Sehbau: The Software Suite

C. Rasche
https://github.com/Sehbau

October 7, 2025

This document describes the Sehbau software suite, a computer vision system that operates with parametric contour and region information. The system distinguishes itself from previous approaches by a much faster and richer feature extraction process and by a curve-partitioning procedure that allows to characterize shapes. It uses a divisive segmentation procedure, that returns all region boundaries irrespective of contrast and size. The features are thoroughly parameterized and the resulting description can be used for identification and categorization of any structure; for the description of common textures; for saliency analysis; and for determining motion flow. All those recognition processes are carried out based on the *same* feature extraction output, and *not* with different feature-extraction phases as in other methodologies. This enables to build fast and flexible recognition pipelines. The software suite is the ideal starting point for building an active vision system. The most comprehensive description of the system is available under:

`https://www.researchgate.net/publication/391240551`

Chapter 2 explains the description; it generates 7 descriptor spaces and carries out a texture analysis. Chapter 3 explains the representation formats that have been implemented thereof. The starting point for recognition is the program for *descriptor extraction*, called `dscx`, whose file output is explained in Chapter 4. The remaining chapters explain how to match and deploy the descriptor output and how to further analyze a scene. The entire suite is applied in a mock example for place recognition (Chapter 11). The chapter on applications proposes how to deploy the suite for specific tasks (Chapter 12). Examples of how to administer the programs are given with Matlab and Python. The code notation is explained in Appendix E.3.

The software can be found on

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

for the following systems, all 64 bit (x86):

Windows	`SEHBAU_win.zip`
Ubuntu, 22.04.4 LTS	`SEHBAU_ubu.tar.gz`
Debian	`SEHBAU_deb.tar.gz`
Fedora	`SEHBAU_fed.tar.gz`

This document is also available on:

`https://www.researchgate.net/publication/391238505`

Contents

Chapter 1

Introduction

We firstly overview the content of the software suite (Section 1.1), followed by introducing the principal sequence of programs, with which one can perform image, focus and shape matching (Section 1.2).

1.1 Folder Content, Demo Scripts

The folder /SEHBAU contains the following directories, with blue denoting program binaries, that exist in those directories:

/AdminMb	administrative code using Matlab
/AdminPy	administrative code using Python
/DemoBaum	demo for global-to-local segmentation, baumgrau, baumfarb
/DemoPlcRec	demo for place recognition (plcAll.m/.py)
/DemoSgrRGB	demo for foreground-background segregation, sgrRGB
/Demos	various scripts demonstrating parameter changes
/DescExtr	descriptor extraction, dscx, and
	conversions to vectors: h2arr, collhimg, d2vmx, collvec
/FocSel	focus of attention: selects descriptors from a region, focsel
/MtchHst	matching attribute histograms, mhst, mkolL
/MtchVec	matching descriptor vectors, mvec, motvec
/ShpExtr	shape extraction for a patch (image), shpx
/ShpMtch	shape matching, mshp
exsbAll.m/.py	summary script running all example scripts

The program binaries do not require Matlab or Python to be executed, but explaining how to administer them is best exemplified in those two languages. The demo scripts for both lie often in the same folder, the function routines lie in separate directories named /AdminMb and /AdminPy, respectively. The Matlab scripts are the most elaborate and we therefore use extension .m throughout the documentation to indicate that we talk of an administrative routine or a demo script. Most of the Matlab scripts have also been tranlated to Python. Before running any script one should specify the path where the main folder lies, by setting variable rootSehBau in script globalsSB.m/.py.

The names of the example scripts contain generally the prefix exsb (example Sehbau) and exist in various folders; they must be run from their respective folder as we often use relative paths to the program binaries and example images. The simplest two example scripts are:

`DescExtr/exsbDscxSimp.m (.py)`	descriptor extraction `dscx`
`MtchVec/exsbMatch.m (.py)`	matching with `mvec`

They use the fewest utility routines for reason of clarity. Other scripts utilize wrapper functions to facilitate argument passing.

The script in the main folder, called `exsbAll.m`, runs all example scripts, starting with those two simple demo scripts. The example scripts for place recognition contain the prefix `plc`. The script `globalsSB.m` provides paths and sets some global variables. More explanations on the administrative code are given in Appendix E.

1.2 Survey of Programs and Usage

The principal programs are surveyed first (Section 1.2.1), followed by a short introduction of their usage (Section 1.2.2).

1.2.1 Principal Programs and Pipeline

To launch recognition, we firstly carry out feature extraction and description for the entire image, executed with a program binary called `dscx`, which produces a number of files, called image files here:

`dscx` → `image files` → `mvec`

With some of those files we can carry out classification using a traditional classifier. With other files we can carry out identification by using a program binary called `mvec`: it matches the outputted descriptions of two or more images. This matching process does not require any pretraining or presampling of any kind. It is a pattern matching process that can be applied to any image of arbitrary content and dimension.

This combination of programs alone is already a powerful content analyzer, and we could simply move the camera direction to different spots in our visual environment and repeat the recognition process. Or we zoom and repeat the process. But before we move the camera direction, we also want to maximally exploit the description in the image files by starting to focus on certain parts of it. For that purpose two processes exist, called *focus selection* and *shape description*. These processes can be regarded as attentional shifts.

Focus Selection The process of focus selection extracts a subset of the outputted description (from the image files) and saves it again as separate files called focus files. The choice of what to select can be based on the output from the image files. There is no further computation carried out, the selection merely rearranges a subset of the output and saves it to the focus files. The selection is carried out with a binary called `focsel`:

`image files` → `focsel` → `focus files` → `mvec`

The focus files can be matched again with program `mvec`. This 'focusing' allows to classify or identify arbitrary (rectangular) parts of an image separately, leading to better hypotheses.

Shape Description Unlike focus selection, the process of shape description carries out more computation. It extracts an image patch given a specified bounding box, and then performs a simple color segmentation given a specified color cue. The resulting region boundary is then analyzed structurally and that description is saved to a shape file:

`image files` → `shpx` → `shape file` → `mshp`

This separate shape processing is more accurate than the shape processing that takes place in binary `dscx`, because we utilize a color cue, obtained for example from the image files. Shape files can be matched with program `mshp`.

We elaborate on the individual programs and processes introduced so far:

- `dscx` [descriptor extraction]: the binary outputs the features and descriptors into a number of different files of which the three main ones are:
  - the *description* file with extension `.dsc`, containing the descriptor attributes with which one can span a multi-dimensional space, useful for identification.
  - the *histogram* file with extension `.hst`, expressing the attributes as histograms, useful for fast classification with a 'traditional' classifier (Linear Discriminant Analysis [LDA], SVM, RandomForest [RF], etc.); or to build a cascade classifier.
  - the *saliency* file with extension `.slc`, containing scene statistics, some object proposals and texture information (Section 2.5). This information can be used to decide where to apply focus selection and shape description. And it can be used for visual orienting, for example for auto-focusing, zooming, performing a saccade (change of camera direction), etc.

