO code essentially does the following

- Provide interfaces to define sets of k-order task spaces and costs/constraints in these task spaces at various time slices; which constitutes a MotionProblem. Such a MotionProblem definition is very semantic, referring to the kinematics of the robot.
- Abstracts and converts a MotionProblem definition into the general form (6) using a kinematics engine. The resulting MotionProblemFunction is not semantic anymore and provides the interface to the generic optimization code.
- Converts the problem definition (6) into the general forms (3) and (33) using appropriate matrix packings to exploit the chain structure of the problem. This code does not refer to any robotics or kinematics anymore.
- Applies various optimizers. This is generic code.

The code introduces specialized matrix packings to exploit the structure of J and to efficiently compute the banded matrix $J^\top J$. Note that the rows of J have at most $(k+1)n$ non-zero elements since a row refers to exactly one task and depends only on one specific tuple $(x_{t-k}, .., x_t)$. Therefore, although J is generally a $D \times (T+1)n$ matrix (with $D = \sum_t \dim(f_t)$), each row can be packed to store only $(k+1)n$ non-zero elements. We introduced a *row-shifted* matrix packing representation for this. Using specialized methods to compute $J^\top J$ and $J^\top x$ for any vector x for the row-shifted packing, we can efficiently compute the banded Hessian and any other terms we need in Gauss-Newton methods.

6.1 Formal problem representation

The following definitions also document the API of the code.

KinematicEngine is a mapping $\Gamma : \; x \mapsto \Gamma(x)$ that maps a joint configuration to a data structure $\Gamma(x)$ which allows to efficiently evaluate task maps. Typically $\Gamma(x)$ stores the frames of all bodies/shapes/objects and collision information. More abstractly, $\Gamma(x)$ is any data structure that is sufficient to define the task maps below.

Note: In the code there is yet no abstraction KinematicEngine. Only one specific engine (KinematicWorld) is used. It would be straight-forward to introduce an abstraction for kinematic engines pin-pointing exactly their role for defining task maps.

[1] M. Toussaint: A tutorial on Newton methods for constrained trajectory optimization and relations to SLAM, Gaussian Process smoothing, optimal control, and probabilistic inference. In Geometric and Numerical Foundations of Movements, Springer, 2016.

TaskMap is a mapping $\phi : (\Gamma_{-k}, .., \Gamma_0) \mapsto (y, J)$ which gets $k+1$ kinematic data structures as input and returns some vector $y \in \mathbb{R}^d$ and (on request) its Jacobian $J \in \mathbb{R}(d \times n)$.

Task is a tuple $c = (\phi, \varrho_{1:T}, y^*_{1:T}, \mathsf{tt})$ where ϕ is a TaskMap and the parameters $\varrho_{1:T}, y^*_{1:T} \in \mathbb{R}^{T\times d}$ allow for an additional linear transformation in each time slice. Here, $d = \dim(\phi)$ is the dimensionality of the task map. This defines the transformed task map

$$\hat{\phi}_t(x_{t-k}, .., x_t) = \mathrm{diag}(\varrho_t)(\phi(\Gamma(x_{t-k}), .., \Gamma(x_t)) - y^*_t) \;, \tag{34}$$

which depending on $\mathsf{tt} \in \{$fTT, sumOfSqrTT, ineqT, eqT$\}$ is interpreted as cost or constraint feature. Note that, in the cost case, $y^*_{1:T}$ has the semantics of a reference target for the task variable, and $\varrho^*_{1:T}$ of a precision. In the code, $\varrho_{1:T}, y^*_{1:T}$ may optionally be given as 1×1, $1\times T$+1, $d\times 1$, or $d\times T$+1 matrices—and are interpreted constant along the missing dimensions.

MotionProblem is a tuple $(T, \mathcal{C}, x_{-k+1:0})$ which gives the number of time steps, a list $\mathcal{C} = \{c_i\}$ of Tasks, and a *prefix* $x_{-k:-1} \in \mathbb{R}^{k\times n}$. The prefix allows to evaluate tasks also for time $t \leq k$, where the prefix defines the kinematic configurations $\Gamma(x_{-k+1}), .., \Gamma(x_0)$ at negative times. This defines the KOMO problem.

6.2 User Interfaces

6.2.1 Easy

For convenience there is a single high-level method to call the optimization, defined in <Motion/komo.h>

```
/// Return a trajectory that moves the endeffector to a desired target position
arr moveTo(ors::KinematicWorld& world, //in initial state
           ors::Shape& endeff,         //endeffector to be moved
           ors::Shape& target,         //target shape
           byte whichAxesToAlign=0,    //bit coded options to align axes
           uint iterate=1);            //usually the optimization methods may be called just
                                       //once; multiple calls -> safety
```

The method returns an optimized joint space trajectory so that the endeff reaches the target. Optionally the optimizer additionaly aligns some axes between the coordinate frames. This is just one typical use case; others would include constraining vector-alignments to zero (orthogonal) instead of +1 (parallel), or directly specifying quaternions, or using many other existing task maps. See expert interface.

