# Homework 3: Salient Points and Google-lite

Goals: Explore and gain practical insight into
- Salient points, image matching, similarity and RANSAC
- Web search engines
- Text and image queries, indexing and ranking results

## (1) Salient Point Detector: SIFT (low difficulty)

This part works on the machines in room 302/303 under Linux and was written by Song Wu.

Several talks in class have introduced and discussed the SIFT salient point detector and descriptor. In the file, **getsift.zip**, there is source code for

- SIFT salient point detector
- SIFT salient descriptor
- Simple matching -> file matches.jpg
- Advanced matching using RANSAC -> file matches.ransac.jpg

RANSAC is a technique used in certain situations to perform spatial matching between two sets of 3D points. Specifically, it tries to find the set of correspondences between the two sets that fit an affine transformation (translation, rotation, scale).

First, make a directory (i.e. sifthomework) and then unzip the file getsift.zip to that directory. All of the source code is included. To compile the source code type "make" at the command line. The file that calls the sift functions is called test.cpp. It shows how to call the functions we use.

After you compile the source code, you will get an executable named getsift which is called with 2 arguments.

**(a) Compare the notredame images using**

```
getsift notredame1.jpg notredame2.jpg
```

Examine the visualizations of the sift points from both images (notredame1.jpg.sift.jpg and notredame2.jpg.sift.jpg). Please only spend a five minutes comparing the point visualizations and then give some “rough” comments on how often the salient points are found between the images (i.e. rarely, sometimes, often, etc.). Also, insert a screenshot showing the image results.

**(b) Compare matching methods.** Run getsift with cruise1.jpg and cruise2.jpg. Comment on how accurate the salient point detector is by looking at the simple matches from matches.jpg and the advanced matching in matches.ransac.jpg. Explain.

**(c) Evaluate Rotation Invariance.** Put a book on a table and take some medium resolution (roughly 800x600) pictures at different angles in-plane with the book and out-of-plane (e.g. 0 degrees, 10 deg, 20 deg, 30 deg, 40 deg, 50 deg, 60 deg.). Insert a screenshot showing your images of the book and comment on how the accuracy of the simple and advance matching changes when the angle increases. At what point does it fail completely (or not)?

**(d) Which images are more similar?** Note that “main” is in test.cpp. Alter the code so that three images are given to a program “similarimage” which tells you which of the input images is most similar to the first.

```
similarimage [image1.jpg] [image2.jpg] [image3.jpg]
```

Based on “m” which is the “total matches”, have your program output whether [image2.jpg] or [image3.jpg] was more similar to [image1.jpg]. Take a screenshot of your program working for the report.

## (2) Web Spider (medium difficulty)

In this homework, we will be utilizing the functions that you wrote in the previous homework (you must use the Haut HTML parser):

```
html = char *GetWebPage (char *url);
weblinks = char *GetLinksFromWebPage (char *html, char*url);
```

The main goal is to turn the functions you wrote into a web spider using breadth first search. Write a breadth-first search (refer to class slides) robot which inserts the weblinks into a "linked list" (this is "q" from the slides and should have html pages). It is also necessary to put the old links (the ones that have already been downloaded) into a tree (e.g. b-tree) structure. In general, before downloading a link, it is important to check if it has already been downloaded by finding out if it exists in the tree structure (note that hashes are not allowed). Ideally, this should prevent your spider from downloading the same webpage repeatedly. While the web spider is running, it should print each weblink that it is downloading and print which weblinks were found. Start the webspider as

```
./webspider  https://www.universiteitleiden.nl/
```

Have it run for 300 iterations starting at https://www.universiteitleiden.nl/ and take a screenshot of the output.

In Journal.pdf, please show the screenshot and mention what was implemented (e.g. breadth-first search using Haut html parser with a linked list for new links and b-tree for duplicate detection of old links) .

## (3) Google-lite: Inverted Index Search (medium-high difficulty)

Here we create Google-lite - a functional web search engine based loosely on the original Google. *Note that your source code* must compile using *make* and have the functionality described below. Recall that Google created several inverted indices. We will create them next.

