The MLR Guide

Marc Toussaint

January 9, 2012

Sources for this document: mlr/share/doc/guide.tex   *Please contribute!*

Contents

1 Guide to Things

This is the central hub for information. Goals of this document:

- Simply saying "what is there" and who could be asked to get further information.

- The page should directly link to technical infos, be searchable, and be a kind of guide to beginners on where to start reading and find information.

1.1 Guide to MLR coding

- Set up **accounts** on achtauge and redmine (https://maserati.mi.fu-berlin.de/redmine/projects/mlr)

- The default directory layout:

  - — /git/¡dir¿— for your git repository (e.g., — /git/mlr— or — /git/one—)
  - — /lib— for your external lib installations

- Install the following **Ubuntu packages**:
  ```
  liblapack-dev
  freeglut3-dev
  libqhull-dev
  libf2c2-dev
  libann-dev
  libplib-dev
  gnuplot
  meld
  libdc1394-22-dev
  ```

- Download the **tar-Ball** with all other required **external libs** (don't extract for now!)

  —mlrLibs.tar.gz—, in Wiki (Maserati), Main, MLR, Documents

  The external libs are:
  ```
  esdcan-usb331-linux-2.6.x-x86-3.8.3
  esdcan-usb331-linux-2.6.x-x86_64-3.8.3
  ntcan
  ode-0.11
  opencv-2.1
  revel-1.1.0
  schunkLWA
  SWIFT++_1.2
  tony-mdp
  urg-0.0.2
  xvidcore-1.1.3
  ```
  Concerning opencv-2.1: Some people in the group already use opencv-2.3. When using Ubuntu, packages can be obtained from —http://packages.ros.org/ros/ubuntu/—. For convenience, the gpg-key is in the root folder.

- Optional: set up your SSH-key in Maserati via Wiki; described in detail in Sec. 2.1

- Install git :-)

- Create your **git repository** clone
  ```
  cd ~/git
  git clone https://<user>@maserati.mi.fu-berlin.de/git/mlr <dir>
  OR
  git clone git@maserati.mi.fu-berlin.de:mlr <dir>
  ```
  This puts a clone in the directory —¡dir¿—. See section 2.1 on how to use git.

- **Run INSTALL script** in the repository root folder. You need to provide the tarball with the external libs.
  ```
  ./INSTALL
  or
  FILE_TARBALL=mlrLibs.tar.gz ./INSTALL
  ```
  This script does several things for you:

  - It makes **external** libs in a lib-directory and builds symbolic links to them in —git/share/extern—.
  - Builds a **make-config** file. Here, you can specify your individual compiler options (e.g., in-/excluding libs).
  - Makes all **MLR-libs** in —share/lib/—. These are:
    ```
    libmt
    libnj
    libsd
    libnp
    libdz
    librelational
    ```
    If problems occur, try to comment the following lines in the —src/MT/Makefile—:
    ```
    #FREEGLUT = 1
    #ANN = 1
    #LAPACK = 1
    ```
    This reduces functionality of the code, but should at least compile.
  - Makes the **tests** in —share/test/—
  - Try —test/array/x.exe—

1.1.1   Guide to the Makefile system

- —share/make-generic—: central Makefile; should be included by all other Makefiles; provides machinery for building executables, libs etc.; checks for plenty of flags and includes external and MLR libs as appropriate

- —Makefile—: found everywhere (in local directories for building libs, executables in tests etc.); sets local compiler options

- —share/make-config—: included by —make-generic—; allows to set individual compiler options for all projects

1.2   Guide to the robot hardware (Marc & Stanio)

- Startup hardware

  - turn on robot
  - mlrMountHardware
  - mlrJoystick

- mlrJoystick -openArm 1 -openHand 1 -openSkin 1

- Debugging hardware:

  - 09-testSchunkBasics
  - 09-testHandMotion
  - 09-testHandSense

- Testing perception:

  - 10-testEarlyVision
  - 10-testPerception

- Learning about the control architecture:

