Relja_Retrieval v2.0:

Manual for Black Box Usage

Relja Arandjelović

24 September 2014

1. Introduction

This document demonstrates how to use the relja_retrieval library that I have developed during my DPhil/Postdoc in the Visual Geometry Group, Department of Engineering Science, University of Oxford. The document is meant for people who don't care how it works and just want to use it to index a database (on a single machine or on a cluster), or run the backend/frontend, or communicate with the API. For details on how it all works and how to develop with the relja_retrieval library, consult **docs/Conceptual_Overview.pdf** and **docs/Manual_for_Developers.pdf** .

2. Prerequisites

- Ability to install dependencies (quite straight forward especially on Ubuntu)
- Basic Linux usage
- Ability to compile/link a c++ program (well, to debug if something goes wrong in the building process, e.g. due to missing libraries, multiple versions of a library, etc)

3. Install dependencies

install/ contains relevant scripts for Ubuntu, cd into it.
For all the instructions I assume one has administrator rights for the machine. If not, one needs to talk to the administrator to install the dependencies, or install each one locally. How to do this is beyond the scope of this manual as it is basic knowledge unrelated to relja_retrieval (i.e. make sure all dependencies are in include and library paths, for python libraries they should be in PYTHONPATH).

Note for installing on Titan: you should install everything from an interactive session (**qrsh -pe mpislots 1**); some dependencies are already installed (boost, openmpi, python-numpy) so you can simply load them as modules (you should also load the relevant g++ and python modules):
for modname in apps/cmake/2.8.9/gcc-4.4.6 apps/python/2.7.3/gcc-4.4.6 compilers/gcc/system libs/boost/1.51.0/gcc-4.4.6+openmpi-1.6.3 libs/gcc/system libs/numpy/1.6.2/gcc-4.4.6+python-2.7.3+atlas-3.10.0+westmere mpi/openmpi/1.6.3/gcc-4.4.6; do module add $modname; done

For various usage requirements, different dependencies are needed:

3.1. To do it all

This is the fast track without understanding details. For details see the following subsections.

Need to install:

- See following subsections

Most of it can be done easily on Ubuntu:
./install_all_deps.sh

Now you only need to install FASTANN and pypar (external dependencies).

- FASTANN:

cd relja_retrieval_all/fastann
mkdir build; cd build
cmake ../src
make
sudo make install

- pypar:

cd relja_retrieval_all/pypar_2.1.4_94/source
python setup.py build

sudo python setup.py install
To check if pypar has been installed properly (make sure to leave the pypar directory, e.g. go to **relja_retrieval_all/** , as otherwise this will fail!):
mpirun -np 3 python -c "import pypar; pypar.finalize();"
Should give:
“Pypar (version 2.1.4) initialised MPI OK with 3 processors”
If you get an error saying something like libmpi.so.0 not found, do:
sudo ln -s /usr/lib/libmpi.so /usr/lib/libmpi.so.0

3.2. To run the precompiled backend:

Need to install:

- libboost: It has several submodules, the backend requires:
    - thread, system, random, program-options, filesystem, date-time
- libmagick++
- libprotobuf
- FASTANN: available at http://www.robots.ox.ac.uk/~vgg/software/fastanncluster/ , but you should have received a copy with a tiny change (removing the ‘-march=native’ from compilation flags as this is problematic in cases where CPUs might be different, e.g. on a cluster built of heterogenous nodes)

On Ubuntu: The first three can be installed simply by calling the
./online_backend.sh

You will probably need to build FASTANN from scratch unless someone gave you a precompiled version. For this you will need g++, make and cmake, to install these on Ubuntu:
sudo apt-get install build-essential cmake
Now to build FASTANN see its README.txt, but basically do:
cd relja_retrieval_all/fastann; mkdir build; cd build; cmake ../src; make; sudo make install;

3.3. To run the frontend

Need to install:

- python 2.x
- python-numpy
- python-cherrypy3
- (optional but recommended) jp_draw . Usually one does want to install this in order to be able to draw matches on the details page. This in turn requires:
    - libcairo2
    - libjpeg62