- `mvec` [matching vectors]: matches the descriptors as outputted by the program `dscx` or `focsel`, and returns metric measurements for various types of descriptors. This is useful for identification of structure. The use of this binary will be explained in Chapter 5.

- `focsel` [focus selection]: extracts the description of a desired region, a so-called *focus*, from the description file as generated by `dscx`. The region is defined by the user as a bounding box and can outline an object proposal or part proposal, ie. obtained from the saliency file; or it can be an annotation. The bounding box can be of arbitrary dimension and size. `focsel` extracts both, vectors and histograms, extensions `.dsf` and `.hsf`, resp. The vectors can be matched with other focii with the program `mvec`. To be further detailed in Chapter 7.

- `shpx`/`mshp` [shape extraction and matching]: refines the segmentation of a shape and saves it to file with extension `.shp`, which then can be used for matching (Chapters 8 and 9). This shape can be any silhouette, be it the letter of some text in the wild, or an object with homogenous color, or scene part.

Refining the Pipelines With that program survey, we can now refine the above pipelines. Since the deployment of vector matching with binary `mvec` is a relatively costly process, it makes sense to preselect candidates by firstly classifying the histogram output, process *Clsf*, and then to apply `mvec` on the identified subset of representations:

`dscx` → *Clsf*(`.hst`) → `mvec`(`.dsc`)

This cascade classifier can be carried out with a trained classifier (LDA, SVM, RF, etc.), which makes sense if we assume a clearly defined category. Or it can be based on histogram matching only, if the goal is to identify a structure, in which case the term *cascade identifier* is more appropriate. For the former there exists enough software; for the latter we provide a separate program called `mhst` (Chapter 6).

Focus selection can be based on the output provided by the saliency file, or any output of `dscx`. For matching, we can again make a selection based on classification:

`focsel`(`.slc` | `.dsc`) → *Clsf*(`.hsf`) → `mvec`(`.dsf`)

Focus selection allows applying tailored representations, without performing a complete feature extraction and description. It is useful in particular, if we wish to analyze structure containing contour information or texture. If the goal is to analyze rather a shape silhouette in more detail, then we apply shape description and matching:

`shpx`(`.slc`) → `mshp`

Or one can apply both processes to the same region, patch respectively.

1.2.2 Introduction to Usage

The following examples give an idea of how to provide the arguments to the program binaries.

The task is to compare two images named `imgA.jpg` and `imgB.jpg`. Firstly, we generate the descriptors and provide a filename as output, in this example using the single letters `A` and `B`:

```
> dscx imgA.jpg Desc/A
> dscx imgB.jpg Desc/B
```

This will write the description files called `A.dsc` and `B.dsc` to directory `/Desc`. In a first round we compare the images as a whole, for which we feed the description files as arguments to program `mvec`,

```
> mvec Desc/A.dsc Desc/B.dsc
```

which returns dissimilarity and similarity metrics, either as standard output (`stdout`) or as file.

In a second round, we compare two different regions, for which we now deploy `focsel`. We select the upper left quadrant as bounding box, `0 128 0 128` (assuming image sizes are both 256x256):

```
> focsel dscA.dsc 0 128 0 128 focAupplef
```

This will write the subset of descriptors to file `focAupplef.dsf`. We extract an equally sized region from image B and call it `focBsomewhere` (operation not formulated here). Then we match those two regions:

```
> mvec focAupplef.dsf focBsomewhere.dsf
```

which again returns the metric measurements. We integrate the results ad libitum.

Parameters Parameters can be provided by long option or by file. Their naming is occasionally different. Long options are denoted in violet, ie. `--prm`. Parameters provided by file are denoted in red, ie. `prm`.

Chapter 2

Structural Description

The structural description is based on contours and regions. The extraction of those is called *feature extraction*, resulting in lists of contours and regions (Section 2.2). Those features are then partitioned, parameterized and integrated, which is referred to as *feature description*. The output of feature description consists of so-called *descriptors*, explained in Section 2.3.

$$\texttt{pixels} \xmapsto[extraction]{feature} \texttt{contours \& regions} \xmapsto[description]{feature} \texttt{descriptors}$$

The outputted description is quite rich and can be deployed in various ways, ie. selected and interpreted according to the specific task (Section 3). And it can be used to describe scene textures (Section 2.4) and naturally is suitable for saliency and proposals (Sections 2.5). This entire extraction and description process is termed *descriptor extraction*, hence the program name `dscx`. Program `dscx` outputs the descriptors and texture blobs, but also bare region information. We now firstly explain what architectures are available to extract the features from.

2.1 Architecture

Contour and region features are extracted from an image space `IS`. A space consists of a stack of maps, also called *levels* here. Its height is specified as number of levels with parameter `nLev`. Two types of architecture are available: a pyramid space and a (cubic) scale space. We firstly explain the architecture and parameters for the pyramid, then the ones for the scale space.

2.1.1 Pyramid

The following schematic shows a pyramid made of four levels, `nLev=4`, 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 and would correspond to the pyramid of a 128x128 pixel image. The number of levels can be set as command argument using double dash to specify a long option (`--nLev`):

```
> dscx imgA.jpg /dscA --nLev 2
```

The parameter can also be set by file, the details of that follow in later sections. For each level of the pyramid, contours and regions are extracted, which results in a vast set of features. For efficient recognition, some sort of selection must take place. We firstly introduce feature extraction and its selection parameters (Section 2.2), followed by the descriptors developed and their selection procedures (Section 2.3).

The pyramid architecture is suitable for the fast analysis of arbitrary image content. It is the default architecture. The maximum allowable number of levels is 10.

2.1.2 Scale Space

A scale space is better suited when subtle differences need to be discriminated. We can set the architecture to be a scale space by long option `--is`:

```
> dscx imgA.jpg /dscA --is 2
```

and specifying a value of two; a value equal one specifies the pyramid. The parameter can also be specified by file with string `imgSpc`, mentioned again under Section 4.3.1.

A scale space takes a little bit longer to compute than a pyramid, as no reduction in space occurs. And it generates more features for the same reason. The default height is five levels, the maximum allowable height is 10. For large images, specifying many levels may quickly lead to memory limitations (see Appendix D.2).

An example script comparing the output of the two spaces is given in `exsbImgSpaces.m` in directory `/demos`. The image space `IS` is only saved if long option `--saveIsp` is set. It is then written to file with extension `.isp`.

The plotting scripts mentioned in later sections, are setup to demonstrate the output for the pyramid, but can be easily modified for the output of a scale space.

2.1.3 General

There exist more parameters that can be considered part of the architecture, such as the dimensionality for spatial histogramming, and the window size for texture formation. Those will be mentioned later.

There are no particular image preprocessing algorithms carried out before generation of the image space `IS`. The descriptor extraction output is the same for the original resolution in either image space, the pyramid or the scale space. Their output starts to differ from the second level on, from `lev = 1,2,..nLev-1` (zero-indexing).