  - 10-miniExample (launches minimal set of Processes explicitly by hand)
  - RobotActionInterface (launches Processes using the RobotActionInterface)
  - 10-planningDemo

1.3 Guide to BiROS (parallel processing architecture) (Marc)

1.3.1 Where to start learning

1.3.2 Status: the basic mechanisms in process.h are robust and reliable.

1.4 Guide to ORS (robot simulation data structures) (Marc)

There are numerous robot simulation environments out there. Why introduce a new one? Well, actually the aim of ORS is not to develop a new full robot simulation environment. Instead, it aims to provide basic data structures that help to link to existing external libraries and algorithms together. In that way we can flexibly provide a simulation environment with different features enabled by linking to external libraries – but always with the same basic interface as provided by the ORS data structures.

The data structures are really simple and basic (see ors.h): Vector, Quaternion, Frame, Joint, Body, kinematic Graph – that's basically it.

Beyond the data structures, ORS also implements two functional things:

- The jacobian (but that's trivial, given the kinematic graph)

- So-called TaskVariables: These are very useful for designing motion control contraints (more precisely: cost terms for motion rate control or stochastic optimal control). The typical 'endeffector position' is only one type of task variable – I've implemented many more (inclluding their Jacobians) which capture relative positioning, orientations, alignments, angles, between arbitrary frames of the kinematic graph; capturing collision costs, joint limit avoidance costs, etc etc. In practice, playing with task variables is often the core in designing elegant motions.

  ORS can currently link to the following external libraries:

- SWIFT++ for collision (and proximity) detection. (Note: there is hardly any alternative out there to do proper proximity(!) detection. Bullet, ODE, SOLID, etc don't)

  (INSERT LIST OF ALTERNATIVE LIBS)

- ODE for simple dynamic simulation including collision behavior (SWIFT++ only detects collisions, but doesn't simulate their effects)

- OpenGL for display

- blender (this is a bit outdated – but I once wrote a python script to export/import blender geometries to ORS data structures)

Perhaps on the wishlist for ORS communication:

- OpenRAVE / OpenGRASP

- IBDS (perhaps much better dynamic simulation than ODE)

- daVinci, bullet, OpenTissue

- Yuval Tassas CapSim (new methods for EXACT contact simulation)

1.4.1 Where to start learning about ORS

- Robot lecture: the basic lecture on 3D geometry (to learn about my conventions) and perhaps the lecture which introduces task variables (and their Jacobians)

- Have a look at ALL ors_* test projects in the git repository – they're really basic. Understand them all. The —ors_editor— is useful to edit the ors-file describing geometries. Play with the —test.ors— file while watching the effect in the window.

- The ors doxygen.

1.5 Guide to RobotActionInterface

1.5.1 What is the RobotActionInterface?

The RobotActionInterface (RAI) provides high level commands which can directly be executed on the real robot. One example is "pickObject(o)", which you can call from a main.cpp file. This section will show you all the neccessary details of the Interface in case you want to understand the underlying mechanisms.

1.5.2 Pointer to implementation (Pimpl) paradigm or idiom

Instead of writing every internal variables/function into the class header, we use a second class (the private implemenation), which provides all non-public details. This has several advantages. First, the headers will be smaller, so the user will only see the relevant data. Second, we do not need to include or forward classes in the primary one, which makes compilation faster. And third, if we change internal variables in the private class, the primary header stays the same. All files, which include this header, do not have to be recompiled either.

see also 6 and

http://c2.com/cgi/wiki?PimplIdiom
http://en.wikipedia.org/wiki/Opaque_pointer

1.5.3 General structure

In principle, the interface initializes all neccessary processes, connects the inputs and outputs together, and finally starts every process as a seperate thread. This is provided by the "open()" command. It will automatically load all the standard processes like the vision, planner and controller. You can also specify processes manually via the MT.cfg config file.