On Ubuntu all of these apart from jp_draw (but including jp_draw's dependencies) can be installed easily by calling:
./online_frontend.sh

You will probably need to build jp_draw unless someone gave you a precompiled version (unlikely). Read the src/external/jp_draw/README on how to do this, but basically:
Install dependencies on Ubuntu:
sudo apt-get install build-essential python python-numpy cython libcairo2-dev libjpeg62
Then do the standard Python installation procedure:
cd src/external/jp_draw; python setup.py build; sudo python setup.py install;

3.4. To build the backend

Need to install:
- Everything from "to run the precompiled backend"
- build tools: g++, make, cmake
- imagemagick
- protobuf-compiler

On Ubuntu:
./build_backend.sh

3.5. To build the frontend

Need to install:
- Everything from "to run the frontend"
- build tools: g++, make
- cython

On Ubuntu:
./build_frontend.sh

3.6. To be able to perform indexing

Need to install:
- Everything from "to build the backend"
- (optional but recommended) openmpi, libboost's mpi submodule: these are needed if a cluster is going to be used for indexing. Not needed if all indexing will be done on a single CPU (really not recommended) or using a node with many CPUs (this works fine, though visual vocabulary computation doesn't support this, indexing does)
- (optional but recommended) pypar, dkmeans_relja: these are needed for building a visual vocabulary (well pypar isn't technically needed but for practical purposes it is,

unless you want to use a single core for this)

To install all apart from pypar on Ubuntu:
./build_offline.sh

pypar is just required for building a visual vocabulary on a cluster (with MPI). You should have received a modified copy of pypar_2.1.4_94 (if for some reason you need a different version of pypar then you need to figure out how to modify the patch in src/external/dkmeans_relja/pypar_2.1.4_94_patch/). To install follow the standard Python installation procedure:
cd relja_retrieval_all/pypar_2.1.4_94/source/; **python setup.py build; sudo python setup.py install;**

To check if pypar has been installed properly (make sure to leave the pypar directory, e.g. go to **relja_retrieval_all/** , as otherwise this will fail!):
mpirun -np 3 python -c "import pypar; pypar.finalize();"
Should give:
“Pypar (version 2.1.4) initialised MPI OK with 3 processors”
If you get an error saying something like libmpi.so.0 not found, do:
sudo ln -s /usr/lib/libmpi.so /usr/lib/libmpi.so.0

To install dkmeans_relja go to **src/external/dkmeans_relja/** and again do the standard python installation:
python setup.py build; sudo python setup.py install;

4. Build the code

This part requires cmake, but even if you don't know about it, it should be quite straight forward. Go to the root directory of relja_retrieval, do:
mkdir build; cd build
cmake -i ../src

This will ask you for a few options regarding what you want to use the code for, e.g. if you don't need it for indexing on a cluster certain parts of the code will not be included and you won't require certain dependencies.

For all options that are fine and I don't mention here, just press <enter> to use the defaults. This is a place where if a dependency is not found, the program will report an error and tell you which library is missing. Also if you decided to install libraries to non-standard locations (e.g. this might be the case when you don't administrative rights, e.g. on the cluster, to specify locations of the libraries; a typical example is FASTANN which won't be installed on the cluster). More about options:

CMAKE_BUILD_TYPE : Type in **Release** for fastest runtime.
cMPI : If you plan to use the code on the cluster for indexing, type in **ON**, otherwise **OFF** .
cREGISTER : If you want to be able to register images in the online demo (i.e. only if you are using my frintend) : **ON** otherwise **OFF** .

If cmake finishes with errors, fix the missing library problems, otherwise we are ready to compile the stuff. For this we use **make** (you can try **make -j** for building using multiple threads, I do it generally but it used too much processing power sometimes).

- If you want to run indexing: **make compute_index_v2**
- If you want to run the backend: **make api_v2**

That's all.

