The MLR Guide

Marc Toussaint

September 28, 2011

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 How to setup accounts, Ubuntu packages, external libs

- You need at least an account on achtauge and redmine (https://maserati.mi.fu-berlin.de/redmine/projects/mlr)
- The default directory layout:
  - `~/git/mlr` for your git repository
  - `~/lib` for your external lib installations
- Create your git repository:

```
cd ~/git
git clone ssh://<achtauge-user>@achtauge.imp.fu-berlin.de/mnt/data/git/share mlr
```

  See section 2.1 on how to use git.

- Install the following packages:

```
liblapack-dev
freeglut3-dev
libqhull-dev
libf2c2-dev
libann-dev
libplib-dev
gnuplot
meld
```

- For a full installation (e.g., interaction with the robot) one also needs the external libs

```
cuda@
cudaSDK@
opencv-2.1@
ntcan@
schunkLWA@
schunkSDH-11-05-11@
```

  These are symbolic-linked in the `git/share/extern` directory. Call the `./LINK ~\lib` script to generate the relevant links.

- TODO: In the future all external libs should be handled like ODE_0.11: minimal GET/BUILD script and .gitignore
- Try 'make' in the `git/share` directory. 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.

- Generally, the `git/share/make-config` allows to set compile options for all projects, the local Makefiles trigger these options by defining 'ANN=1' or alike.
- Try test/array/x.exe

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

1.6 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.

1.6.1 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.

1.6.2 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.

1.6.3 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.

1.7 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.

1.8 Guide to further existing software

1.8.1 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.

1.8.2 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

1.8.3 Computer Vision methods (Marc)

LINK to my notes on the CV system

1.8.4 Robotics algorithms (Nikolay & Marc)

- RRTs
- Trajectory Prediction
- etc

1.8.5 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/

1.8.6 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 workflow

Most likely (I would wonder if not) the git man pages are installed together with the git binaries in your distribution. The fastest way to check the syntax or parameters of a git subcommand are the man pages.

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.

Usually, `git foo help` is synonym for `man git foo`.

```
git clone ssh://<achtauge-user>@achtauge.imp.fu-berlin.de/mnt/data/git/share mlr
git config --global diff.external git-diff.sh
git config --global push.default tracking

git status
git diff
git add|mv|rm|... <files>
git commit -am 'message'  (or --all --message 'message')
git pull --all
git push --all

git branch -a
git remote show achtauge
git checkout <branch>
git branch <new-branch>
git branch --track <new-branch> <remote-branch>
git branch --set-upstream <existing-branch> <remote-branch-to-be-tracked>
git merge <other-branch>

.gitignore
git update-index --assume-unchanged <files>
```

with a git-diff.sh file (in your local bin)

```
#!/bin/bash
/usr/bin/meld "$2" "$5" > /dev/null 2>&1
```

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 HEAD^^^2` shows the second branch before merging. In other words, direct after merge and commit, the parents of HEAD are `HEAD^1` and `HEAD^2`

  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/0612/34502.html

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'
```

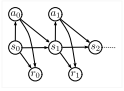
2.3 LaTeX Conventions

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 `\graphicspath{{pics/}}` 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 rubber length)
  * \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 be nonnegative
  * \rightmargin horizontal distance betwen the right margins of the enviroment and the list; must be nonnegative
  * \listparindent amount of extra space for paragraph indent after the first in an item; can be negative
  * \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 line of an item
  * \labelwidth normal width of the box containing the label; if the actual label is bigger, the natural width is
  * \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

initialize $\lambda = 1$
compute (ϕ, J) at x and $l = \phi^\top \phi$
repeat
  compute Δ to solve $(J^\top J + \lambda \mathbf{I})\ \Delta = -\sum J^\top \phi$
  $x' \leftarrow x + \Delta$
  compute (ϕ, J) at x' and $l' = \phi^\top \phi$
  if $l' > l$ **then**
    $\lambda \leftarrow 2\lambda$
  else
    $\lambda \leftarrow 0.8\lambda$
    $x \leftarrow x'$
  end if
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 `nameOfARoutine(..);` 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