1.5.4 Execution Motions

The core of the interface is the controller, which is responsible for executing trajectories on the robot, stopping the motors or moving the hand. This is done by setting one or more "TaskVariables" (TV) (Implemented in ors.h). When executing a task, the controller considers all active TV's, and calculates the optimal control (see also SOC/AICO). A "Task" is defined as a set of TV's, where each TV can have a different weight. For example, if you want to go to a specific position, you could define that there is a TV for reaching this point, a TV for collisions, a TV to avoid motor limits and a TV to follow the shortest path. We have already defined several releant tasks. (see robot.h/.cpp for implementation details) The list below shows our set of tasks and what the should accomplish.

- BasicRobotTask(DoNothing): Deactivate all TV's

- BasicRobotTask(Stop): Set TV's, so that all motors stop immediately

- BasicRobotTask(Homing): Go to a predefined home position

- BasicRobotTask(OpenHand): Do not move any motors, but the finger.

- BasicRobotTask(CloseHand): Close hand, till we have a certain pressure on the skin. Otherwise till the limit.

- BasicRobotTask(Reach): Move endeffector to predefined reach point

- BasicRobotTask(FollowTrajectory): Follow a trajectory from the planner process. Thereby taking care of limits and collisions

- BasicRobotTask(Joystick): Activate joystick control. Each button gives rise to a different set of TV's.

1.6 Guide to SOC & AICO (Stochastic Optimal Control and motion planning routines) (Marc)

1.7 Guide to the Array class (the data structure most used throughout for vectors, matricies, tensors, lists, graphs, etc)

This is a standard Array lib – just like many others. (It actually evolved from a lib started in Bochum, 15 years ago...) The main design goals are:

- compatibility with typical mathematical notation when coding equations (That's in fact most important: It is curcial for us to be able to code as directly as possible equations we've derived on paper!)

- compatibility with Matlab/Octave conventions (mostly...).

- full transparacy and minimalism (for easier debugging),

- Many features: unlimited rank tensors, fast memory handling, links (by default) to LAPACK (@@is just as fast as any other optimized matrix library), lists – and everything implemented by only a SINGLE basic class, not a whole lib of classes.

1.7.1 Tip: Debugging with arrays

If —*A.p@A.N—

1.7.2 Important conventions

- In all my code, a "list" is an array of pointers. Why? Years back I've implemented double linked lists (O(1) complexity for insertion/deletion) as was typically taught (back then) – but in practise that's total nonsense. Insertion/deletion in a pointer array is also O(1). And array of point is so so much simpler and easier to debug.

  Convention: A "ThingList" is a —typedef MT::Array¡Thing*¿— !

- Equally for graphs: A "graph" is a list of edges and a list of nodes! (What else?!) I've implemented basic graph template routines. They assume that the node type contains a list of in-edges, and a list of out-edges, and the edge type contains a pointer to the from-node and a pointer to the to-node.

1.7.3 Where to start learning

- Directly start looking at the 'array' test in the git repository. Just by reading though all examples I think one learns most.

- The array doxygen.

- The class has more features (functions) than one might think: I've seem many students reimplementing routines that are actually already there – frankly I don't know how to avoid that.

1.7.4 Future changes:

- Instead of having explisit listX routines (for insertion, deletion, sort, etc) there will be a 'listMode' flag in the array class, indicating that this array is to be treated as a list of pointers.

1.8 Guide to the visual perception software

The current perception module does a very simple thing:

1. We have a left and right image. 'EarlyVision' is computing the HSV for these images. We assume to know a specific target $hsv^* \in [0, 255]^3$ values together standard deviations $\sigma_{hsv} \in [0, 255]^3$. We compute the evidence $\theta_i = \exp(-(hsv_i - hsv^*)^2/\sigma_{hsv}^2)$ [sorry for sloppy notation] for each pixel i in the left and right image.

2. Given θ_i in an image, we call OpenCV's flood-fill that finds the contour of the highest θ-value region. Let's call the contour ∂C. (We do this for both images.)