This interface specifies the relevant coordinate frames by referring to Shapes. Shapes (`ors::Shape`) are rigidly attached to bodies ("links") and usually represent a (convex) collision mesh/primitive. However, a Shape can also just be a marker frame (`ShapeType markerST=5`), in which case it is just a convenience to define reference frames attached to bodies. So, the best way to determine the geometric parameters of the endeffector and target (offsets, relative orientations etc) is by transforming the respective shape frames (`Shape::rel`).

The method uses implicit parameters (grabbed from cfg file or command line or default):

```
double posPrec = MT::getParameter<double>("KOMO/moveTo/precision", 1e3);
double colPrec = MT::getParameter<double>("KOMO/moveTo/collisionPrecision", -1e0);
double margin = MT::getParameter<double>("KOMO/moveTo/collisionMargin", .1);
double zeroVelPrec = MT::getParameter<double>("KOMO/moveTo/finalVelocityZeroPrecision", 1e1);
double alignPrec = MT::getParameter<double>("KOMO/moveTo/alignPrecision", 1e3);
```

6.2.2 Using a specs file

Example:

```
KOMO{
  T = 100
  duration = 5
}

Task sqrAccelerations{
  map={ type=qItself }
  order=2    # accelerations (default is 0)
  time=[0 1] # from start to end (default is [0 1])
  type=cost  # squared costs (default is 'cost')
  scale=1    # factor of the map (default is [1])
  target=[0] # offset of the map (default is [0])
}

Task finalHandPosition{
  map={ type=pos ref1=hand ref2=obj vec1=[0 0 .1] }
  time=[1 1] # only final
  type=equal # hard equality constraint
}

Task finalAlignmentPosition{
  map={ type=vecAlign ref1=hand vec1=[1 0 0] vec2=[0 1 0]}
  time=[1 1] # only final
  type=equal # hard equality constraint
  target=[1] # scalar product between vec1@hand and vec2@world shall be 1
}

Task collisions{
  map={ type=collisionIneq margin=0.05 }
  type=inEq # hard inequality constraint
}
```

6.2.3 Expert using the included kinematics engine

See the implementation of `moveTo`! This really is the core guide to build your own cost functions.

More generically, if the user would like to implement new TaskMaps or use some of the existing ones:

- The user can define new k-order task maps by instantiating the abstraction. There exist a number of predefined task maps. The specification of a task map usually has only a few parameters like "which endeffector shape(s) are you referring to". Typically, a good convention is to define task maps in a way such that *zero* is a desired state or the constraint boundary, such as relative coordinates, alignments or orientation. (But that is not necessary, see the linear transformation below.)
- To define an optimization problem, the user creates a list of tasks, where each task is defined by a task map and parameters that define how the map is interpreted as a) a cost term or b) an inequality constraint. This interpretation allows: a linear transformation separately for each t (=setting a reference/target and precision); how maps imply a constraint. This interpretation has a significant number of parameters: for each time slice different targets/precisions could be defined.

6.2.4 Expert with own kinematics engine

The code needs a data structure $\Gamma(q_t)$ to represent the (kinematic) state q_t, where coordinate frames of all bodies/shapes/objects have been precomputed so that evaluation of task maps is fast. Currently this is `KinematicWorld`.

Users that prefer using the own kinematics engine can instantiate the abstraction. Note that the engine needs to fulfill two roles: it must have a `setJointState` method that also precomputes all frames of all bodies/shapes/objects. And it must be siffucient as argument of your task map instantiations.

6.2.5 Optimizers

The user can also only use the optimizers, directly instantiating the k-order Markov problem abstraction; or, yet a level below, directly instantiating the `ConstrainedProblem` abstraction. Examples are given in `examples/Optim/kOrderMarkov` and `examples/Optim/constrained`. Have a look at the specific implementations of the benchmark problems, esp. the `ParticleAroundWalls` problem.

6.2.6 Parameters & Reporting

Every run of the code generates a MT.log file, which tells about every parameter that was internally used. You can overwrite any of these parameters on command line or in an MT.cfg file.

Inspecting the cost report after an optimization is important. Currently, the code goes through the task list $\mathcal{C}$ and reports for each the costs associated to it. There are also methods to display the cost arising in the different tasks over time.

6.3 Potential Improvements

There is many places the code code be improved (beyond documenting it better):

- The KinematicEngine should be abstracted to allow for easier plugin of alternative engines.
- Our kinematics engine uses SWIFT++ for proximity and penetration computation. The methods would profit enormously from better (faster, more accurate) proximity engines (signed distance functions, sphere-swept primitives).

6.4 Disclaimer

This document by no means aims to document all aspects of the code, esp. those relating to the used kinematics engine etc. It only tries to introduce to the concepts and design decisions behind the KOMO code.

More documentation of optimization and kinematics concepts used in the code can be drawn from my teaching lectures on Optimization and Robotics.