If one thinks that image filtering might be of profit, one can try the filter options provided with parameter `imgFlt`, see Appendix B for details.

2.2 Feature Extraction

Features are segments of pixels. Two types are extracted, contour segments and region segments. Contour segments are obtained from an analysis of the topological landscape along its ridges, rivers and steep slopes, the latter generally called edges. Region segments are obtained from thresholding the intensity distribution.

2.2.1 Contours

Three types of contours are extracted, namely ridge, river and edge contours, sometimes abbreviated as `RRE`. They are kept separately initially and we depict that as three separate pyramids. We use symbols . and | to symbolically express that the pyramid content is individual, meaning the levels have different list lengths with different segments lengths (no actual correspondence is depicted with those symbols):

```
              ridges                rivers                edges

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

The threshold parameter for accepting a pixel value as a contour pixel, is set with `Cnt.minCtr` by file or `--cntMinCtr` by long option. By default the value is 0.05 and is relative to the maximum value of the range image (taken with a 3x3 neighborhood). Directory /`DescExtr/Examples` contains an example script for the detection of these three types of pixels, called `e_CntMap.m`.

2.2.2 Regions

Regions are detected by a hierarchical thresholding process, that returns essentially all regions, resulting in a vast output. The depth of this hierachical (tree) output is typically set to value equal three for image sizes up to ca. 100k pixels, ie. `depth=3` in our code.

The thresholding process is applied to each level of the architecture. The following illustrates the map output for an architecture with four levels and depth equal three, using again zero-indexing:

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

There are two ways to observe this segmentation output. One is by saving the region pixels to file using the long option `--saveRpx` that generates a file with extension `regPix`. The regions can be loaded by routine `LoadRegPix.m`, an example is shown in script `exsbPlotRegPix.m`. More explanations can be found in Appendix C.

Another way to observe the output is by use of the program `baumgray` in folder /`DemoBaum`. Here the tree output is generated for the original resolution only, that is `lev=0`, see Section A.2.

For all regions of those 12 maps, their boundaries are extracted:

`regions` $\longmapsto$ `boundaries`

The boundaries are concatenated across depth (per level), resulting in a space-like formation as depicted already above for contours. The boundary pixels can be saved to a file with extension `BonPix` (Section C.4). If one wanted to recreate the full tree, one had to separate the different depths by use of the depth index (Section C.4).

Sensitivity and Selection As pointed out, this divisive region segmentation process is capable of returning regions of lowest contrast possible; it finds any nuance in gray-shade or color. This sensitivity results in a vast feature output that is useful for search scenarios, such as defect detection of material surfaces, anomaly detection, detection of occluded objects or of low-contrast objects, etc.

But for most recognition tasks it requires some sort of selection, based on parameters such as contrast, minimum size, minimum spacing, etc. This selection is better performed when we start parameterizing the boundaries. We therefore do not carry out a selection by contrast in this divisive process, and since we also wish to be able to understand low-contrast scenes, such as underwater scenes, night scenes, foggy scenes, etc.

For images larger than $100k$ pixels, a depth of equal four could also be beneficial. A case where depth equal five is beneficial we have never observed; it is nonetheless the maximum allowable depth possible.

It one is interested only in the large regions of an image, then one can control the proliferation of small regions by parameter `Reg.minPixNode` (or `--regMinPixNode` by long option), more details in Section 4.3.3. This can slightly accelerate the region detection process. Their numerosity is however largely irrelevant in many tasks, as we typically select global features (descriptors) for recognition of structure.

2.3 Descriptors

A parameterized feature is called *descriptor* and its measured parameters are called *attributes*. We first give an overview of this formation process, and then introduce the individual descriptor types.

2.3.1 Overview Formation

The features are parameterized in two principal ways. One way is to parameterize them directly without any further partitioning, called direct parameterization. Another way is to partition the boundaries and then parameterizing the resulting partitions. This results in four basic descriptors from which we then form more complex descriptors and a texture description.

Basic Descriptors

The contour features, short-noted `RRE features`, are directly parameterized resulting in the so-called `RRE` set:

$$\texttt{RRE features} \xmapsto[paramet.]{direct} \texttt{RRE set (full)} \xmapsto{skeletonization} \texttt{skeleton}$$

The RRE set is also called the *full* set. It is then reduced to a skeleton set that is relatively robust to luminance variations.

Boundaries are described in two ways. The first, a direct parameterization, is based on a simple radial analysis, resulting in the so-called radial-shape descriptor or sometimes just radial descriptor:

$$\texttt{boundary} \xmapsto[paramet.]{direct} \texttt{radial-shape (full)}$$

which is also considered a full set. There exists no further reduction. The second boundary description carries out a space-analysis, that partitions the a boundary into curved and straighter segments, called *arcs* and *straighters*, respectively:

$$\texttt{boundary} \xmapsto[partit.]{curve} \texttt{arc \& straighter (full)} \xmapsto[nization]{skeleto-} \texttt{arc \& str (gerust)}$$

The result is also called the *full* set and is then reduced to a skeleton called *gerust* (scaffold) here.

Entry Conditions Unlike for contour features, not all boundaries are parameterized. There exist entry conditions that are necessary due to the vast output of the divisive segmentation. The conditions are based on contrast and segment size and exist for admitting boundaries to the processes of direct parameterization and curve partitioning. That is, the full sets for radial-shape, arcs and straighters, are strictly speaking already a reduced set, but are nevertheless called as such, because it still represents a vast amount of information. They are saved to file only if a flag is set. It is only the full set for the radial descriptors that is saved to the `dsc` file, as it is of reasonable size.

The default values for entry conditions and skeletonization parameters are set such, that one can perform decent classification and identification of the images as appearing in image collections with daily scenes. For a specific task or a specialized image collection with unusual characteristics, one might have to adjust some of the parameters, which will be discussed throughout the explanations and be subject when discussing applications (Chapter 12).

These four basic descriptors alone provide relatively good categorization and identification performance for structure already. But for interpreting our environment more efficiently, we form more complex descriptions.

Complex Descriptors and Texture Description

The RRE set is used to generate two types of descriptions, a texture description, as well as a group descriptor, called bundle:

$$\texttt{RRE set} \xmapsto{analysis} \texttt{texture, bundle}$$

The arc and straighter partitions of a shape are used to create a shape description based on their statistics, also called *partitioned-shape* descriptor. The partitioned-shape description in turn is used to obtain a geometrically more precise description called tetragons, if the shape shows certain qualities.

$$\texttt{arc \& straighter} \xmapsto[analysis]{statistical} \texttt{partitioned shape} \xmapsto[analysis]{geometric} \texttt{tetragon}$$

2.3.2 Descriptor Types

We summarize and explain some of the characteristics of the various descriptor types and mention some key parameters. The first four are the basic descriptor types, the remaining are the complex descriptors. The full set of attributes and parameters will be provided in later sections.

1) Contour (`cnt`)

This descriptor expresses ridge, river and edge contours (RRE). An individual contour is described by its length and angular orientation. As introduced above already, the set of all three contour types constitute the full set, the RRE set. Based on it, we derive three types of descriptions:

- Skeleton: a selected set of longer contours, called *skeleton* sometimes. They are drawn from the RRE set and therefore have exactly the same attributes as the RRE set. The skeleton is saved to the description file and appears as `ACNT` when loading its descriptor space.
- Bundle: groups of contours, to be further introduced below.
- Texture: a texture analysis based on the orientation angle and length of segments, to be introduced in Section 2.4.

As said above, the RRE set is *not* saved by default, due to their size. It can be saved by turning on long option `--saveRRE`, in which case it is saved to a separate file with extension `dscRRE`.

The reduction from the RRE set to the skeleton set occurs by a global-to-local selection procedure, whose parameters are minimum spacing and minimum contour length, see `sklMinSpc` and `sklMinLen` of Section 4.3.2.

2) Radial Shape (`rsg`)

The entry conditions for description are based on minimum contrast and minimum size, see `rsgMinPix` and `rsgMinCtr` of Section 4.3.4. The attributes are derived from the radial signature of the boundary. The most significant ones are mean radius, elongation and degree of concavity. More attributes to be mentioned below.

3) Arc (`arc`)

The entry conditions for partitioning are also based on contrast and size. The significant arc attributes are degree of kurvature, size and directional angle. The full set of arc segments is reduced to a skeleton called here *gerüst* (scaffold). Only this gerust subset is saved to the `dsc` file. The full set can be saved as well, by turning on long option `--saveCVP`; it is then written to a separate file with extension `dscCVP`.

4) Straighter (`str`)

The entry conditions are the same as for arc description. Straighter segments lie between the arc segments in a shape and are described by their length, orientation and degree of straightness. They are called in comparitive degree, because they appear straight in context - they are not necessarily fully straight. For example, the two longer sides of an oval are straighter, in comparison to their curved ends.

Analogous to arcs, only the gerust set is saved to the `dsc` file. The full set is saved together with the full set of arcs to the `dscCVP` file (if long option `--saveCVP` is set).

5) Partitioned Shape (`shp`)

This is the third description of boundaries, and is based on the segment statistics of arcs and straighters for each individual shape. It contains dozens of attributes.

By default, this shape description takes place only for regions that are fully inside the image, that is not touching any image border. There exist scenes that lack any inside shapes, such as landscape scenes, photos of smooth surfaces, etc. In that case, it might be worthwhile including shapes that touch the image border. This can be regulated with parameter `bordTouches`. By default it is set to 0 and ignores any regions, that touch an image border. With value equal 1, a shape can touch one image side; with value equal 2, two sides; with value equal 3 three sides; and with value equal 4 all four sides. The more image sides a shape touches, the more likely it represents a background region; or an object very close to the camera. More information about this descriptor can be found on:

https://www.researchgate.net/publication/383039072

6) Tetragon (`ttg`)

This descriptor focuses on shapes that contain at least two loosely parallel straighter segments, whose sides appear to form a tetragon that is loosely aligned with either vertical or horizontal image axis. The idea is to describe in particular horizontal and vertical structures that are ubiquitous in scenes and we therefore determine the axis of such a tetragon. The tetragon is a subset of the partitioned-shape descriptors (`shp`), but with a more refined parameterization. More information can be found on:

https://www.researchgate.net/publication/391670287

7) Bundle (`bnd`)

The bundle descriptor is based on groups of contours obtained from the RRE set. Groups are detected during the process of identifying the skeleton set, and they therefore show the minimum length specified by `sklMinLen`. Groups of contours, that are shorter than that minimum length, are better expressed with the texture analysis.

General Attributes / Bins

Each descriptor type comes with position and angular attributes. The position is often the (normalized) coordinates of the center pixel of the feature, or it is calculated from some keypoints. The angle attribute describes the orientation of the feature in the image plane.

The attribute values are binned for the purpose of histogramming. The bins are saved separately in a file with extension `dsb`.

More explanations on the attributes will follow when introducing the individual programs.

2.3.3 Attribute and Descriptor Space

With the attributes of a descriptor type we span a multi-dimensional *attribute space* `D`. For example for the contour and straighter descriptor we can span a three-dimensional space with its attributes length, orientation and straightness:

```
D( len , ori , str  )
```

A *descriptor instance* is then represented by a point in space. By developing appropriate metrics, we can exploit this space for identification of structure. We can also form histograms from those attribute values, which is an abstraction suitable for fast categorization. This will be further elaborated later (Chapter 3).

The description is carried out for each level of the image space `IS`. For one descriptor type, this results in a list of attribute spaces, which we also refer to as the *descriptor image space* `DS`, or in short *descriptor space*:

$$\texttt{DS} \;=\; \texttt{D}^{\texttt{l}} \quad \texttt{for} \quad \texttt{l = 0,..,nLev-1}$$

The program `dscx` outputs the descriptor space for each descriptor type to the description file with extension `dsc`. This is summarized in Section 4.2.1.

2.4 Texture

The texture description is based on the full set of contour descriptors (the RRE set) of the first level of its descriptor space (`lev=0`). The description occurs in several steps. In a first step we observe the statistics for the segments lying within a window. One type of statistics are histograms of the contour's attributes, called *kolumns*. Then we analyze the statistics in such kolumn histograms, thereof forming a variety of texture maps. Those maps are then described globally by some simple statistics, and also by region, identifying texture *blobs*.

$$\texttt{RRE set} \xmapsto{histograms} \texttt{kolumns} \xmapsto[analysis]{statistical} \texttt{texture maps} \xmapsto[analysis]{component} \texttt{blobs}$$

The kolumns and texture maps are saved to individual files; the statistics of the texture maps and their blobs to the saliency file (`.slc`) introduced previously.

2.4.1 Kolumns

Kolumns are formed similar to the technique of spatial histogramming of features, as applied in the local feature approach. For a small rectangular window of the image, ie. 16x16 pixels, a 4-bin histogram of the contours' orientation angles is formed, called a kolumn histogram, or simply kolumn. We also form kolumns with the length attribute. The two types are referred to as ori-kolumns and length-kolumns, respectively.

The windows overlap by half their window side. For example for a 256x256 pixel image we choose a 16x16 pixel window: the overlap is then 8 pixels, and the resulting spatial dimensionality measures 32x32 kolumns. Kolumns are saved to a file with extension `.kol`, if flag `--saveKol` is set. By default the window measures 16x16 pixels; its size can be changed by long option `--txws` (Section 4.3.7).

The deployment of the kolumns will be discussed alongside the use of histograms of descriptors, ie. when introducing the representation formats (Section 3) or the file output (Section 4.2.2).

2.4.2 Texture Maps

Each kolumn is analyzed for its statistics and the derived parameteric values are placed back into separate maps of size 32x32 (256x256 image; window size equal 16 pixels). The simplest two measures are the count of contour segments and the lack of any segments in the window, abbreviated `num` and `blk`. When contours are present (in a window), then five orientation parameters are determined, that we also call texture *biases*:

- `Num`: total number of contours present in the window, also called the numerosity bias.
- `Blk`: a window is considered blank (void), if three or fewer contour segments are present. This allows detecting sky region, water surface (with no reflection), or any region void of texture, even if there is an 'errand' streak present.
- `Nil`: degree of 'no' orientation (nil dominance). Often corresponds to foliage.
- `Vrt`: degree of vertical orientation (dominance). Often corresponds to texture in natural scenes, ie. grass, crops, stems, etc.

- `Hor`: horizontal orientation dominance, for example uneven water surface; objects arranged with increasing (spatial) distance, such as a column of cars parked along a street.
- `Axi`: axial orientation dominance, that is vertical and horizontal (approximately) equally present, ie. windows in urban scenes.
- `Uni`: single (uniform) orientation dominance. This includes vertical and horizontal dominance, but excludes axial dominance (as there are two orientations present). Can occur at any angle. Rather rare in regular scenes, but potentially useful for textures with diagonal dominance.

So far only the ori-kolumns have been used for a parametric description. Other maps are generated using the contrast and chromatic attributes; and combinations of maps are generated too. They are all written to a file with extension `.txm`, if flag `--saveTxm` is set.

2.4.3 Global Statistics and Blobs

A number of the texture maps is analyzed for its global statistics and the presence of individual regions. The global statistics appear as data structure `Gst` in the saliency file and are useful as a triage in the structural analysis of a scene.

A blob is a connected component in a texture map. For a texture map, we find the connected components and determine their bounding boxes in the original image resolution. Since the texture map is of coarse resolution, ie. 32x32 pixels only, the corresponding bounding boxes in the original resolution are approximate only. The information for the blobs appears as data structure `Blb` in the saliency file. Blobs often correspond to objects and scene parts.

2.5 Saliency and Proposals

Saliency is here understood as the structural information that allows to decide what to analyse next. It is not a specific description, nor is there a particular algorithm that generates some measure of saliency. It is rather a statistical summary of the outputted description which allows to choose what is of potential interest.

For example, if the image is full of texture, then we are interested in finding local variations by auto-correlating the kolumns. Conversely, if the image lacks any texture blobs, then any long contour in the pyramid of skeletons `ACNT` or bundles `ABNDG` is of potential interest.

On the other hand, if the image is full of texture, but contains a single, long contour at a lower level of the image space, then it is exactly that long contour that might be of interest. And in an image full of large shapes, the presence of an isolated blob might be the interesting structure.

In order to make those decisions, we need a summary of the structural description, which is saved to the saliency file (`slc`) in our system.

The saliency file holds in particular information on texture and on selected shapes (Section 4.2.4), that can be considered proposals. It does not hold any specific proposals from long contours or their groups; it provides only some attribute statistics.

For a specific task, one would rather design an appropriate saliency calculating procedure. For example if we search for small objects, one would immediately focus on the texture analysis, see also Section 12.3.

2.5.1 Proposals

As explained above, some proposals have made it into the saliency file. Better proposals can be obtained through an analysis of persistency in the descriptor space `DS` (Section 2.3.3). Such proposals are more likely to correspond to an object or scene part than the presence of an isolated descriptor in space.

More specifically, if one descriptor type appears in the same image location in two or more adjacent levels of `DS`, then it is considered a proposal. For example if we find a partitioned-shape descriptor in level 0 and another in level 1 at the same or nearby location, then that is taken as a proposal. Such a persistent proposal can be a foreground object or a background scene part. It can also be a specular reflection, that often appears persistent in the image space as well. Discriminating between structural proposals and persistent reflections would be a next step.

Finding such persistent descriptors is a first step toward the creation of a single map, that is suitable for planning actions. The output of the texture blob analysis (introduced previously) can also be considered part of this first step, but here we focus on the segment descriptors, in particular partitioned shapes and tetragons.

The proposals of such shapes and tetragons, and their bounding boxes, are saved to files with extensions `qdsc` and `qbbx`, more details in Section 4.2.5.

Chapter 3

Representation Formats

The strength of the structural description is its versatility: it allows the representation of both instance and category, as well as forms in between; it can represent the spectrum from abstract to accurate and is therefore suitable for building a cascade classifier.

The most straightforward format to represent structure is to deploy histograms of attribute values. This was carried out already for kolumn formation (Section 2.4.1). Subsequently we introduce a more elaborate version of histogramming (Section 3.1). Histograms are suitable for traditional classifiers (LDA, SVM, RF) and allow for fast classification of abstract categories and they can serve to preselect candidates for more specific matching in a cascade classifier. Or histograms are merely matched, a situation more appropriate if the recognition goal (of the cascade) is identification.

Another relatively straightforward representation format is the use of the attributes in a multi-dimensional space, the attribute space as introduced above already. In that format, a descriptor instance is represented as a point in space and we develop metrics to compare or discriminate different points. This is the most precise format and is therefore suitable for identification of structure (Section 3.2).

Those two formats cover the two ends of the representation spectrum very well, yet sub-ordinate categories are not well captured yet, explained in Section 3.3. To represent those, there are other formats one can think of, in particular with the description of texture blobs and in connection with the associative capabilities of Deep Networks, which will be mentioned last (Section 3.4).

3.1 Histogram of Attributes

Histograms can be formed for the entire image or for rectangular regions. For the kolumns, they were formed for an array of very small regions, called window there. We firstly recapitulate the kolumn histograms, then proceed to introduce the histograms for image and focus.

3.1.1 Kolumns

The kolumn histograms were formed with only the RRE contours (Section 2.4.1) since the goal was texture description. For a window size of 16x16 pixels one might consider also histogramming the attributes of the radial-shape descriptor. For smaller sizes it is difficult to obtain significantly more specificity as already given be the RRE set.

For a 16x16 window size we obtain an array of 32x32 kolumns for a 256x256 pixel image. Multiplied with the number of bins for orientation (4) and length (10) we arrive at a total dimensionality of 14336 bins. This representation format depends on the image dimensions.

Kolumns can be matched with a program binary named `mkolL`. Kolumns can be regarded as a fine version of spatial histogramming with a small set of attributes. When we later form histograms for the image or regions, coming up next, we make use of the full set of descriptors and their attributes.

Performance The kolumns provide decent accuracy in scene identification for limited environments, that is they are potentially useful for preselection in a cascade identifier. Their use in scene categorization has not been tested yet; it probably makes most sense to concatenate them with the image histogram.

3.1.2 Image Histogram