3. Given the contour ∂C we compute a distance-to-contour field/image: for each pixel i we compute $d_i = \min_{j \in \partial C} \|i - j\|$ (using some OpenCV routine). We do this for both images. The image d_i is a good potential cost function to let 2D contour models converge to the HSV contour.

4. We have three different parametric 2D contour models: 1) for a circle (1 parameter), 2) for a polygon with 6 vertices and parallel opposing edges (a bit like a hexagon, can fit any 2D-projected 3D box), 3) a contour model that corresponds to a 2D projected cylinder.

   We fit a contour model to the HSV contour by minimizing the sum of d_i for all points on the contour model. We do this on both images. Fitting is done by gradient descent (RPROP). We get parameters of the 2D contours with sub-pixel accuracy.

5. Given the fitted 2D contours in the right and left image, we triangulate them. Giving us a 2D contour mapped into 3D space. From there it is trivial to fit a 3D ball, cylinder, or box.

Note: It is not by accident that we stay 2D for until the last step: in our experience it proved more robust to try to fit shapes/contours in 2D first with as much accuracy as possible before triangulating.

1.9 Guide to further existing software

1.9.1 Optimization methods (Marc)

There are quite a bit of generic optimization methods implemented – but not well documented/organized yet:

- Rprop (best gradient descent method)

- GaussNewton

- CMA

- Some Genetic & Evolutionary Algorithms (e.g., similar to CMA)

  Ask Marc.

1.9.2 Inference and Machine Learning methods (Marc)

We have probabilistic inference code (infer lib) and also basic Machine Learning methods (as introduced in the ML lecture). Ask Marc.

- Gaussian Processes

- ridge regression, logistic regression, etc

- MDPs, POMDPs

1.9.3 Computer Vision methods (Marc)

LINK to my notes on the CV system

1.9.4 Robotics algorithms (Nikolay & Marc)

- RRTs

- Trajectory Prediction

- etc

1.9.5 Relational RL (Tobias)

- Robot Manipulation Simulator: http://userpage.fu-berlin.de/tlang/RMSim/

- libPRADA, a library for relational planning and rule learning: http://userpage.fu-berlin.de/tlang/prada/

1.9.6 coding utilities

- String class

- Parameters read from cmd line or config file

- plotting in opengl & gnuplot

- opengl using freeglut, fltk or qt

2 How To do things

2.1 git

Check the **man pages** of git and git subcommands. Usually, —git foo help— is synonym for —man git foo—.

As a preliminary to using git, please take an hour to read the two **tutorials** on git provided in the man pages. In your terminal write —man gittutorial—. When done with it read on with —man gittutorial-2—. Depending on your background and experience with other revision control systems there might be better approaches to introduce git, but at any rate, these two tutorials give basic insight into git's concepts.

For deeper understanding —man gitcore-tutorial— is your friend.

2.1.1 Getting git started

- install git :-)

- Optional: define the ssh key of your local machine in maserati (more details below) (only for convenience)

- Get git clone

```
cd git  // go to git directory
git clone https://<user>@maserati.mi.fu-berlin.de/git/mlr <dir>
OR
git clone git@maserati.mi.fu-berlin.de:mlr <DIR>
// where <DIR> is your local target root directory for the GIT-clone
```

- that's already it!

2.1.2 Setting up maserati as git server

Most steps are only required if your local git used a different git server before.
First type

```
git remote -v
```

to see which git server you use. In case it is not maserati you should change this!

