Sehbau: Overview Software

An overview of the software suite for creating a visual system is given. The suite comprises a set of programs carrying out different recognition phases such as descriptor extraction, descriptor matching, unsupervised segmentation, retrieval and learning. Feature extraction is based on contour and region information and therefore immediately interpretable. The output can be combined with other methodologies such as Deep Networks and Local Features. The overview explains the architecture for feature extraction and the presently available descriptors. And it surveys the individual programs, both implemented and planned.

Repositories `https://github.com/Sehbau`

We firstly introduce the three essential programs, with which one can perform image and region matching (Section 1). Then we explain some basic parameters of the architecture and the descriptors developed so far in *C* (Section 2). We survey the programs in three sections: the core processes, namely descriptor extraction and their matching (Section 3); the learning programs (Section 4); and the utility programs that facilitate matching and segmentation (Section 5). Section 6 provides demonstration programs.

I provide utility scripts in Matlab, as that code is easier to read and maintain than Python (less parentheses and dots), and because it offers a wider range of plotting tools. The notation is explained in Section 7. For more notation see also my Computer Vision overview:
`https://www.researchgate.net/publication/336460083`

1 Essential Functionality

The programs are surveyed first (Section 1.1). Then we give a short introduction of their specific usage (Section 1.2).

1.1 The Essential Trio

The three essential programs are called **dscx**, **mvec** and **focx**:

- **dscx** [descriptor extraction]: takes an image as input and carries out feature extraction and descriptor generation. The program outputs the features and descriptors into a number of different files. The most important one is the *vector* file, with file extension **vec**, also called '**vec** file' sometimes. It contains the descriptor attributes useful for classification. Other files contain the bounding boxes (file extension **BonBox**) and the contour endpoints (extension **EndPts**) for generating object proposals. The file output is useful in combination with other methodologies (Deep Networks, Local Features); or it can be used to perform further integration to generate more description. Some more details will be given in Section 2. The full details are in the documentation at:
  `https://github.com/Sehbau/DescExtraction`

- **mvec** [matching vectors]: matches the descriptors as outputted by the program **dscx** (or **focx**), returning metrics for various types of descriptors.
  `https://github.com/Sehbau/MtchVec`

- **focx** [focus extraction]: extracts descriptor vectors of a desired region, a so-called *focus*, from the vector file as generated by **dscx**. The region is defined by the user as a bounding box and can signify an object (part) proposal or an annotation. The bounding box can be of arbitrary size. An extracted focus can be matched with other focii with the (above) program **mvec**.
  `https://github.com/Sehbau/FocExtr`

The coarse process flow is therefore, as one might have guessed:

dscx → **mvec**
↘
focx → **mvec**

The first deployment of **mvec** (in upper row) can serve to match the entire image with another image, or any other (vector) representation of similar size. Its second deployment (in the second row) serves to match individual regions extracted by **focx** with regions of another image or any other representation. The choice of which region (focus) is to be selected for further processing can be made based on the contour and region output provided by **dscx**, namely the files with extension **BonBox** and **EndPts**.

The use of the vector-matching program **mvec** provides the most accurate form of matching. Faster classification and matching can be obtained by using the histograms as outputted by **dscx** or by **focx**, to be explained in the respective documents. This will allow a robot to orient itself much faster than any other methodology.

The repository named **/DemoPlcRec** provides an example of how to use these programs. In the following we introduce their usage.

1.2 Introduction to Usage

The following explanations introduce the use of the programs in a slightly simplified manner for reason of clarity. The documentation in the respective folders provide the details.

We have two images we wish to compare, named **imgA.jpg** and **imgB.jpg**. Firstly, we generate the descriptors and place them into the vector files called **dscA.vec** and **dscB.vec**:

```
> dscx imgA.jpg dscA.vec
> dscx imgB.jpg dscB.vec
```

In a first round we compare the images at once for which we feed the vector files as arguments to program **mvec**,

```
> mvec dscA.vec dscB.vec
```

which returns dissimilarity and similarity metrics, either as standard out or as file. In a second round, we compare individual regions, for which we now deploy **focx**. We extract such a region (focus) by specifying the bounding box, here the upper left quadrant of the image, `0 128 0 128`, assuming image sizes are both 256x256. We write this subset of descriptors to a file named **focAupplef.vef**:

```
> focx dscA.vec 0 128 0 128 focAupplef.vef
```

Then we match two quadrants as follows, in this case upper left of A versus some region of B:

```
> mvec focAupplef.vef focBsomewhere.vef
```

which again returns the metrics. We integrate the results ad libitum.

