Guide to the MLR Code

M Toussaint

March 27, 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	$\text{diag}(a)$	`diag(a)`
	$\text{diag}(a)B$ 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)

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 '=')
```

[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.

```
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.

$$\phi^{\mathsf{q}}_i(q) = \text{quaternion}(R_{W\to i}) \qquad \text{quaternion} \in \mathbb{R}^4 \tag{23}$$
$$J^{\mathsf{q}}_i(q)_{\cdot k} = \begin{pmatrix} 0 \\ \frac{1}{2}(A_i)_{\cdot k} \end{pmatrix} \circ \phi^{\mathsf{q}}_i(q) \qquad J^{\mathsf{q}}_i(q) \in \mathbb{R}^{4\times n} \tag{24}$$
$$\phi^{\mathsf{q}}_{i-j}(q) = \phi^{\mathsf{q}}_i - \phi^{\mathsf{q}}_j \qquad \text{difference} \in \mathbb{R}^4 \tag{25}$$
$$J^{\mathsf{q}}_{i-j}(q) = J^{\mathsf{q}}_i - J^{\mathsf{q}}_j \tag{26}$$
$$\phi^{\mathsf{q}}_{i|j}(q) = (\phi^{\mathsf{q}}_j)^{-1} \circ \phi^{\mathsf{q}}_i \qquad \text{relative} \tag{27}$$
$$J^{\mathsf{q}}_{i|j}(q) = \text{not implemented} \tag{28}$$

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

$$\phi^{\mathsf{w}}_{i|j}(q) = w(\phi^{\mathsf{q}}_j)^{-1} \circ \phi^{\mathsf{q}}_i) \qquad \text{relative rotation vector} \in \mathbb{R}^3 \tag{29}$$
$$J^{\mathsf{w}}_{i|j}(q) = AJ^{\mathsf{q}}_i - J^{\mathsf{q}}_j \tag{30}$$
$$\tag{31}$$

Alignment parameters: shape indices i, j, attached vectors v, w
$\phi^{\text{align}}_{iv|jw}(q) = (\phi^{\mathsf{v}}_{jw})^\top \phi^{\mathsf{v}}_{iv}$
$J^{\text{align}}_{iv|jw}(q) = (\phi^{\mathsf{v}}_{jw})^\top J^{\mathsf{v}}_{iv} + \phi^{\mathsf{v}\top}_{iv} J^{\mathsf{v}}_{jw}$
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

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

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

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

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

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

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

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.

[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.

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.

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

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

which depending on $\mathsf{tt} \in \{\mathsf{fTT}, \mathsf{sumOfSqrTT}, \mathsf{ineqT}, \mathsf{eqT}\}$ 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.

7 Control

7.1 Slow and fast control loop

There are two nested control loops:

In the slow loop ($\sim$ 50Hz, non-strict, non-real-time) the controller has full access to the results of pre-computed optimizations, full models of the robots kinematics (dynamics?) and potentially delayed information on the robot (current pose, forces, contacts, etc). The slow loop may realize computationally complex things, e.g., operational space control, re-adaptation of a plan (phase adaptation, recalibration of task maps), model predictive control, online planning, etc.

The fast loop is 1kHz, strictly and real-time. It has direct access to the current robot state q (needs to compute $\dot{q}$ from filtered differentiation of q) as well as the current readouts of the F/T sensors u_{ft}. **We constrain the fast controller to be a linear regulator in these observables and their integral:**

$$e \leftarrow \gamma e + (f^* - J_{\text{ft}}^\dagger u_{\text{ft}}) \quad \text{or} \quad \dot{e} = (f^* - J_{\text{ft}}^\dagger u_{\text{ft}}) + (1-\gamma)e \tag{35}$$

$$u = u_0 + k_p^{\text{base}} \cdot K_p(q^* - q) + k_d^{\text{base}} \cdot K_d(\dot{q}^* - \dot{q}) + K_I e \; . \tag{36}$$

This a (redundant) parameterization of a regulator linear in $(q, \dot{q}, e)$. We choose this parameterization because $q^*, \dot{q}^*, f^*$ can be interpreted as "references". But actually, we could just drop them (absorb them in u_0) without loosing generality. In addition to this, the fast loop respects control limits by clipping $u \leftarrow \texttt{clip}(u, -u_{max}, u_{max}$ element-wise. u_{max} is a constant set in configuration files, not a fluent. `[TODO: Additional mechanisms should also in the fast loop guarantee velocity and joint limits.]`

The parameter vectors k_p^{base} and k_d^{base} are constants set in the PR2 configuration files. They are hand-tuned so that setting $K_p = K_d = I$ leads to acceptable (rather low gain) behavior. The $\cdot$ denotes an element-wise product.