- in your web browser go to

  ```
  https://maserati.mi.fu-berlin.de/redmine/
  ```

  and login with your account on maserati

- Browse to *My account* (up right), *Public keys* (almost up right) and click *new value*.
  Create a sshkey in the webpage, e.g. by copying some ssh file from your computer, here we use gedit for example:

  ```
  cd .ssh    // in home-folder of your local machine
  gedit id_rsa.pub
  ```

  copy everything for key value and activate it on the server

- type in your computer to add maserati as server:

  ```
  git remote add maserati git@maserati.mi.fu-berlin.de:mlr
  ```

- go to your main git folder and type

  ```
  gedit .git/config
  ```

  then change all your remotes to maserati, remove any old/conflicting origins entries, it should look like this:

  ```
  [branch "master"]
    remote = maserati
    merge = refs/heads/master
  [remote "maserati"]
    url = git@maserati.mi.fu-berlin.de:mlr
    fetch = +refs/heads/*:refs/remotes/maserati/*
  ```

2.1.3 git commands

Quick list:
```
git config --global push.default tracking
git config --global diff.external git-diff.sh
// with a git-diff.sh file (in your local bin)
// #!/bin/bash
// /usr/bin/meld "$2" "$5" > /dev/null 2>&1

git status
git diff
git add|mv|rm|... <files>
git commit -am 'message'  (or --all --message 'message')
git pull --all //DON'T do this! fetch --all is ok, but pull might
               //merge EVERYTHING in the current working branch
git push --all

git branch -a
// overview over all branches and configs for push and pull
git remote show maserati
git checkout <branch>
git branch <new-branch>
git branch --track <new-branch> <remote-branch>
// set branch to pull from some remote branch
git branch --set-upstream <existing-branch> <remote-branch-to-be-tracked>
// Example: git branch --set-upstream branch123 origin/branch123
git merge <other-branch>

.gitignore
git update-index --assume-unchanged <files>
```
Some other contribution to the git cheat sheet:

- init

  - —git init— just inits the —$PWD—, nothing added, nothing

  - create central repo for current development

    1. at home, —cd /dev/mpcdzen—, —git init—

    2. —git add mpcdzen.sh mpcdzen.conf—

    3. —git commit -m initial—

    4. —ssh my.server.there— go to where my 'central' repos are.

    5. —cd /gitrepos/; git clone –bare ssh://my.home.here/home/me/dev/mpcdzen—

    6. everywhere: —git clone ssh://my.home.here/home/me/dev/mpcdzen—

- when cloning on the same machine: —git clone –bare –no-hardlinks -l /dev/mpcdzen— can be helpful

- commit

    - —git add -u ../path/— : add all *tracked* files under path

    - —git add ../path/— : add *all* files under path to staging area

    - —git commit -a — : add all modified tracked files from the *whole working copy* and commit

- diff

    - —git diff— : how current files and HEAD differ

    - —git diff –cached — : diff staged files against HEAD

- status

    - —git status— : list staged, modified tracked, untracked files

    - if —file— is not tracked you can use .gitignore or .git/info/exclude to tell git not to take care of it. If it is already tracked, this is the way to say the same: —git update-index –assume-unchanged ../file—

- remote

    - —git add remote mhmgr ssh://my.homemachine/git/repo— kinda introduces my home machine's repo to this one. Then, —git fetch mhmgr— and I'm set to go for comparing, merging etc.

- refs' naming

    - —HEAD‸— addresses the parent of —HEAD—, e.g., assuming I've merged 2 branches and committed 3 times on the current branch, —git show HEAD2̂— shows the second branch before merging. In other words, direct after merge and commit, the parents of HEAD are —HEAD1̂— and —HEAD2̂—

    illustration from man git-rev-parse:

```
        G   H   I   J
         \ /     \ /
          D   E   F
           \  |  / \
            \ | /   |
             \|/    |
              B     C
               \   /
                \ /
                 A

        A =      = A^0
        B = A^   = A^1     = A~1
        C = A^2  = A^2
        D = A^^  = A^1^1   = A~2
        E = B^2  = A^^2
        F = B^3  = A^^3
        G = A^^^ = A^1^1^1 = A~3
        H = D^2  = B^^2     = A^^^2  = A~2^2
        I = F^   = B^3^     = A^^3^
        J = F^2  = B^3^2    = A^^3^2
```