When we form histograms for the entire image, we take all attributes of all descriptor types. We generically refer to those individual attribute histograms as the *image histogram*. For matching with program `mhst`, the attribute histograms are matched separately and the distances integrated to an ensemble distance. For classification one can concatenate them to a single histogram.

The attribute histograms are generated with 5 to 12 bins each, resulting in univariate distributions. Bivariate distributions are also formed with a subset of pairs of attributes. A few trivariate histograms are generated as well, that are assigned to bivariate distributions. These types are also refered to as *flat* histograms since they are taken for all descriptor instances across the entire image.

To exploit the position of the individual descriptor instances, spatial histograms are formed as well, similar to the kolumns but coarser in array size. Spatial histograms are formed from a grid of non-overlapping cells, where the grid size measures 3x3 by default. For each cell, both the univariate and the bivariate distributions are formed. Thus, in total four types of histograms are generated:

flat, univariate	one-dimensional
flat, bivariate	two- and three-dimensional
spatial, univariate	one-dimensional, taken from a `m` x `n` grid
spatial, bivariate	two-dimensional, taken from a `m` x `n` grid

Program `dscx` generates all four of them and they are saved in a separate file with extension `.hst`. The total dimensionality of the image histogram is currently at ca. 24*k* for a 3x3 grid. The image histogram is generated with the full set of descriptors, the RRE set and the full set of curve partitions (CVP). For contours, we therefore have four sets of histograms: ridge, river, edge and skeleton.

The flat histograms are *not* dependent on the image size, as they are formed for the entire image. They have a fixed dimensionality and can therefore be compared irrespective of the size of their image source. The size of the spatial histograms however depend on the grid size. They can only be matched across images, if they are formed with same size.

Histograms can be matched with the program mhst (Chapter 6). The histograms can also be collected with program binary collhimg to form a matrix suitable for training a traditional classifier.

The histograms that we provide can be considered generic. For specific categories, different bin counts and edges for attributes can make quite some difference in classification, as well as the choice of generated bivariate histograms and the dimensions of spatial histogramming. To explore such parameter variations we offer only a few options so far. But one can easily develop one's own histogram variations, using the vector matrices as outputted by programs collvec and d2vmx.

Performance Training a Random Forest classifier often provides good results for abstract categories, easily in the range of Deep Networks that were optimized for ressources (and typically have lower recognition accuracies than full Deep Networks). A Restricted Boltzman Machine (RBM) can perform also well. The careful development of an ensemble classifier has quite some potential to improve the classification accuracy.

We recommend starting classification with the first three types of histograms: flat univariate, flat bivariate and spatial univariate. The fourth type, the spatial bivariate histograms, did not consistently improve classification accuracy. It is perhaps best to apply feature-selection schemes.

3.1.3 Focus Histogram

To form histograms of attributes for a rectangular region, we deploy program binary fochst1. They are formed with the attributes of the first four descriptors: contour skeleton, radial shape, arc and straighter. Univariate and bivariate distributions are formed as was done for the image histogram. The total histogram is called *focus histogram*. It is naturally flat as the purpose of forming a focus histogram is to select a subset.

The focus histogram is saved to a file with extension .hsf. Its total dimensionality is currently at 1784 bins. Including more histograms from other descriptors, ie. from partitioned shapes, is certainly worth testing.

The focus histogram is also independent on the input size. Focus histograms can therefore be compared irrespective of the size of the region they were formed from. They can be matched with binary mhst, the same program as for matching image histogram. But they cannot be matched to an image histogram due to their different dimensionality. They could however be matched to the flat histograms of the image histogram in principle, but it is more straightforward to form a separate focus histogram that is taken from the entire image.

3.2 Vector Based

The vector matching programs mvec and mshp calculate a dissimilarity and a similarity measure. For both, the metric uses (in most cases) individual weights for the attributes, that is one weight per attribute for all descriptor instances (not for individual instances). The metric includes the difference in position and angle by default, which makes the representation rather rigid, and we can think of it then as a *rigid vector template*. It is useful for identification of structure, whose pose does not change significantly, such as in place recognition, or motion detection.

Program `mvec` matches the entire descriptor space `DS` provided. The measurement values are returned per descriptor type. That allows to form ensembles of any kind. The size of the spaces must match, specifically they must have the same height (`nLev`) and therefore will derive from similarly sized images (or focii). For program `mshp` there is no space involved; the shape can come from any input size.

Since structure sometimes appears with quite some variability, one can loosen the template by lowering certain weight values in order to accomodate the variability, in which case the vector template becomes a bit wobbly; that however increases the probability of perceptual aliasing. If we turn the weight values off for position (to value equal zero), then the representation is rather a statistical one, namely a set of structural elements of certain orientation. If we turn off the weight values for angles, then that reduces the representation to a mere set of structural elements. The matching programs allow to control those weight parameters.

Ideally, the metric would use individual attribute weight values for each descriptor instance in order to express category-typical variability. This has not been implemented yet and the rigid template therefore shows limited capability to express nuances in categories, such as the subordinate categories of the Indoor-67 collection with its 67 classes of room interiors.

The proximity metric favors equal list length. Program `mvec` is therefore less suitable for detecting a specific pattern appearing in varying contexts, or a silhouette containing varying texture. For that purpose one would utilize the processes of focus selection and shape matching (`focsel` and `mshp1`).

3.3 Subordinate Fuzziness

As mentioned above, the current representation formats and their matching programs do not express subordinate categories very well, such as the Indoor-67 collection. Whether a more flexible vector-matching scheme will be sufficient to eliminate the fuzziness, needs to be tested. Since however identification works so well, it is worthwile pursuing a Nearest-Neighbor analysis with the goal to connect category instances in representation space. A first step into that direction would be to correlate all instances of a category, which can be done with routine `MVECLXLfull.m` (`/AdminMb/MtchVec/`), see example script `plcCorrSelf.m`. Using the dissimilarity matrix we would identify clusters that we take as the representatives in a NN approach.

Another possibility to sharpen subordinate categorization is to combine the result with a Convolutional Neural Network (CNN), or perhaps some other network, that effortlessly learns subordinate categories. One possibility would be to form an ensemble and then train the network according to the confusions of the classification results for our system. More on methodological fusion in Section 12.4.

3.4 Other Formats

We now mention other potential representation formats. One is based on exploiting the texture maps, in particular the individual regions (Section 3.4.1). Another is based on clustering the descriptors akin to 'words' in the local feature approach (Section 3.4.2). Another uses a modification of adaptive boosting (Section 3.4.3).

3.4.1 Texture Maps

Here we would exploit the texture maps as introduced in Section 4.2.3. One could match those maps as a whole, akin to kolumn matching, which would be faster but probably less accurate than kolumn matching. Or we extract the individual regions of the 7 orientation maps in data structure `OTX` (vertical, horizontal, ...) and develop a matching scheme comparing two lists of regions. The regions themselves could be described by a simple description such as the radial-shape descriptor. This type of matching might be equally suitable for identification or for preselection.

3.4.2 Syllables (Words)

In this format, we build 'words' analogous to the ones as used by the Local Feature approach. In the Local Feature approach these words are typically determined from the entire training set, spanning all categories. That generic word formation is not specific enough in our structural description, as the attribute space of a descriptor type is of rather small dimensionality in comparison to the high-dimensional histogram of image gradients. Instead, this approach requires the clustering per category as we look for category-characteristic structures. In our case, these category-category words represent rather syllables.

The procedure is as follows. We deploy program `collvec` to build an attribute matrix for each category (Section 4.5). Then we search for densities in the attribute space, which we take as category-characteristic vectors. The number of those vectors naturally varies between categories. When we apply those vectors to a testing sample, we then determine the number of best matches for each category and choose the category with the most matches.

3.4.3 AdaBoost

Adaptive boosting tries to find a hyperplane in the attribute space that discriminates categories. One would collect again the attribute matrices for each category (program `collvec`) and then search for those hyperplanes. This was successfully implemented in Matlab and a translation to C is pending.

Chapter 4

Descriptor Extraction [/DescExtr]

The program dscx outputs a number of files, some of which were mentioned already. They are written to subdirectory /Desc in our examples (of directory /DescExtr). The other subdirectories contain the following:

/Desc	output directory for description generated by dscx
/Imgs	sample images for immediate probing
/Params	example files for setting parameters
/Regist	text files containing lists of filenames
/Vect	output directory for vector files as generated by d2vmx et al.

The use of the program is explained first (Section 4.1), followed by explaining what type of data files it generates (Section 4.2). Then we introduce the available flags and options (Section 4.3). While the output is in a format that is suitable for our programs, it is less amenable for use with traditional classifiers (LDA, SVM, Random Forest, etc.). How these files can be converted to form vectors and matrices for such classifiers is explained in Sections 4.4 and 4.5 (by binaries h2arr, d2vmx, collhimg, collvec).

4.1 Program Use [dscx]

Two arguments are required, the image file path and the output file path for the data files:

```
> dscx pathImg pathOutFile
```

The input image can be jpg or png. The output file path must include a slash (as the program checks for that). Here is an example,

```
> dscx Imgs/img1.jpg  Desc/img1
```

in which the output filename img1 is chosen to be the same as the image name, for convenience. This will then write the following files into directory Desc:

img1.dsc	descriptor attributes, used by mvec, converted with d2vmx
img1.hst	descriptor histograms, used by mhst, converted with h2arr
img1.slc	saliency information.
img1.dsb	descriptor bins, used by fochst1/fochstL.

The following files provide further description, but are written only if a flag is set:

img1.kol	histogram kolumns, used by mkolL
img1.txm	texture maps
img1.qbbx	bounding boxes of proposals
img1.qdsc	descriptors of proposals
img1.dscRRE	full set of ridge/river/edge attributes, used by motvec
img1.dscCVP	full set of arc/straighter attributes

img1.BonPix	boundary pixels
img1.CntEpt	the endpoints of ridge, river and edge segments.
img1.Bbox	bounding boxes of regions.
img1.BonBbox	bounding boxes concatenated across depth.
img1.BonAsp	aspects of boundaries

The last five files of the list - from `img1.BonPix` on - are introduced in Appendix C. The others will be explained in the upcoming sections.

The script `exsbDscxSimp.m` shows how to execute the program from Matlab, using fewest utility routines for clarity. The script `exsbDscxFull.m` is a demo including utility and plotting routines. An example for a wrapper routine is given with script `RennDscx.m` that also verifies proper termination of the program. The program output (`stdout`) should terminate with the string **`EndOfProgram`**.

Image Content Requirements The program should work for any image content for sizes up to ca. 320x320 pixels, or any ratio of that size. For larger images, our program assumes that the input image is a regular scene, and not an image made of excessive or artificial texture, see Appendix D.2 for more explanations.

Parameter Intervals The program generally assumes that the parameters passed to it are in a reasonable interval. Only for few parameters we have included an interval check that returns a specific error message. For the other parameters, the program will fail somewhere during feature extraction or feature description.

4.2 Output Files

The directory /AdminMb/DescExtr contains Matlab function scripts to load the data files and to display the features. Example scripts are given in directory /Examples and deploy the output of the dsc file:

exsbPlotDesc.m	descriptor attributes, function scripts in /Vect
exsbPlotHist.m	histograms, function scripts in /Hist
exsbPlotShape.m	illustrates partitioned shapes
exsbPlotTtrg.m	illustrates the tetragons
exsbPlotBbox.m	bounding boxes, routines in /Bbox
exsbPlotBon.m	region boundaries, functions in /Bound
exsbPlotBonPix.m	function scripts in /Bound
exsbPlotSalc.m	saliency information

The following sections explain how to load the individual files and will point out more example scripts.

4.2.1 Description Image (.dsc)

The description file with extension .dsc contains the descriptor attributes and can be loaded with routine LoadDescImag.m,