About the integral term: $J_{\text{ft}}^\dagger$ allows us to linearly project the sensor signals to any other space in which we have a target f^* and integrate the error.

$K_p, K_d, K_I, J_{\text{ft}}^\dagger$ are arbitrary matrices; u_0 an arbitrary control bias. Therefore, the **control mode** of the fast loop is determined by the tuple

$$M = (q^*, \dot{q}^*, f^*, u_0, K_p, K_d, K_I, J_{\text{ft}}^\dagger, \gamma) \; . \tag{37}$$

This is the message that the slow loop needs to send to the fast loop – the slow loop can change the control mode at any time.

Inversely, the fast loop passes the message

$$(q, \dot{q}, f, u) \tag{38}$$

to the slow loop, giving it information on the true current state $(q, \dot{q})$, sensor readings f, and computed controls u.

The core question therefore is how the slow loop computes the message M to realize the desired control behaviors. The list of basic desired control behaviors is:

1. Follow a pre-computed trajectory $(q_{0:T}, \tau)$, where τ is the time resolution
2. Follow the position-reference that is online computed by a operational space (or inverse kinematics) controller; the K_p should such that P-gains can be set/added/removed along *endeffector* spaces rather than only uniformly configuration space
3. Establish a contact
4. Stabalize a contact force
5. Limit F/T (to avoid breaking a handle)
6. Sliding (moving tangentially) on a surface (or along a DOF like an opening door) which is perceived via the F/T signal

7.2 Operational Space Control: Computing gains by projecting operational space gains

The appendix B derived the necessary equations in all generality. In practise, it is sufficient to modify the K_p only, using the Jacobian of the desired task space. In equation (85) we have

$$\bar{K}_p = A^{-1} J^\top C K_p J \ , \quad A = H + J^\top C J \ , \tag{39}$$

where we assumed $M = \mathbf{I}$ (quasi-dynamic model) and no other tasks. Further, assuming $C = c$ and $K_p = k$ are scalars we have

$$\bar{K}_p = k(H/c + J^\top J)^{-1} J^\top J \ . \tag{40}$$

I actually tested just $kJ^\top J$.

TODO: Let `FeedbackController` really compute these projected PD behaviors, instead of only $q^*, \dot{q}^*$! Then all of this is automatic!

7.3 Controlling the F/T signal—the *sensed* force

7.3.1 Preliminaries: Understanding force transmission

The following law of force propagation is well known,

$$u = J^\top f \tag{41}$$

where f is a force in the endeffector (e.g., the negative of its gravity load), J the position Jacobian of the endeffector, and u are the torques "perceived" in each of the joints due to the force f. This law is correct only under the assumption that nothing moves. Inversely, this law is typically used to compensate forces: Assume you have a load on an endeffector, gravity pulls it down. The gravity force pulling the load down propagates to torques u in each joint – if you want to compensate this torque the motors need to create the reative torque.

The same also holds for force-torque $f \in \mathbb{R}^6$, where the Jacobian is the stacking of the position and the axial Jacobian.

Typically, f is lower-dimensional than u. So, actually, there should be many u that generate a desired f^*? What is the optimal one? Well, assume $f^* = 0$ for a moment. Then, any choice of u will accelerate the robot (assuming gravity compensation). The only choice to generate $f^* = 0$ and not to accelerate the robot is $u = 0$. Equally, the only choice to generate any f^* without accelerating the robot is $u = J^\top f^*$.

When we include system dynamics in the equation, we have the general

$$u = M\ddot{q} + h + J^\top f \ . \tag{42}$$

where M (the inertia matrix) and h (the coreolis and gravity forces) depend on $(q, \dot{q})$. One way to read this equation is: the torques you "feel" in the joints are the reactive torques of the robot's inertia (that derive the acceleration) plus the torque you feel from the endeff force f.

7.3.2 Controlling the direct F/T signal with a fixated endeff

Consider the following exercise: Fix the endeffector rigidly, e.g. to a table with a clamp (Schraubzwinge). Write a controller that generates any desired f^* in the F/T sensor with the least effort, and stably, and staying close to a desired homing posture.

If we unrealistically assume that our model is correct then the solution simply is (42); for $\ddot{q} = 0$ and a gravity-compensated robot just (41); where

$$J = J_{\text{ft}} \,, \tag{43}$$

which is the position and axial Jacobian of the F/T sensor w.r.t. q.