4.1. Build on a cluster (Titan example)

As hinted earlier, setting things up on a cluster requires slightly more work as you typically won't have administrator rights, so for the libraries you installed locally (most likely FASTANN, dkmeans_relja, protocol buffers) you will need to type in the relevant paths during the **cmake** command. Otherwise, the process is the same as before.

Important note: You don't want to build on the head node, but you want to log in into an interactive shell and compile there. This is because library versions will be different (e.g. boost is likely to exist on head node, but the computational nodes will have newer boosts), libraries will be missing on the head node, and the head node probably is physically different than the computational nodes. This holds for any compilation, not just relja_retrieval, so if you are compiling dependencies (e.g. FASTANN) you should do it the same way!

Here is an example how to do it on Titan (setup from September 2014). First go into interactive shell:
qrsh -pe mpislots 1
Now set up the compilation environment:

for modname in apps/cmake/2.8.9/gcc-4.4.6 libs/boost/1.51.0/gcc-4.4.6+openmpi-1.6.3 libs/gcc/system mpi/openmpi/1.6.3/gcc-4.4.6; do module add $modname; done

This basically tells the cluster to set up libraries such as boost etc.
Now do the same process as above with **cmake -i ../src**
All my locally installed dependencies I put into
/users/relja/Relja/Programs/commonC
so I can quickly tell cmake to search for missing libraries there:
cmake -DCMAKE_PREFIX_PATH=/users/relja/Relja/Programs/commonC -i ../src

You can use my installations of the libraries if you want, but I would recommend you to install them locally into your home directory as: 1) I don't know how long will I have my home directory on titan for, 2) I might use it and change the installations therefore potentially breaking your jos, 3) you should learn how to set everything up, e.g. if titan is upgraded you will likelly need to reinstall the dependencies and rebuild relja_retrieval .

4.2. Build to deploy

Note, currently the default setup is to build a bunch of shared libaries. To build only a single library, which simplifies copying the library around, change the **set(BUILD_SHARED_LIBS "ON")** line in **src/CMakeLists.txt** ON -> OFF.

5. Set up configuration files

The configuration file is in **src/ui/web/config/config.cfg** (the location isn't the most fortunate one as I used to use the configuration file just for setting up the frontend, but now it is also used for indexing so ideally the file should be somewhere else). It has sections which you are free to call however you want (probably avoid spaces and strange characters), which correspond to particular indexing / runtime details for one database and for one setting of parameters (e.g. vocabulary size). The sections start with [section_name], and this section_name is exactly what you will use later when you run indexing, the backend or the frontend (there I call it dsetname). Note: to comment out a line, start it with # (not spaces allowed before it, it needs to be the first character).

Have a look at the first section **[ox5k]**, it should be relatively self-explanatory. There are a bunch of variables that need to be defined:

- Descriptor details:
    - *RootSIFT*: should always be set to **true**, specifying to use RootSIFT and not SIFT
    - *SIFTscale3*: whether to use the default scaling (support area from which SIFT is extracted) used by James Philbin (e.g. in his 2007 paper), or use the better scale=3 (as used default by most people); set this to **true** always to indeed use the better setting.

- My frontend options:
    - *titlePrefix*: Prefix of the title of every web page
    - *defaultView*: Can be *tile* to use the tile view by default (e.g. see the ballads demo on vgg web page)
    - *pathManHide*: When displaying results, remove this prefix from the absolute file name in order not to look better
    - *homeText*: What is written on the top left part of the page for the "home" link
    - *topLink*: A link used for the "home"
    - *bottomText*: To add text to the bottom of every page
    - *apiScoreThr* : Threshold on the score returned by my API

- Indexing options:
    - *tmpDir*: Where to store temporary data
    - *dsetFn, clstFn, iidxFn, fidxFn, wghtFn*: The names of relevant files to be created (for developers, the files are: the dataset information, visual vocabulary, inverted index, forward index, and idf weights, respectively)
    - *imagelistFn*: This file contains the list of images to be indexed (see more a bit later)