- fork file with history

  git doesn't have cp subcommand. The equivalent to `git cp a b` is

  ```
  cp a b
  git add b
  ```

  git doesn't have means to track hisory of files. It tracks —blobs—. See gitcore-tutorial for details. The history is available in, e.g. —git log—, through —-C— option (Then git tries to find copies. It is good idea to commit forked files together – then -C has clue where to search. If you would not, —-C -C— (double -C) or –find-copies-harder (or so) does the trick) http://www.gelato.unsw.edu.au/archives/git/0612/3450

- proxy

  if you clone via `https` and happen to be behind a proxy, here is how you go:

  ```
  git config --global http.proxy http://proxydomain:proxyport
  ```

  and to inspect a configured option:

  ```
  git config --get http.proxy
  ```

2.1.4 Proposal for simplifying mlr git macros!

GIT-newBranch $1 create a new local branch; also push the branch to maserati; set the default tracking (the remote branch will always be named equally)

GIT-getBranch $1 create a local branch tracking an existing remote one called maserati/$1; ensures that the branch will always be named equally; ensure that the default tracking/mergin paths are set properly

GIT-delBranch $1 remove the local! branch; ensure that everything has been pushed to maserati;

GIT-info dump all possible info (git branch -a, git status, git info etc..)

2.1.5 Standard git workflows

Merging individual local branch with remote branch   Assume you are in the individual branch `branch123` on your local machine and want to merge with the `master` branch in the remote repository. Here, `origin` is a variable for the remote git repository server (i.e., maserati in our case).

```
// You are in the individual branch branch123 on your local machine.
git fetch origin master      // fetch from remote master
git merge origin master      // merge from remote master
git push                     // push local branch123 to remote branch123
git checkout master          // switch to master branch
git fetch origin branch123   // fetch from remote branch123
git merge origin branch123   // merge from remote branch123
git push                     // push local master to remote master
```

2.2 svn workflow

```
svn co --username=<YOUR ZEDAT LOGIN> --password=<YOUR ZEDAT PW> https://svn.imp.fu-berlin.de/mlr
svn add|del|mv <files>
svn up
svn commit -m 'message'
```
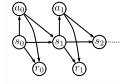
2.3 LaTeX Conventions

2.3.1 References

```
\usepackage[round]{natbib}
\bibliographystyle{abbrvnat}
```

2.3.2 Generating figures / graphical models

1. Use Inkscape. See the example mdp.svg in the pics path (note the —— command):

   Use the 'Tex Text' plugin. Use layers to generate animations. Use 'save a copy' to export as pdf (potentially with some layers switched off)

2. Use xfig with 'special tag' text. Use the following fig2pdf script to convert:

```
rm -f $1.pdf
rm -f $1.tex
fig2dev -Lpdftex -p $* $1.fig $1.pdf
fig2dev -Lpstex_t -p $* $1.fig $1.tex

cp $HOME/write/tex/figs/header.tex z.tex
printf "%s\n" "\\input{$1.tex}" >> z.tex
printf "%s\n" "\\end{document}" >> z.tex
more z.tex
pdflatex z.tex
pdfcrop z.pdf $1.pdf
```

2.3.3 Space tricks

Be creating in using any of the following

```
\renewcommand{\baselinestretch}{.98}
\renewcommand{\arraystretch}{1.2}
\renewcommand{\textfloatsep}{3ex}
\renewcommand{\floatpagefraction}{.6}
\renewcommand{\dblfloatpagefraction}{.6}
\setlength{\mathindent}{2.5em}
\setlength{\jot}{0pt} %zwischen den math zeilen
\setlength{\abovedisplayskip}{-10pt}
\setlength{\belowdisplayskip}{-10pt}
\setlength{\mathsurround}{-10pt}
\renewcommand{\floatsep}{-1ex}
\renewcommand{\topfraction}{1}
\renewcommand{\bottomfraction}{1}
\renewcommand{\textfraction}{0}
\columnsep 5ex
\parindent 3ex
\parskip 1ex
```