However, this equation **does not use any F/T sensor feedback** to generate the desired F/T signal. This cannot work well in practise. We can resolve this with an I-controller on the F/T signal error.

$$e = \int_t dt[f^* - f] \tag{44}$$

$$u = J^\top \alpha e \,. \tag{45}$$

The α here has the meaning of an exponential decay of the signal error—which we can show assuming the perfect model. Under perfect model assumption, the F/T sensor measures

$$f = J^\dagger u \,, \quad J^\dagger J^\top \equiv \mathbf{I} \,, \quad J^\dagger = (JJ^\top)^{-1} J \tag{46}$$

$$\text{Note: } J^\dagger u = J^\dagger J^\top f = (JJ^\top)^{-1} JJ^\top f = f \tag{47}$$

$$\tag{48}$$

Note that $JJ^\top$ is a $d \times d$-matrix and invertible and $J^\dagger$ the appropriate left-pseudo-inverse of $J^\top$. Inserting this perfect-model measurement in the control law (44) we get

$$\dot{e} = f^* - J^\dagger u \tag{49}$$

$$\dot{u} = J^\top \alpha (f^* - J^\dagger u) = \alpha J^\top f^* - \alpha \underbrace{J^\top J^\dagger}_{} = \mathbf{I} u = \alpha (J^\top f^* - u) \,. \tag{50}$$

Here, $J^\top J^\dagger = J^\top (JJ^\top)^{-1} J^\top$ is actually the projection that projects any joint torques u into the space that directly relates to endeffector forces and not to accelerations. However, if the u was chosen by some law $J^\top f$, then u will always lie within this projection (will never lead to accelerations of the robot), and therefore it actually is the identity matrix.

Now, the above states that $\dot{u} = \alpha(J^\top f^* - u)$, which says that u exponentially approaches the perfect-model correct torque $J^\top f^*$, which a decay rate α. Therefore, α can be considered a decay rate.

Open: What if we have a $\ddot{q}$ as well? Two possibilities: 1) Reiterate the above reasoning with $\ddot{q}$. 2) Just add the signals.

7.3.3 Control the indirectly sensed contact force of endeff

Exercise: We have the F/T sensor, but attached to it a hand and a contact point with some relative transformation to the F/T sensor. This point is in contact with a table. What we want to control is the force between point and table, which is just a 1D thing.

This is best addressed by thinking of the F/T sensor as if it was a 6D joint (like a ball joint). If we have a force f at some endeffector then we "feel" this force in all joints of the robot as $u = J^\top f$. This includes the F/T sensor joints! So the Jacobian of the endeffector variable (be it 1D or 3D) w.r.t. the sensor pseudo-ball-joint exactly gives the measurement equation. Let's denote this Jacobian as

$$J_{\text{ft}} \in \mathbb{R}^{d \times 6} \,, \tag{51}$$

where $d = 1$ if it is only the distance to the table, or $d = 3$ if it is all forces. Further, let's denote by

$$J \in \mathbb{R}^{d \times n} \tag{52}$$

the Jacobian w.r.t. all the real robot joints.

As above, the *perceived* endeffector force (this time perceived by the F/T sensor) is

$$u_{\text{ft}} = J_{\text{ft}}^{\top} f \quad \Rightarrow \quad f = J_{\text{ft}}^{\dagger} u_{\text{ft}} \,, \tag{53}$$

where $u_{\text{ft}} \in \mathbb{R}^6$ is the F/T signal. Again we may use an I-controller to correct for the error between desired endeffector force f^* and perceived one:

$$\dot{e} = f^* - f = f^* - J_{\text{ft}}^{\dagger} u_{\text{ft}} \tag{54}$$

$$u = J^{\top} \alpha e \,. \tag{55}$$

Note that the last equation generates joint torques proportional to the normal endeffector Jacobian J because e is an error in endeffector force space (not F/T signal space).

This fits to our contoller setup by

$$J_{\text{ft}}^{\dagger} \leftarrow J_{\text{ft}}^{\dagger} \,, \quad f^* \leftarrow f^* \,, \quad \gamma \leftarrow 1 \,, \quad K_I \leftarrow \alpha J^{\top} \,. \tag{56}$$

When force control is turned off, we need to remember to set $\gamma = 0, e = 0$ to ensure that next time it is turned on again it doesn't blow.

Open: What happens for $\gamma < 1$? Is this equivalent to $\alpha < 1$? Perhaps not. ($\sum_{t=0}^{\infty} \gamma^t = \frac{1}{1-\gamma}$)

7.3.4 q-control under force constraints

Assume we have a P(I)D controller on q—typically a PID in some task space that has been projected to joint space. We would like to execute that desired reference behavior but subject to constraints on the sensed endeffector force

$$f_{lo} \leq J_{\text{ft}}^{\dagger} f \leq f_{hi} \,. \tag{57}$$

These are $2d$ constraints.