    - *databasePath*: The path to the database (see more a bit later)
    - *trainImagelistFn* and *trainDatabasePath*, if specified these are used for the training descriptors, otherwise they are they default *toimagelistFn* and *databasePath* (see the description if those for more info). Generally for real-world datasets you wouldn't include these, but for scientific papers, e.g. reporting results on Oxford while training on Paris, you need it.
    - *trainNumDescs*: What subsample of descriptors to use for training the vocabulary (this is an upper bound - if the training database is small, the number of training descriptors will be smaller). I generally don't go over 20M. A special value -1 is allowed which means "use all descriptors"
    - *vocSize*: The size of the visual vocabulary (=number of cluster centres), for BoW usually 500-100k, for Hamming usually 100-200k (200k is probably better, for speed of retrieval, if your database is large)
    - *trainFilesPrefix*: Where to store any additional files files produced by training (e.g. Hamming Embedding information)
    - *hammEmbBits*: If Hamming Embedding is used, specify the number of bits (typically 64 for smallish datasets, 32 for large). Remove this line if you want BoW, though I don't recommend it

- Runtime backend options:
    - Most of the stuff from "Indexing options" and "Descriptor details" is needed: *RootSIFT, SIFTscale3, databasePath, dsetFn, clstFn, iidxFn, fidxFn, wghtFn, vocSize, trainFilesPrefix, hammEmbBits*

More details about setting the variables *imagelistFn*, *databasePath* :
imagelistFn should contain all the images to be indexed, e.g. for Oxford Buildings 5k the list for me starts with (~/Relja/Databases/OxfordBuildings/imagelist_oxc1_5k.txt):
oxc1/new_000355.jpg
oxc1/oxford_000637.jpg
oxc1/new_000818.jpg
..
(the first time I made it I forgot to sort it by file name, but it doesn't matter)
The files are listed using relative paths and not absolute paths - for readability and portability. The root of the database is set in *databasePath* . Now if you need to move the database somewhere else, you just change *databasePath* , and all still works. As can be seen from the config, my value of *databasePath* is ~/Relja/Databases/OxfordBuildings/ , and this should be such that *databasePath*+items in the *imagelistFn* file = full absolute file names. In other words, the *imagelistFn* will be internally converted to:
~/Relja/Databases/OxfordBuildings/oxc1/new_000355.jpg
~/Relja/Databases/OxfordBuildings/oxc1/oxford_000637.jpg
…

Important to note: *databasePath* should end with a / while the items in *imagelistFn* shouldn't start with a / . It’s semi-intentionally fragile because of some peculiarities with the AXES API that you don’t need to worry about.

The way to generate an *imagelistFn*:

- If working with a video database, someone should provide you with a list of keyframes that you potentially might need to run some simple text processing (e.g. look up how to use: grep, sed, awk or just make a python script to do it)
- If you need to generate it yourself (often the case), you are most likely best off to use the find command, e.g. cd into the *databasePath* folder and:

  find * -print -type l -o -type f | grep jpg > imagelist.txt

  This basically tells it to list only files and not folders, also include soft links, and to only keep files which contain jpg in them. Note that you should probably change this command as it will theoretically include files called bla_jpg_bla.txt, or maybe you don’t want to concentrate only on jpg but want to include png etc (also should probably use case invariant version of grep in order to include *.JPG files etc). Often image databases are organized nicely so that only images are in it and there are no soft links, in which case, you would just do:

  find * -print -type f > imagelist.txt

  For **AXES** demos - it is quite specific what should be the exact paths returned by the APIs etc, so be careful to test it properly and pay attention to make sure returned paths are ok (check with other AXES involved people in VGG). They might give a folder which contains some meta files, and then the folder with frames is in a subfolder called something like c<database_name>, e.g. cAXESMini/ . It is important that your imagelist.txt doesn’t start with cAXES but from a folder within, so the *databasePath* will be path/to/cAXES/ while imagelist.txt will contain the remainder of the path, e.g.
  v20080512_001000_bbcone_weatherview/s000000001/keyframe000000024.jpg
  v20080512_001000_bbcone_weatherview/s000000001/keyframe000000076.jpg
  v20080512_001000_bbcone_weatherview/s000000001/keyframe000000132.jpg
  …

6. Perform indexing

Read the “Set up configuration files” section first.