Or more compact lists:

```
\begin{list}{--}{\leftmargin4ex \rightmargin0ex \labelsep1ex
  \labelwidth2ex \topsep-\parskip \parsep.5ex \itemsep0pt}
\item ...
\item ...
\end{list}
```

The —list— parameter documentation:

```
* \topsep amount of extra vertical space at top of list
* \partopsep extra length at top if environment is prececed by a blank line (it should be a rubb
* \itemsep amount of extra vertical space between items
* \parsep amount of vertical space between paragraphs within an item
* \leftmargin horizontal distance between the left margins of the environment and the list; must
* \rightmargin horizontal distance betwen the right margins of the enviroment and the list; must
* \listparindent amount of extra space for paragraph indent after the first in an item; can be n
* \itemindent indentation of first line of an item; can be negative
* \labelsep separation between end of the box containing the label and the text of the first lin
* \labelwidth normal width of the box containing the label; if the actual label is bigger, the n
* \makelabel{label} generates the label printed by the \item command
```

2.3.4   Generating pseudo code

See Algorithm 1

Algorithm 1 Gauss-Newton with adaptive Levenberg Marquardt parameter

Input: start point x, tolerance δ, routines for $x \mapsto (\phi(x), J(x))$
Output: converged point x

1: initialize $\lambda = 1$
2: compute (ϕ, J) at x and $l = \phi^\top \phi$
3: **repeat**
4:     compute Δ to solve $(J^\top J + \lambda \mathbf{I}) \, \Delta = - \sum J^\top \phi$
5:     $x' \leftarrow x + \Delta$
6:     compute (ϕ, J) at x' and $l' = \phi^\top \phi$
7:     **if** $l' > l$ **then**
8:         $\lambda \leftarrow 2\lambda$
9:     **else**
10:         $\lambda \leftarrow 0.8\lambda$
11:         $x \leftarrow x'$
12:     **end if**
13: **until** $\|\Delta\| < \delta$

2.4   Coding Conventions

1. Take the Google C++ style as default: http://google-styleguide.googlecode.com/svn/ trunk/cppguide.xml. The rules below overwrite Google conventions.

2. Output parameters before input parameters in function declarations.

3. Naming convention —name_of_a_routine(ouput,input);— is perhaps better than my old —nameO-fARoutine(..);— style.

4. Make headers as clean and short and concise as possible!

5. Try to avoid —#includes— in header files as much as possible!! In particular, when a class's methods require an external library; this external library should be included only in the cpp-file.

   → In particular, try to avoid —#includes ¡external_lib¿— in headers!! (I tried to get rid of them as much as possible in all my *.h)

6. Try to avoid —#includes— in header files as much as possible!! If you think the class needs to contain members/data structures defined in an external library and therefore you need to include its header in your header — that's often not true. Instead, hide all members in a *"hidden self"*:

   Bad example:

```
//h-file:
#include <OpenCV> //BAD

struct MyClass{
  OpenCV_DataStructure data;
};
```

Good example:

```
//h-file:

struct sMyClass; //forward declaration of a 'hidden self' that will
contain all members hidden from the header

struct MyClass{
  sMyClass *s; //maybe call it 'self' instead
};

//cpp-file:
#include <OpenCV> //GOOD

struct sMyClass{
  OpenCV_DataStructure data;
};

MyClass::MyClass(){
  s = new sMyClass;
}
```

7. Move documentation to cpp files. Advantages: You can write as long and lengthy documentation as you want without destroying the beauty of the header. Doxygen will compile this without problem to provide a nice documentation. If people want to read the documentation from source directly—it's not much of a hassle to find the definition in the cpp file.

8. Never ever use —#ifdef— directives in a header file if this influences definition of classes, especially which members (and 'size') a class has!

   The following debugging horror may happen: You define a class to have different members depending on a compiler flag. You compile your library with some compiler flags. The user includes your header with other compiler flags. Everything seems to compile and link fine. But when the user accesses members of the class, he actually refers to different memory addesses as your routines in your library.