As with lagrange parameters, we can simply activate the constraints when violated: When one of the components violates the constraint, control the force to be exactly $f_{lo|hi}$. For the latter, use the f-error-integral method as above. This should eventually have higher priority to any other gains (keep other I-gains limited!).

7.3.5 I-gains on position?

7.4 Technical Details and Issues

7.4.1 Ctrl-Message documentation

One message type for setting the control mode AND feedback from the controller.

Setting the control model: $(q^*, \dot{q}^*, f^*, u_0, K_p, K_d, K_f)$. Can be set any time.

Feedback from the controller: $(q, \dot{q}, f, u)$. Published with 1kHz.

7.4.2 Filtering of the differentiation of q

[Peter: please fill in]

7.5 Enforcing control, velocity, joint and force limits

Enforcing control limits is really simple: Just clip the computed u.

Enforcing velocity limits turned out difficult: The velocity signal is so noisy, a direct feedback coupling was bad. Also, the IF-case of velocity-limit-violation turned off and on quickly and introduced even more noise (rattling motors...)

I have ignored limits totally so far – should be handled (as collisions) in the slow loop.

FORCE LIMITS! Not idea how to handle this.

Maybe a route to a more principled approach to all of these: Take the Augmented Lagrangian way to handle constraints as a template: First associate only a soft squared penalty with margin penetration. Then compute/update the respective dual parameters that push you out of the margin.

.1 Reference: General OPEN-LOOP ideal contact force controller

.1.1 General case

This controller is sort-of open loop! It does not take into account any F/T feedback. What is receives as a specification is $\ddot{y}^*$ (desired endeff accel) and λ^* (desired contact force); as well as models of the maps ϕ, J_ϕ, g, J_g. We discuss below how this can properly be made a feedback regulator. We write the problem as a general constraint problem

$$\min_{u,\ddot{q},\lambda} \quad \|u - a\|_H^2 \tag{58}$$

$$\text{s.t.} \quad u = M\ddot{q} + h + J_g^\top \lambda \tag{59}$$

$$J_\phi \ddot{q} = c \tag{60}$$

$$\lambda = \lambda^* \tag{61}$$

$$J_g \ddot{q} = b \tag{62}$$

Notes:

- The role of a becomes clearer when we treat the blue constraint below
- The 2nd constraint relates to an arbitrary task map $\phi : q \mapsto y$ with Jacobian J_ϕ, $c = \ddot{y}^* - \dot{J}_\phi \dot{q}$ and some desired task space acceleration $\ddot{y}^*$.
- We have a set of functions $g : q \to \mathbb{R}^m$ with Jacobian J_g which play the role of inequality constraints
- The 3rd constraint captures desired contact forces with the constraints
- The blue constraint expresses that a) assuming the contact is active there must not be acceleration w.r.t. g or b) if the contact is not active we might want to control the acceleration towards it (to make it active). (This constrains the dynamics. Without this constraint the dynamics could assume that the contact forces λ are generated while the endeff is moving (e.g., by strange external forces). This constraint makes it consistent.)

To derive closed form solutions, each of these equality constraints can be handled in two ways: relax it to become a squared penalty (and then potentially taking the infinite precision limit); or resolve it.

We resolve the 1st and 3rd constraint, and relax the 2nd and 4th to later take the limit. The solution is

$$\mathbf{h} := h + J_g^\top \lambda^* \tag{63}$$

$$f(\ddot{q}) = \|M\ddot{q} - (a - \mathbf{h})\|_H^2 + \|J_g\ddot{q} - b\|_B^2 + \|J_\phi\ddot{q} - c\|_C^2 \tag{64}$$

$$\ddot{q}^* = (M^\top H M + J_g^\top B J_g + J_\phi^\top C J_\phi)^{-1}[M^\top H(a - \mathbf{h}) + J_g^\top B b + J_\phi^\top C c] \tag{65}$$

The limit $B \to \infty$:

$$\ddot{q}^* = (X + J_g^\top B J_g)^{-1}[J_g^\top B b + x] \tag{66}$$
$$= J_{g\,XB}^{\#} b + (\mathbf{I} - J_{g\,XB}^{\#} J_g)\,(X^{-1}x) \tag{67}$$

And note that $(X^{-1}x)$ is the solution to only having the other terms. Given $\ddot{q}^*$, the optimal control is computed as $u = M\ddot{q} + h + J_g^\top \lambda^*$. We still did not take the limit of the C-term (endeffector position control). We could using the hierarchical limit case.