```
[DSC Kt]   = LoadDescImag( fipaImg );
```

Data structure DSC contains the following descriptor image spaces:

.ACNT	skeleton of contours (not full RRE set)
.ARSG	radial description of region boundaries
.AARC	selected set of arc segments (not full set)
.ASTR	selected set of straighter segments (not full set)
.ASHP	partitioned shapes (typically not touching borders)
.ATTRG	tetragons, preciser shape info of elongated shapes
.ABNDG	bundles, clusters of contours

They are loaded with a routine named Read[Dsc]Spc.m, that in turn calls a routine Read[Dsc]Att.m. For example for contours those routines are named ReadCntSpc.m and ReadCntAtt.m. The output can be manipulated as shown in script exsbPlotDesc.m. For the partitioned-shape and tetragon descriptors there exist separate scripts, exsbPlotShape.m and exsbPlotTtrg.m. We proceed with explaining the organization of the attributes.

Attribute Organization

The attribute values are organized per type, not per vector (descriptor instance). More formally, they are struct-of-arrays, not array-of-structs, nor are they a matrix. For example, we extract the attributes of the first level of the descriptor space:

```
Cnt = DSC.ACNT{1};    % extracts the attributes for one level of the space
```

Then the attributes are available as fields, ie. the length attribute as an array in field Cnt.Len, the orientation attribute in field Cnt.Ori, etc. Thus, for the purpose of clustering or classification in Matlab (or Python), one has to concatenate them (horizontally) to a [nDsc x nAtt] matrix. The program d2vmx generates this matrix.

The attributes of a descriptor are often organized into (sub)groups. Such a group of attributes is often loaded as a matrix with the following routine:

```
[ARR szD] = ReadMtrxDat( fid, 'float=>single' )
```

where `ARR` is of size `[nDsc nAtt]`, namely number of descriptors times number of attributes. It can be converted to a struct-of-arrays with the following utility routine