### (a) Web Indices

#### Create the following

(note: you should use docIDs for at least the LinkIndex. It is your choice if you use them for all of the URLs in indices such as Titleindex, Pageindex, etc.)
**WebIndex:** For each link that we added to q, parse it for the constituent words and create an inverted index under the directory "**webindex**" where each word is a file and the contents of the file are the weblinks that contain the word. [please see the class notes for more details]

i.e. http://www.liacs.nl/masters-education should be parsed into the following words

www    liacs    nl    masters    education

Note that a handy function for appending text to a file is fprintf:
```
FILE *fp; if(fp = fopen(myword,"a") != 0);fprintf(fp,"%s\n",myurl);fclose(fp);
```
where myword and myurl are strings (must end with '\0') and
"a" means append to end of file. The file is created if it does not exist.

**TitleIndex:** Extract the title information from the webpage. For each link that we added to q, parse the title for the constituent words and create an inverted index under the directory "**titleindex**" where each word is a file and the contents of the file are the weblinks that contain the word.

**LinkIndex:** For each webpage we download, we create an inverted index under the directory "**linkindex**" where each URL in the webpage becomes a file and the contents of the file are the weblinks that point to webpages which contain the URL. Due to file creation problems, the preferred option is to use a docID (convert URL to MD5, CRC64, etc.) for the filename. To be precise, each file contains the weblinks whose webpage contains links that point to the URL corresponding to the filename. (ask me for help if this is confusing)
**Repository**: Save the webpages in the directory "**repository**"

*(optional – this is not necessary)* **PageIndex**: For each html webpage parse it for the constituent words (not html tags) and create an inverted index under the directory "**pageindex**" where each word is a file and the contents of the file are the weblinks that point to the webpage that contain the word.

Insert screenshots into Journal.pdf to show the contents of WebIndex, TitleIndex and LinkIndex.

## (b) Webquery

Google assumes that webpages which have query terms in the title and the URL are the most important. In addition, Google uses the LinkIndex to estimate how popular the webpage and the existence of query terms in the webpage. Create a command line program called **webquery** that takes as input a single keyword query and prints a list of URLS which are relevant to the query.

Create a working web search page (like Google). An example is given in file: **webinterface.mongoose.zip** and introduced in the class slides. Each entry in the results should look roughly like Google:

--------------------------------------------
*query term: Leiden*
Leiden University
**www.universiteitleiden.nl**
--------------------------------------------

If you were able to parse the text of the webpage for PageIndex, then it should look like (extra credit will be given):

--------------------------------------------
*query term: Leiden*
Leiden University
**www.universiteitleiden.nl**
Leiden University, The Netherlands, was founded in 1575 and is one of Europe's leading international research universities. It has seven...
--------------------------------------------

Insert at least 4 screenshots of a query and results into Journal.pdf. Also mention in Journal.pdf what you used to sort the result URLs (e.g. title then URL then LinkIndex (number of pages pointing to the webpage) or something else)

## (c) Image Search

Also, implement text queries for images (like Google images). Use at least the title of the webpage and the "alt" text for indexing. Insert at least 4 screenshots of a query and results into Journal.pdf

## Submission Checklist

*Always check the LML Course Manager for the due date!*
*The program must compile with "make" on Linux and run on one of the machines in room 302/303*

Place in a ZIP file the following and submit on the LML Course Manager. *The top level of the zip file should contain a directory called project.firstname.lastname which contains*:
- The file named "**Journal.pdf**" which should list at the top:
    - Your full name and student ID
    - The name of the machine you had it working on. e.g. 0009747, usually on the side of the machine.
    - Your answers and comments to the problems you solved.
- A directory called "sourcedirectory"
    - The source code and Makefile (the project must compile using "make") and working executables

For example, for a student named Johan Cruijff it should look like:

```
homework3.zip
    project.johan.cruijff
        Journal.pdf
        sourcedirectory
```

*Good luck!*