.2 Reference: Pullback of operational space linear controllers

The above assumes that at any instance in time we want a certain task-space acceleration $\ddot{y}^*$ and translates this to an optimal joint control in that instant in time. If we want to implement a certain feedback behavior in the task space, that is, we have a desired feedback control law $\pi:\ y, \dot{y} \mapsto \ddot{y}$, we can evaluate π at every point in time and project to operational space control.

$$\ddot{y} = \ddot{\phi}(q) = \ddot{(}Jq) = \dot{(}\dot{J}q + J\dot{q}) = 2\dot{J}\dot{q} + J\ddot{q} \tag{68}$$
$$\ddot{y}^* = K_p y + K_d \dot{y} + k \tag{69}$$
$$J\ddot{q} \stackrel{!}{=} c = \ddot{y}^* - 2\dot{J}\dot{q} = K_p y + K_d \dot{y} + k - 2\dot{J}\dot{q} \tag{70}$$
$$\approx K_p(J(q - q_0) + \phi(q_0)) + K_d J\dot{q} + k \tag{71}$$
$$= K_p J q + K_d J\dot{q} + k' \,, \quad k' = k + K_p(\phi(q_0) - Jq_0) \tag{72}$$
$$\ddot{q}^* = A^{-1}[... + J^\top C c] = A^{-1}[... + J^\top C(K_p J q + K_d J\dot{q} + k')] \tag{73}$$
$$= A^{-1}[...] + A^{-1}J^\top C K_p J q + A^{-1}J^\top C K_d J\dot{q} + A^{-1}J^\top C k' \tag{74}$$
$$= \bar{K}_p q + \bar{K}_d \dot{q} + \bar{k} \,, \quad \bar{k} = A^{-1}[...] + A^{-1}J^\top C k' \,, \quad \bar{K}_p = A^{-1}J^\top C K_p J \,, \quad \bar{K}_d = A^{-1}J^\top C K_d J \tag{75}$$

.3 How to make this FEEDBACK?

W.r.t. y (endeff pos) it is clear how to make this feedback: We can impose a PD behavior on the endeffector

$$\ddot{y}^* = k_p(y^* - y) + k_d(\dot{y}^* - \dot{y})$$

and send this desired endeff accel to the general controller.

What about λ^*??

.4 What do we want?

desired task space acceleration law

$$\ddot{y}^* = \ddot{y}^{\text{ref}} + K_p(y^{\text{ref}} - y) + K_d(\dot{y}^{\text{ref}} - \dot{y}) + K_{Ip}\int (y^{\text{ref}} - y) + K_{Id}\int (\dot{y}^{\text{ref}} - \dot{y}) \tag{76}$$

That defines a desired *acceleration*. But if the system was precise in enforcing this acceleration it would be non-compliant. Note: strictly speaking, if this law says $\ddot{y}^* = 0$ (e.g., because K_p and K_d are zero), then this is a strict (non-compliant) statement.

precision/compliance Given a desried $\ddot{y}^*$, the precision along some dimensions may not that important. We may capture this with the precision (or compliance) matrix C. As a convention, let the eig$(C) \in [0, 1]$, and an eigenvalue of 1 states full precision, while an eigen value of 0 states full compliance.

This implies an objective term

$$\|J_\phi \ddot{q} - \ddot{y}^*\|_C^2$$

Null cost reference Typically one defines control costs $\|u\|_H^2 = \|M\ddot{q} + F\|_H^2$. However, this becomes semantically tricky. When defining what is 'desired' I propose to stay on the level of accelerations. So we have a desired ('null') acceleration law

$$\ddot{q}_0^* = \ddot{q}_0^{\text{ref}} + K_p^q(q_0^{\text{ref}} - q) + K_d^q(\dot{q}_0^{\text{ref}} - \dot{q}) \tag{77}$$

and consider costs

$$\|\ddot{q} - \ddot{q}_0^*\|_H^2 \ .$$

Note that, by appropriate choices of parameters, the typical control cost can be mimicked. However, the semantics is somewhat different. For instance, setting K_d^q and $\dot{q}_0^{\text{ref}}$ implies that we want to damp motion, and choosing controls u that implement this daming are at *null costs*. Equally, for non-zero F and $\ddot{q}_0^* = 0$, applying controls that ensure zero acceleration are at *null costs*. That's we I call it *null cost reference*. This is rather different to generally penalize $\|u\|_H^2$, which would imply costs for any controls u, even if they just implement counteracting F of generating the null reference.