The programs allow to take lists as input, ie. one can specify a text file as input that contains file names or bounding boxes. The details of that will be explained in the respective documents.

2 Architecture and Descriptors

The descriptor extraction program **dscx** generates an image pyramid of `nLev` levels, ie. `nLev=5` for a 256 x 256 pixels. We depict this as follows, shown for four levels only for simplicity (ie. 128x128 pixels) using zero-indexing:

```
lev 3          --
lev 2         ----
lev 1       --------
lev 0  ----------------
```

The bottom level of the pyramid, `lev 0`, holds the original image resolution. Higher pyramid levels are generated by downsampling with an integer factor equal two. Downsampling continues until the map is equal 16 pixels; or just larger, for the smaller side length. In the depicted schema, level equal 3 would be the top level.

For each level, its contours are extracted. They are kept as a pyramid depicted now with 'l' as generically representing 'lists':

```
lev 3          ||
lev 2         ||||
lev 1       ||||||||
lev 0  ||||||||||||||||
```

When we subselect contours from a region, ie. using program **focx**, then we extract the corresponding pyramid. For example selecting from a quarter region of the image, the corresponding subset of the list is organized as follows,

```
lev 2          ||
lev 1         ||||
lev 0       ||||||||
```

and starts with level 0 but reaches only level 2 as we extract a subset of the pyramid.

For region segmentation, each level is processed by a hierarchical segmentation, whose tree `depth` is typically set to three for images up to ca. 400 x 500 pixels. The following illustrates the map output for an architecture with four levels and depth equal three:

```
            depth 0           depth 1           depth 2
lev 3         --                --                --
lev 2        ----              ----              ----
lev 1      --------          --------          --------
lev 0  ----------------  ----------------  ----------------
```

The demonstration program **demotree** displays this tree for the original resolution, that is `lev=0`, see Section 6.

Each of those maps is then analyzed for connected components and their boundaries are extracted. The boundaries are concatented across depth (per level), resulting in a pyramid structure as depicted already above for contours. The boundaries themselves will be used for a simple description, called *radial* descriptors. But they are also partitioned into curved segments (arcs) and straighter segments (straighters).

The features are then abstracted by a number of attributes such as geometric and appearance parameters. Thus we have four basic descriptor types:

- Contour (**cnt**): there are three types of contours: ridge (**rdg**), river (**riv**) and edge (**edg**) contours, all obtained by a simple analysis of the intensity topology.

- Radial shape (**rsg**): parameterization based on the radial signature of a region (connected component).

- Arc segment (**arc**): curved boundary segments obtained from partitioning the region boundaries.

- Straighter segment (**str**): straighter boundary segments obtained from partitioning the region boundaries.

Using those basic descriptors we form more complex descriptions, such as groups, with many more attributes. One that is available already are shape abstractions:

- Shape abstraction (**shp**): a description based on the segment statistics of arcs and straighters for each individual shape containing dozens of attributes.

The program **dscx** outputs all lists of descriptor vectors to a **vec** file, as float values. It also outputs them as bins to a **veb** file for building histograms, as integer values. The bin centers are given by specifying the number of desired bins per descriptor attribute.

The programs also output the attributes as histograms in a separate file with extension **.hst/.hsf** (**dscx** and **focx**, resp.). Four types of histograms can be generated:

- flat, univariate: one-dimensional for the entire image
- flat, bivariate: two-dimensional for the entire image
- spatial, univariate: taken from a 3x3 grid by default
- spatial, bivariate: taken from a 3x3 grid by default

Program **dscx** generates all four of them; program **focx** generates only the flat histograms.

3 Core Processes

The core processes carry out feature extraction and descriptor generation, their matching for recognition and motion estimation.

- **dscx**: carries out feature extraction and description as introduced already above and outputs various files that will be loaded by other recognition processes. Currently it takes only one image at a time and it is therefore somewhat slow due to repeated memory allocation for individual calls. Future versions will include processing a list of images for faster extraction.
  https://github.com/Sehbau/DescExtraction

- **mvec**: matches descriptor vectors and outputs dissimilarity and similarity measurements for each pyramidal layer and descriptor. It can be applied to any structure: shape, object or scene.
  https://github.com/Sehbau/MtchVec

- **mshp** [prototyped]: matches (boundary) shapes by correlating their partitioned arc and straighter vectors. This is more specific than the use of **mvec**, where vectors are matched irrespective of their boundary origin (and for a pyramid).