This is strongly recommended to be done on the cluster, as otherwise it will take ages to run. However, just to get familiar with how to index everything for the first time, it is best to do it on your machine with a small dummy dataset (e.g. 10 images, and make the visual vocabulary tiny, e.g. 1000).

Tip: Physical location of your data will change across servers, e.g. mounting points for various disks are different on your local machine and on a cluster. I found it very useful to standardize this as much as possible. WIth a new database, I copy it to wherever there is space (e.g. /work3/relja/Databases/NewOne) and determine where all the new data (e.g. inverted index) should be stored (e.g. /mnt/sharedscratch/users/relja/NewOne), and then I create soft links to a standard location, e.g. ~/Databases/ , ~/Data/). This enables you to appear to have all your data at the same location at various machines, e.g. on your local machine you might place the database in /data2/db/ , you soft link it to ~/Databases, and now on both titan and your local machine you can access the relevant files in ~/Databases . Simple tip but very useful. If you don’t know about soft linking, it is very simple, e.g.
ln -s /mnt/sharedscratch/users/relja/NewOne ~/Data/NewOne

Set up a script for indexing, e.g. call a file _compute.sh and put inside:
~~~~~~~~~~~~~~~~~~

```
#!/bin/bash
#$ -cwd
source /etc/profile.d/modules.sh

for modname in apps/cmake/2.8.9/gcc-4.4.6 apps/python/2.7.3/gcc-4.4.6 compilers/gcc/system libs/boost/1.51.0/gcc-4.4.6+openmpi-1.6.3 libs/gcc/system libs/numpy/1.6.2/gcc-4.4.6+python-2.7.3+atlas-3.10.0+westmere mpi/openmpi/1.6.3/gcc-4.4.6; do module add $modname; done

export PYTHONPATH=$PYTHONPATH:/users/relja/Relja/Programs/commonPy/lib/python:/users/relja/Relja/Programs/commonPy/lib/python2.7/site-packages/

cd /users/relja/Relja/Code/relja_retrieval/build

dsetname=ox5k

mpirun -np $NSLOTS v2/indexing/compute_index_v2 trainDescs ${dsetname}
```
~~~~~~~~~~~~~~~~~~

```
mpirun -np $NSLOTS python ../src/v2/indexing/compute_clusters.py ${dsetname}
mpirun -np $NSLOTS v2/indexing/compute_index_v2 trainAssign ${dsetname}
mpirun -np $NSLOTS v2/indexing/compute_index_v2 trainHamm ${dsetname}
mpirun -np $NSLOTS v2/indexing/compute_index_v2 index ${dsetname}
~~~~~~~~~~~~~~~~
```

The modname line loads the appropriate modules (note that they are slightly different from the ones needed at compilation time, e.g. we included **numpy** needed for the python part, for building a visual vocabulary).

Then the export PYTHONPATH stuff sets up the paths towards the locally installed python libraries (i.e. dkmeans_relja, pypar).

The cd command goes into the build directory where the code has been built.

The dsetname=ox5k line sets up what dataset you want to index, i.e. this is the section name used in the configuration file.

Finally, the following lines (mpirun ..) do various stages of the processing, lines:

1) trainDesc: extracts a subsample of descriptors to be used for training (e.g. building a visual vocabulary, finding PCA needed for Hamming Embedding, etc)
2) compute_clusters.py : computes the visual vocabulary, i.e. runs approximate k-means
3) trainAssign : Needed only for Hamming Embedding (i.e. not needed for BoW), it assigns all the training descriptors (from (1)) to nearest visual words
4) trainHamm : Needed only for Hamming Embedding (i.e. not needed for BoW), it finds the appropriate Hamming Embedding parameters from unsupervised training data (PCA, random rotation, medians)
5) index : Most computationally expensive (along with compute_clusters.py), does the actual indexing.

Note: The code assumes that all the relevant folders you specified in the configuration file exist, so make them before running the indexing, otherwise it will fail! Also, if an image is not found it is simply skipped (e.g. there are sometimes a few corrupt images in a database of millions, you don't want indexing to fail and crash because of a single image). However, be careful, if the configuration file is not written properly, the indexing might not find any image and finish very early due to skipping everything.

You should run indexing on a tiny database first on your machine to see if all is ok. Then log into an interactive node and run clustering from there on a tiny database, to verify all is ok (**qrsh -pe mpislots 1** , and then run the **_compute.sh** script). Finally, when all looks ok, run the full code on the cluster.

On your local machine, you clearly don't need the cluster related stuff, but just run:

```
dsetname=ox5k
v2/indexing/compute_index_v2 trainDescs ${dsetname}
python ../src/v2/indexing/compute_clusters.py ${dsetname}
v2/indexing/compute_index_v2 trainAssign ${dsetname}
v2/indexing/compute_index_v2 trainHamm ${dsetname}
v2/indexing/compute_index_v2 index ${dsetname}
```

For testing if everything works with MPI, you can also prepend these 5 lines with
mpirun -np 4
to test it with 4 CPUs (otherwise the code will use multithreading for parallelization, but compute_clusters.py isn't able to do this so it will run in a single CPU).

When satisfied that everything works, submit the **_compute.sh** job on the cluster by running:
qsub -pe mpislots 20 _compute.sh
(where 20 is obviously replaced with as many CPUs as you want, usually more than 20)

I also find it useful, especially when indexing a large database, for Titan to email me automatically if the job is aborted or there is an error:
qsub -M youremail@robots.ox.ac.uk -pe mpislots 20 -m ea _compute.sh
I use this command often so I have created a script qsub_r :
~~~~~~~~~~~~~~~~~~
```
#!/bin/sh
qsub -M relja@robots.ox.ac.uk -pe mpislots $@
```
~~~~~~~~~~~~~~~~~~
and now I just do:
qsub_r 20 -m ea _compute.sh

Note: The program makes a file **indexingstatus.bin** which contains some debug information, e.g. if you stop indexing in the middle, and then resubmit the same job, it will be able to continue from where it stopped (well, not exactly from where it stopped, could be that the point in time is quite far behind, so it is certainly not recommended to restart, but could be useful). If you want to reindex the database (e.g. you made a mistake in the config file, etc), you will need to also delete the **indexingstatus.bin** .

To check job status (e.g. if it is running) investigate the *qstat* command, and if it is running the log will be written into _compute.sh.* files, ask labmates if you need more help with the cluster.

AXES demos: in recent times I switched to using a fixed vocabulary for all of them (it is actually required as they wish to be able to process an image once and query several datasets with the same features (i.e. visual words, hamming signatures etc). Therefore, for AXES don't actually run any of the training stages, just run indexing and use the same 200k vocabulary trained on cNisvPro and with 32-bit Hamming Embedding as I run always (e.g. see [cAXES_hamm] in the

config file, specifically *clstFn, vocSize, trainFilesPrefix, hammEmbBits;* clearly you also need to use the same descriptor settings, i.e. RootSIFT with SIFTscale3=true). The relevant files are in **relja_retrieval_all/data/** .
On the AXES BBC server, the binaries are in **~/servers/uo_instances/** (don't deploy source!) and data (e.g. inverted index, config) is in **/data/index/uo_instances/** .

7. Run the backend(s) / frontend(s)

Read the “Set up configuration files” section first.

<u>**My backend**</u>:

cd build
v2/api/api_v2 <backend_port> <dsetname>
e.g.
v2/api/api_v2 35280 ox5k
To have it restart if it crashes (if the process ends; doesn’t really happen), do:
./api_run.sh <backend_port> <dsetname>

<u>**My frontend**</u>:

cd src/ui/web/
python webserver.py <web_port> <dsetname> <backend_port>
e.g.
python webserver.py 8080 ox5k 35280
Similarly to the backend, an automatic restart version is also available:
./web_run.sh <web_port> <dsetname> <backend_port>

<u>**My backend and frontend at the same time in the background(+auto restart)**</u>:

./allstart.sh <web_port> <dsetname> <backend_port>
This also starts the backend and frontend inside a *screen* (see the linux command screen, very useful) each, so that they run in the background (e.g. you can log out, or you can set up a startup script to run this on boot of a demo machine).

<u>**VISOR-compatible backend API (i.e. for AXES/BBC)**</u>:

You should first start my backend, then start a wrapper around my backend that appears as a VISOR backend to everyone else, e.g. :
cd src/ui/visor
python rr_visor_backend.py 45288 BBCc_news_hamm BBCn 35288 u
The sequence of arguments here is::

- the VISOR backend port we are starting
- relja_retrieval name for the dataset (i.e. dsetname above when starting the API)
- a common name used by other people involved in AXES or BBC (e.g. BBCn here, or cAXESMini, etc)
- port of my actual backend (as explained earlier)
- just always write *u*, it specifies that uploading an image is enabled, which is always the case (I should remove it but I don’t want to do it in a rush and break things)

You can run several demos on the same backend port (their respective actual c++

relja_retrieval backends still need to run on a port each) by repeating the last 4 arguments arguments, e.g:
python rr_visor_backend.py 45288 BBCc_news_hamm BBCn 35288 u ox5k ox5k 35289 u

That's all that is relja_retrieval specific, for information on how to start VISOR frontend consult Ken Chatfield or Omkar Parkhi, as this is beyond the scope of this document.

7.1. Talking to my API

Assuming a relja_retrieval c++ backend is running on port 35280, talk to it by issuing:
src/ui/cmd/useAPI.sh localhost 35280

Every request should be terminated by: **END** (note there is a space before END)
An example where we perform a query using an image from the dataset, with docID (i.e. imageID) 50, ROI defined by corners (5,30) and (100,400), and we wish to return 100 top results:

```
<internalQuery><docID>50</docID>
<xl>5</xl><xu>100</xu><yl>30</yl><yu>400</yu>
<numberToReturn>100</numberToReturn>
</internalQuery> $END$
```

The command prompt should now print:

```
<results size="100"><result rank="0" docID="50" score="107.0027"
H="1.000000,0.000000,0.000000,0.000000,1.000000,0.000000,0.000000,0.000000,1.000000"/><
result rank="1" docID="2639" score="20.0005"
H="0.949123,0.000000,373.645623,-0.030421,0.678321,243.197432,0.000000,0.000000,1.0000
00"/> …
```

I.e. it returned a ranked list of results, including docID, score, and homography (if one was found, i.e. the result has been spatially verified). To visualize the results, see the next section.

Now let's see what's are the filenames of the query (usually the first result is the same as the query, as in the above example) and the second result:

```
<dsetGetFn><docID>50</docID></dsetGetFn> $END$
/home/relja/Relja/Databases/OxfordBuildings/oxc1/bodleian_000329.jpg
<dsetGetFn><docID>2639</docID></dsetGetFn> $END$
/home/relja/Relja/Databases/OxfordBuildings/oxc1/bodleian_000170.jpg
```

Looking at these two we will see that the result is correct (for visualization using the gui, see next section)

The full list of supported function and their arguments is easily parsed from from API::getReply() in **src/api/spatial_api.cpp** , or, even easier, from the python **src/api/api.py** and the API class. The python functions show exactly how to make the requests for every function (function names and arguments are self explanatory): ***internalQuery, getInternalMatches, getPutativeInternalMatches, getExternalMatches, getPutativeExternalMatches, externalQuery, multiQuery, processImage, register, registerExternal, dsetGetNumDocs, dsetGetFn, dsetGetDocID, containsFn, dsetGetWidthHeight*** .

For another example, to get the spatially verified matches between the query and the first result:

```
<getInternalMatches>
<docID1>50</docID1>
<xl>5</xl><xu>100</xu><yl>30</yl><yu>400</yu>
<docID2>2639</docID2>
</getInternalMatches> $END$
```

This returns:

```
<matches size="20"><match el1="64.00,191.00,0.00332356,-0.00037640,0.00114454"
el2="433.00,371.00,0.00425460,-0.00045629,0.00138595"/><match
el1="42.00,158.00,0.00240543,-0.00055158,0.00080560"
el2="413.00,351.00,0.00280013,-0.00105117,0.00144404"/><match
el1="43.00,161.00,0.00179557,-0.00054271,0.00063401"
el2="415.00,352.00,0.00205891,-0.00109575,0.00137658"/>
...
```

i.e. the list of 20 verified matches with ellipse position/shape specifications. See next section for visualization.

Changing getInternalMatches to getPutativeInternalMatches returns the putative matches (i.e. before spatial verification)

7.2. Using the frontend

Navigate to localhost:webport (e.g. http://localhost:8080/) and select images / ROIs to query with.

To execute the first example from the previous section, do:
http://localhost:8080/dosearch?docID=50&xl=5&xu=100&yl=30&yu=400&numberToReturn=100
We can see that, indeed like the previous section states, the query/first result is
oxc1/bodleian_000329.jpg
and the second result is:
oxc1/bodleian_000170.jpg
Note that not the entire path is displayed - that's because the configuration file states:
pathManHide= ~/Relja/Databases/OxfordBuildings
(see section 5.)

Now let's click on the "Detailed matches" link for the second result, tick "Lines" and tick "Regions" and click "Draw again". This now displays the same 20 ellipse pairs as the second example in the previous section. Ticking further on "Putative" and redrawing shows the same matches as the ones from the continuation of example #2 in the previous section.

7.3. Talking to VISOR backend API

This is beyond the scope of this manual as it is not specific to relja_retrieval. However you should know how to do it if you wish to deploy a new AXES demo. You will need access to:
https://bitbucket.org/kencoken/visor/wiki/backend-api
For access to this and more information on VISOR frontend consult Ken Chatfield or Omkar Parkhi.

For convenience, I have a hacky python script which you can use:
jsonrpc.py (it needs access to the **api_request.py** from **ui/web/**)
~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
#!/usr/bin/python
import sys
import json;
from api_request import customRequest;

url= 'localhost:42588';

arg= int(sys.argv[1]);
qid= int(sys.argv[2]);

if arg==0:
    data= {'func': 'getQueryId', 'dataset': 'cAXES'};
elif arg==1:
    data= {'query_id': qid, 'from_dataset': 1, 'func': 'addPosTrs', 'impath':
'/cAXES/v20080512_223000_bbcfour_the_da_vinci_code_the_greatest/s000000034/keyframe000006576.jpg'};
elif arg==11:
    data= {'query_id': qid, 'func': 'saveAnnotations', 'filepath': '/home/axes/servers/uo_instances/tmp/anno.txt'};
elif arg==2:
    data= {'query_id': qid, 'func': 'train'};
elif arg==3:
    data= {'query_id': qid, 'func': 'rank'};
elif arg==4:
    data= {'query_id': qid, 'func': 'getRankingSubset', 'start_idx': 0, 'end_idx': 10};
elif arg==5:
    data= {'query_id': qid, 'func': 'releaseQueryId'};

reply= customRequest( 'localhost', 45288, json.dumps(data), appendEnd= True, kTcpTerminator= "$$$" )

print reply;
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~
You will need to replace cAXES with the name of the new dataset, and update the hardcoded paths (url, impath, filepath). Then do:

```
./jsonrpc.py 0 0; // prints out a qid which you should use for future commands
for i in `seq 1 5`; do ./jsonrpc.py $i $qid; done
```

If you want to test uploading a file, change the second line to:

```
for i in 1 11 `seq 2 5`; do ./jsonrpc.py $i $qid; done
```