Optimal acceleration law We compute the optimal acceperation $\ddot{q}^*$ in its 1st order Taylor approximation w.r.t. q and $\dot{q}$:

$$\begin{aligned}
\ddot{y}^* &= \ddot{y}^{\text{ref}} + K_p(y^{\text{ref}} - y) + K_d(\dot{y}^{\text{ref}} - \dot{y}) && (78)\\
&\approx \ddot{y}^{\text{ref}} + K_p y^{\text{ref}} - K_p(J(q - q_0) + y_0) + K_d\dot{y}^{\text{ref}} - K_d J\dot{q} && (79)\\
&= k - K_p Jq - K_d J\dot{q} \ , \quad k = \ddot{y}^{\text{ref}} + K_p y^{\text{ref}} + K_d\dot{y}^{\text{ref}} + K_p(Jq_0 - y_0) && (80)\\
\ddot{q}_0^* &= k^q - K_p^q q - K_d^q\dot{q} \ , \quad k^q = \ddot{q}_0^{\text{ref}} + K_p^q q_0^{\text{ref}} + K_d^q\dot{q}_0^{\text{ref}} && (81)\\
\ddot{q}^* &= \underset{\ddot{q}}{\text{argmin}} \|\ddot{q} - \ddot{q}_0^*\|_H^2 + \|J\ddot{q} - \ddot{y}^*\|_C^2 && (82)\\
&= (H + J^\top CJ)^{-1}[H\ddot{q}_0^* + J^\top C\ddot{y}^*] && (83)\\
&\approx \bar{k} - \bar{K}_p q - \bar{K}_d\dot{q} && (84)\\
&= \bar{K}_p(q_{\text{ref}} - q) - \bar{K}_d\dot{q} \ , \quad q_{\text{ref}} = \bar{K}_p^{-1}\bar{k} && (85)\\
\bar{K}_p &= (H + J^\top CJ)^{-1}[HK_p^q + J^\top CK_p J] && (86)\\
\bar{K}_d &= (H + J^\top CJ)^{-1}[HK_d^q + J^\top CK_d J] && (87)\\
\bar{k} &= (H + J^\top CJ)^{-1}[Hk^q + J^\top Ck] && (88)
\end{aligned}$$

Transfer to controls Given an optimal acceleration law and the system dynamics, we choose:

$$\begin{aligned}
\ddot{q}^* &= \bar{K}_p q + \bar{K}_d\dot{q} + \bar{k} && (89)\\
u^* &= M\ddot{q}^* + F && (90)\\
&= M\bar{K}_p q + M\bar{K}_d\dot{q} + M\bar{k} + F && (91)
\end{aligned}$$

Note again, only here, the system dynamics enter. The specification of the optimal acceleration law is independent of the dynamics. (Unless H and $\ddot{q}^{\text{ref}}$ are chosen in relation to M and F to mimick typical control costs—but we explicitly avoid this to make desired system behavior somewhat less dependent on (possibly inaccurate) dynamics models).

Error correction of system dynamics The optimal acceleration law gives an explicit desired acceleration. We may estimate the control error by a low pass filter on the acceleration errors. There are two options:

Let $\langle \ddot{q}^* \rangle$ be a low pass filter of the desired accellerations $\ddot{q}^*(q, \dot{q})$; and $\langle \ddot{q} \rangle$ a low pass of the actual true accelerations. We may define $g = \langle \ddot{q} \rangle - \langle \ddot{q}^* \rangle$ and control

$$u = M(\ddot{q}^* + g) + F \ .$$

Or we may assume system dynamics

$$u = M\ddot{q} + F + g \tag{92}$$

for some unknown and variable (slowly changing) $g \in \mathbb{R}^n$ which reflects constant loads on the joints. We may estimate g as a low-pass filter,

$$\begin{aligned} g &= \langle u - M\ddot{q} - F \rangle_{\text{low pass}} = \langle u \rangle - M \langle \ddot{q} \rangle - F && (93)\\ &= \langle M\ddot{q}^* + F + g \rangle - M \langle \ddot{q} \rangle - F && (94)\\ &= M \left[\langle \ddot{q}^* + g \rangle - \langle \ddot{q} \rangle \right] && (95) \end{aligned}$$

which is puzzlingly different to the above. (M is obvious, but the rest?)

Compliant error correction of system dynamics The above describes a scheme that corrects any errors from the desired acceleration. However, in the case of contact, and desired compliance, we do not *want* to enforce exact reference following along certain dimensions. E.g., in the case of a contact we systematically do not accelerate towards, leading to a systematic error in the system equations, an adaptation of g, and perhaps divergence.

Error correction on task space level Let $\langle \ddot{y}^* \rangle$ be a low pass filter of the desired task space accellerations and $\langle \ddot{y} \rangle$ a low pass of the actual true accelerations. We may define $g = \langle \ddot{y} \rangle - \langle \ddot{y}^* \rangle$ and add g to the $\ddot{y}^{\text{ref}}$.