   Therefore try to avoid —#ifdefs— in headers as much as possible! Move them to the cpp file!

9. If you don't have a preferred IDE, use kdevelop.

   K&R formatting conventions!

   Editing: use spaces instead of tabs. 2 characters. Identation with 2 characters.

   Keys: F10,11,12: step, step in, step out, F9: toggle break
   F8 clean, F7 build, F5 debug run, CtrlF5 run, F6 continue

10. fltk lists their conventions – I like them, also their style of formatting and their makefile conventions

3   Points to improve the code

Penetrating collisions The way penetrating collisions are handled is really not nice —ors_swift.cpp:318—. What should ideally happen: find the two points on the meshes that penetrate most, set d as their negative distance, and the penetration normal aligned with their distance. What happens now: set the

distance $d = 0$ and set the normal align with the shape centers. The problem: the collision gradient is not really correct.

What to do: Actually most other collision libs focus on penetration (bullet/ode/solid), only SWIFT focusses on distance queries. What one would have to do, call one of these other collision libs to compute the most-penetrating two points of convex shapes.

3.1 Marc's older notes for code

1. Processes may not access MT.cfg in their step routine (open() is acceptable, but still not optimal).

   Instead, if necessary, parameters need to be made a proper variable. A special gui process offers a way to modify the variable online by the user.

2. The —Lock— (from —process_internal.h—) should actually not be used. The fact that it is used right now shows bad programming: It is always used when one process (or main loop) accesses directly another process instead of via a variable. Change that!

3. Make some central —ors— variable, or so, from which processes can update their ors-structure. Perhaps that needs some signalling – currently this orsCopyStuff after the connectivity is changed (object glued to hand/released) is not at all clean (related to point above).

4. The motion planners so far only optimize the next to go motion. Ideally there should be parallel planners optimizing already future motions → Dmitry's keyframe approach.

5. Make a Variable out of the TaskVariableList!!

6. All variables should have setters and getters of the format

   —arr& get_qreal(Process*);—

   —void set_qreal(Process*,const arr&);—

   These throw errors if called from a process that hasn't locked the variable or is a 'set' is called in readLock.

7. robot.cpp:281: read the current state in the schunk.open() routine! not here!

8. remove setting accelerations in each step of schunk hand motion
   – use radians (SDH constructor) out of the box
   – set and get velocites with one function call to schunk lib

9. SOC: make consistent to what is written in the TOMSY document

10. GRASP HEURISTICS: 1) Instead of conditioning velocities for lifting and placing, condition positions directly. 2) For grasp-approach: instead of conditioning only final position, condition on an approach position-profile: namely in the reference frame object-relative-to-hand. In both cases, use sinus position profiles?

3.2 Nikolay's ideas for vision

1. improve quality of color segmentation, while staying cheap computationally: learn a discriminative classifier $f(x)$ to separate by HSV color pixel of interest x from background

2. to get an intuitive interface without too much program overhead: create mask of desired discrimination as an image and train classifier in external program (matlab or C++), to output model parameters

3. use a mixture of k Gaussians for the activation of pixel x in HSV space: $f(x) = \sum_k \alpha_k exp(-\frac{\|c_k - x\|^2}{\sigma_k})$, probably 2 Gaussians are already more than enough, the current code is equivalent to $k = 1$ single Gaussian

4. just load parameters of trained Gaussian mixture and incorporate in earlyvisionmodule.step(): for each gaussian k we will calculate one response image and than sum them, to use the existing data structures make $\sum_k \alpha_k = 1$

5. optionally, use RGB space in addition to HSV space

6. use the above techniques for the LEDtracker... should simplify dealing with reflections

7. For an example how an image interface can be useful to define labels for training discriminative segmentation: few brush strokes are enough to define regions of foreground and background see frogsegment.jpg, (algorithm implemented by me in matlab)