- **motv** [planned]: computes the motion vectors between descriptors of two frames. This is essentially the same as **mvec**, but will output in particular the motion vectors between nearest neighbors. That will allow to estimate motion flow.

- **knnv** [prototyped]: nearest-neighbor search of structures using a coarse-to-fine strategy to accelerate the retrieval process, as opposed to **mvec** above, that matches the entire pyramid between two structures (and is therefore slower). **knnv** starts with a matching of the top (pyramid) layers, then subselects images and progresses toward

finer levels. Early experiments have shown that this strategy not only speeds up the search, but also improves the sorting.

A matching between attribute histograms is not foreseen so far, as for that there exist already many statistical packages, such as `sklearn` in Python.

4 Learning

A learning process will be provided that determines category-characteristic descriptors by individual matching of vectors.

- **kkcan** [planned]: searches for nearest neighbors across images of one category to find candidate descriptors. This is based on **mvec** above, but outputs the results to file.

- **kkgrp** [planned]: refines the search and selects final set of category-characteristic descriptors.

- **kkmtc** [planned]: matches the category-characteristic descriptors against a new image and outputs the degree of dis-/similarity per category-characteristic descriptor.

5 Utility Processes

The following utility processes facilitate matching and improve segmentation.

- **sgrRGB** [beta]: segregates an RGB image for a given target color, resulting in a black-white image with region boundaries that are more precise than with the gray-scale information as used in program **dscx**.
  https://github.com/Sehbau/SgrRGB

- **focx**: selects descriptors from a region (focus), specified by the user as bounding box, and places them into a file with extension **vef**. The **vef** files are loaded by matching programs such as **mvec** and **knnv** and allow matching multiple representations (categories or instances). This functionality is convenient for general scene analysis.
  https://github.com/Sehbau/FocExtr

- **srch** [planned]: allows search of an object by specifying one category that then will be matched against different locations in an image.

6 Demonstration, Varia

The following mentions the demonstration programs available, as well as administrative and utility routines for Matlab:

- **demotree**: a binary that plots the segmentation tree for a gray-scale analysis (as used in **dscx**), as well as for a chromatic analysis using the RGB channels.
  https://github.com/Sehbau/demotree

- /**DemoPlcRec**: contains a Matlab script that carries out a simple place recognition experiment using programs **dscx**, **mvec** and **focx**. It is an elaborate version of the example introduced in Section 1.2.
  https://github.com/Sehbau/DemoPlcRec

- /**UtilMb**: contains utility (Matlab) scripts for running some of the demos.
  https://github.com/Sehbau/UtilMb

7 Notation

The naming of scripts and functions distinguishes between those that carry out specific computations or routines, and those that comprise a set of the former, also called wrapper sometimes. The former start with a single letter and an underscore, f_, i_, u_, etc. The latter start with a word (or syllable).

- **f_**`Func`: routines starting with **f_** compute important functionality, such as feature extraction, feature manipulation, etc. The function name is composed of 'syllables' of three to four letters, aligned from abstract to more detailed.

- **i_**`Func`: routines starting with **i_** initialize an algorithm, process, etc..

- **u_**`Func`: functions starting with **u_** are utility functions carrying out administration, support, etc.

- **p_**`Func`: functions starting with **p_** are plotting routines.

- **Load**`X`: loads from file with path being specified as function argument. Such functions typically call Read routines, as explained below.

- **Save**`X`: saves data to file with path being specified as function argument. Such functions typically call Write routines, as explained below.

- **Read**`X`: reads from file with filepointer given as function argument. They are usually called from a loading script `LoadX` (see above).

- **Write**`X`: writes to file with filepointer given as function argument. They are usually called from a saving script `SaveX` (see above).

- **Plot**`X`: comprises a longer list of plotting instructions, calling usually plotting routines `p_Func`.

- **Renn**`X`: runs a binary/executable from Matlab using the Matlab function `dos`. It is a wrapper routine facilitating the use of the binary with its options, e.g. `RennDscx.m` runs program **dscx**.

- **run**`X`: is a demonstration script showing how to apply a program, e.g. `runDscx.m` runs program **dscx**. These scripts usually call the corresponding wrapper routine `RennX` as an example.

The following notation is used for variables:

`nX`	number of X, e.g. `nLev`
`stcX`	structure of X
`szX`	size of X, e.g. `szV`, `szH`, vertical, horizontal size

For more notations see also my Computer Vision overview:
https://www.researchgate.net/publication/336460083