Alternatively, we may define g as integral error in the task space and add it to $\ddot{y}^{\text{ref}}$.

In both cases, g adds to k, showing that it adds to $\ddot{q}^*$ as $(H + J^\top C J)^{-1} J^\top C g$.

The matrix C controls compliance.

Measured-force limits

Limit Energy

A Roopi – Abstractions for scripting concurrent processes; Robot Operating Interface

How to program a robot? Possible answers are SMACH, Urbi, or the new FlexBE. A core question though is how eventually you want to represent behaviors. As (hierarchical) finite state machines? Control flow graphs? Or directly as code in some programming language?

My conceptual background in this respect is given in the "'Relational Activity Processes"' paper (ICRA'16), which describes a formal representation of concurrent activity processes. I still think this is a good representation, especially also for learning.

Initially I thought that behaviors should/could strictly be represented as relational machines, strictly following the formal description; also when describing the control flow via relational rules.[2] We then had a test phase in our lab where we enforced a strict interface to a relational activity process via python, allowing online python scripting to control the process.

I see it a bit different now. Here is the new view: We want to be able to use the full power of a standard programming language (C++, Python, Octave) to code behaviors, rather than on some domain-specific language. By 'full power' I mean the logic part (if-then-else), the natural hierarchies and refactoring of code, but especially also the direct access to all data to do computations and make decisions.

The following descripes C++ conventions/data structures on scripting a concurrent process. The behavior script is then directly in C++ (or python). The result is a much thinner infrastructure between the abstractions (concurrent activities and their control flow) and the lower levels (control loops, perception pipelines, etc).

However, the resulting behavior actually still adheres to the RAP formalism: it still is a concurrent activity process. Therefore, the long term vision is that we can start doing imitation learning from such coded behaviors: We can learn the typical statistics of when which activity (with which parameters) is triggered. That moves from the original programming code to a fully statistical description of behavior – potentially opening ways to optimize the behavior, that is, let the system explore control flows and activity parameters autonomously, essentially learning to script behavior.

A.1 Scripting Concurrent Processes

- At the core are `Signaler`s, which are objects that broadcast (via condition variables) to or call callbacks of activities. There are two types of Signalers in Roopi: Variables (see threading section) and activities. Every signaler has an integer **status**. In the case of variables, it is the variable's revision. In the case of activities it is some symbol, such as 'true', 'false', 'convered', 'running', etc.

- An activity can serve as an event: whenever its status becomes 'true', something is triggered. E.g., using

  ```
  Act::Ptr at(const Act::Ptr& event, const std::function<int ()>& script);
  ```

  you can state that, when the event becomes true, a certain piece of code (e.g., passed as a lambda expression) should be executed. Or

  ```
  Act::Ptr whenever(const Act::Ptr& event, const std::function<int()>& script);
  ```

  which says that the script should always be triggered (stepped within a thread) when the event becomes true.

  Further, within a script (or the main) it is natural to wait for events, e.g. using

  ```
  bool wait(const Act::Ptr& event, double timeout=-1.);
  ```

  The 'wait', 'at' and 'whenever' are very similar to URBI and provide the basic control flow mechanism to script concurrent activities. All scripts are run in their own thread; the 'at' and 'whenever' methods of Roopi do not pause the caller. The two methods

  ```
  Act::Ptr run(const std::function<int()>& script);

  Act::Ptr loop(double beatIntervalSec, const std::function<int()>& script);
  ```

[2] By "relational machine" I mean a knowledge base that controls a relational activity process, mostly by having (non-decision) rules.

allow one script to call other scripts in threads without pausing. The return values `Act::Ptr` of the above methods returns the activity that is wrapping the running script – waiting for a status change and reponding to it allows one to react to their outcomes.

- An event activity usually subscribes callbacks with other Signalers, e.g. other activities to monitor their status, or variables to monitor their revision. Using

  ```
  Act::Ptr event(SignalerL sigs, const EventFunction& eventFct);
  ```

  allows the user to define events using his own EventFunction, which is called whenever one of the signalers changes status, and depending on that (or the data content of variables) can change the event status.

- To give an example: The standard `wait(SignalerL sigs)` waits for all signalers to have a status greater zero. What happen is: it first creates an event activity which subscribes a callback to all signalers. Whenever one of the signalers changes status, the callback checks if all statuses are greater zero; if yes, the callback sets the event's status to 'true'. This triggers a broadcast which wakes the caller of `wait`.

That the basic control flow concepts, independent of robotics. Using this we can code hierarchical concurrent processes, where typical activities are just pieces of code that can read/write to variables, set the own status depending on computations, or use the above methods to control/create/destropy other activities (Actor Model in Computer Science).

Conceptually, the set of activities and their statuses corresponds to the relational state in RAP. The policy is not represented as a state machine, but rather as the set of activities themselves, which change statuses or create/destroy activities in a decentralized manner. This is closer to the decentralized rule sets in RAP than finite state machine flow in SMACH. In practise, we often start scripting behaviors in a single serial main script. Then we gradually refactor and encapsulate such scripts to become activities and run them in parallel reactively.

Beyond that, Roopi provides many standard activities, especially control activities that activate a task within a motion controller.

B Threading

MLR provides very simple threading and communication/stepping of threads via shared variables. Conceptually this originated thinking about factor graphs, where factors (threads) do computations by updating (beliefs over) variables. They communicate only via variables. The whole think is a bi-partite graph. In the code, threads access variables via the 'Access' class – the 'VariableData' class itself is actually never touched and not really part of the interface.

Thread: You derive from 'Thread' and:

- You need to implement the open (startup something, e.g., open a window, start a driver, socket, whatever), step (do the computation by accessing the variables and reading from and writing into them), close. All three (also open and close) are executed within the thread.

- The constructor needs to call the constructor of Thread, especially setting the beatIntervalSec. If that is 0, the thread will loop (=iterate stepping with full speed and no idle time). If beatIntervalSec>0, the thread will loop with a certain frequency, autonomously calling 'step' every beatIntervalSec seconds. If beatIntervalSec<0, the

```
Roopi R;

act a = R.startTaskController();
act b = R.startCommunicationWithBaxter();

act liftHand = R.ControlTask(TaskMap(...), target, naturalGains);

//busy wait
for(;;){
  if(liftHand==R_done) break; //-> when status of liftHand is R_done
}

//idle wait
for(;;) if(wait(liftHand)==R_done) break;
//equivalently
R.wait(liftHand, R_done);

//idle wait for multiple activities conjunctive
R.waitAnd({liftHand, gazeBall});
//or
R.waitAnd({liftHand, gazeBall}, {R_done, R_done});

//idle wait for multiple activities disjunctive
R.waitOr({liftHand, gazeBall}, {R_done, R_done});

//beep when the grasp is done
act beep = R.at( {grasp}, R_equals, {R_done}, [](){
    R.beep();
    return R_true;
  });

//whenever variable q_ref has size larger 10 and its revision is >10, beep
act beep2 = R.whenever<arr>("q_ref", [](Access<T>& x){ //lambda expression to define an event
    return x->N>10 && x.getRevision()>10;
  }, [](){ //lambda expression to define an effect
    R.beep();
    return R_true;
  });
```

Table 1: 'Top down wish code' that almost reflects the actually implemented Roopi

thread will 'listen' – that is, it will step whenever one of its variables (that it listens to!) gets a new revision (is been written into by another thread). In the latter case, don't forget to set the 'listensTo' flag in the Access constructor – otherwise the thread won't listen to anything and not step.

Further, the constructor of the derived class should call 'threadOpen' (for or listening threads) or 'threadLoop' (for looping/beating threads). The destructor should call 'threadClose'.

- An 'Access' (to access a global shared memory variable) could be created any time – but standard convention is to have them as members of the thread. Make sure that their constructor is properly called (passing the thread pointer, variable name, and listening flag) within the thread's constructor.

Variable: The 'VariableData' or 'VariableBase' classes are never touched directly by a user. Instead, everything is via an Access. If you create an Access it will look up a global registry on whether such a variable (of given type and name) already exists; if not it'll create it, if yes it'll refer to it. Given an Access `A`, you can access its contents either with `A.readAccess(); A().anyMethodOfData...; A.deAccess();` or `A.get()->anyMethodOfData...`. The latter is the the 'Token' pattern, which is used throughout the code at some places: The get() method returns a little token which holds the mutex while in scope. The same for writeAccess and set.

The Variable data is a revisioned RW-locked container. Revisioned means that it counts the write accesses. The Access class provides methods to wait for a specific or next revision of the variable, or access the variable's revision.

Accesses are usually members of a thread and can be viewd as an edge between variable and thread. (Logging allows to track exactly which thread wrote/read which revision of which variable at which time.) But actually you can create an Access anywhere in the code (passing a NULL pointer as thread). That is, instead of writing `int x;` anywhere in the code, you can write `Access<int> x(NULL, "x");` anywhere in the code, which makes x a globally registered, revisioned, RW-locked integer.