```
SoA = u_AttsArrToStruct( ARR, Labels );
```

where `Labels` is a list of strings corresponding to the attribute types of that group. After that we can access the attributes by fieldname (label), ie. `SoA.Ori`.

The values of most attribute types are normalized to unit range, some only to approximate unit range in case of complex attribute definitions. Angle attributes come mostly in radians; the others normalized to unit range. Depending on the exact type of use of the vectors, one should certainly consider scaling them.

We firstly introduce attributes that are common to most descriptors. Then we introduce the details of the individual descriptor types.

General Attribute Types

The general attribute types include the position of the descriptors in the image (or map), their chromatic values, their contrast, smoothness of the segment, size, angle, etc.

- `PosV, PosH` (also called `vpo, hpo`) : vertical and horizontal position. Typically the center of the feature, ie. the midpoints of contours and boundary segments; the pole for radial descriptors. Read with `ReadAttPos.m`. These values are normalized to $[0, 1]$, unlike the points provided below.
- `RGB`: chromatic red-green-blue triplet. Read with `ReadAttRgb.m`
- `Pts`: the key-points, such as the two endpoints and the point in-between. Read with `ReadDescPtsS.m` (short), or `ReadDescPtsF` (float). Usually in absolute (unscaled) coordinates, corresponding to the pyramid level.
- `Smo, Ter.Smo`: curve smoothness, for arcs and straighters. This is a local measure that expresses the proportion of smoothness in the curve. The measure is useful to discriminate between natural scene boundaries or boundaries of specular reflections versus boundaries of objects or object parts, which tend to be smooth. A L-feature is considered very smooth, because it contains only a small proportion of non-smoothness in its corner.
- `OrgCrv`: shape/region label. The shape from which the boundary segment was taken (arc or straighter).
- `OrgDth`: the depth map at which the boundary segment was obtained (0-indexing). Perhaps useful in an analysis progressing from high to low contrast.

Other attributes that are common to most (or some) descriptors are:

- Les arc length scaled, ie. for contours, arcs and straighters
- Len arc length absolute
- Ori orientation angle, $\in [\frac{-\pi}{2}, \frac{\pi}{2}]$ (half circle)
- Dir directional angle, $\in [0, 2\pi]$ (full circle)
- Ctr contrast, normalized or $\in [0, 255]$

In the following we survey the individual descriptor types.

Contour Attributes ReadCntAtt.m

The most valuable attributes are the length and angular orientation as mentioned above already. Another attribute is the degree of straightness str, which however shows little relevance for vector matching so far.

Radial-Shape Attributes ReadRsgAtt.m

The most valuable attributes are the radius rds, the elongation elo, the degree of concavity cncv, and the circularity cir. The other attributes aid classification (with Histogram-of-Attributes from the hst file), but have not played a large role yet in vector matching.

Arc Attributes ReadArcAtt.m

The geometric parameters are loaded to variable (.Geo) as matrix [nArc nAtt], with the above mentioned routine ReadMtrxDat.m. The first column contains the curvature measure, which is the most valuable attribute. The others improve classification with histograms, but have not played a large role yet in vector matching.

Straighter Attributes ReadArcAtt.m

Contains the same attributes as for contours (above), but some other attributes are loaded as well, that are still under development.

Partitioned-Shape Attributes ReadShpAtt.m

Attributes of the partitioned shape are organized into several groups of attributes and loaded with ReadMtrxDat.m as explained above. Two groups represent a description of the straighter segments of a shape, and are useful for scene analysis of indoor or outdoor scenes, where there exist often vertical and or horizontal structures. The first group is a coarse description using a 8-bin histogram of the straighter orientations, which then is analyzed for various axial alignments. The second group is a more refined description using a 12-bin histogram. In Matlab those two groups are read as matrices as follows, see function script ReadShpAtt.m:

```
        [V.STR szL] = ReadMtrxDat( fileID, 'float=>single' );
        V.SFI       = ReadMtrxDat( fileID, 'float=>single' );
```

where .STR and .SFI are coarse and fine information, resp. In the example script exsbPlotShape, those matrices are turned into data structures by the utility routine u_AttsArrToStruct.m and variables DSC.LabShpScors and DSC.LabShpScors, that contain their fieldnames:

```
Scors = u_AttsArrToStruct( SHPlv.STR, DSC.LabShpScors );
Sfine = u_AttsArrToStruct( SHPlv.SFI, DSC.LabShpSfine );
```

The data structure `Scors` has the following fields, see Table 4.1.

Table 4.1: Coarse attributes of structure `Scors` (8-bin histogram)

`Vrt`	verticality
`Hor`	horizontality
`Dg1`	diagonal 1
`Dg2`	diagonal 2
`Axi`	axiality: both vertical and horionzontal
`Adg`	axiality along diagonals
`Vab`	deviation from verticality
`Hab`	deviation from horizontality
`Dab`	deviation from diagonality
`Tri`	three axes dominating
`Nil`	no axis is dominating: shape irregular

Each attribute represents the degree of a specific structural bias, ranging from 0 (absent) to 1 (fully present). This coarse information is useful for abstract classification. The fields of the finer description, data structure `Sfine`, are as follows (Table 4.2).

Table 4.2: Fine attributes of `Sfine` (12-bin histogram)

`Vrt`	verticality
`Hor`	horizontality
`Vti`	vertical with some inclination
`Hti`	horizontal with some inclination
`Vob`	vertical oblique
`Hob`	horizontal oblique
`Dg2`	diagonal 2
`Dg1`	diagonal 1
`Axi`	axial: both vertical and horizontal
`Uni`	one orientation
`Dul`	two dominant orientations (such as in axial)
`Cvg`	two dominant oris and converging
`Agx`	angle between the two most dominant orientations
`Ori`	orientation value of the (most) dominant orientation
`Nil`	no dominant orientation present
`Dre`	three dominant oris present
`Vir`	four dominant oris present
`Fnf`	five dominant oris present

Some are the same in structural type as listed for `Scors`, but not in value as we deal with a 12-bin histogram.

Tetragon Attributes `ReadTtgAtt.m`

The geometric attributes are organized into four groups (as data structures of the output variable):

.GEOM	basic form parameters, such as length, elongation, etc.
.LAGE	alignment with respect to the vertical and horizontal image axes
.ANGS	angles for converging direction and intersection
.DICV	directional biases

A tetragon's cornerpoints and its axis are contained in the following variables:

- .Cop the four corner points aligned clockwise starting from upper left, followed by the upper right, etc. They are the intersection of the detected segments and therefore not necessarily congruent with the actual shape, for instance for a rectangle with round corners, the corner points would outline the rectangle as if it had sharp corners.
- .Ax contains the two endpoints of the axis.

4.2.2 Histograms (.hst, .kol)

The image histogram is saved to a file with extension hst. It can be loaded with routine LoadDescHist.m, as demonstrated in script exsbDscxSimp.m:

```
[HFU HFB HSP Nbin] = LoadDescHist([fipsOut '.hst']);
```

where data structures HFU, HFB and HSP contain the histograms, flat univariate, flat bivariate and spatial (both uni- and bivariate), resp. Variable Nbin contains the number of bins used for each attribute, for the various types of histograms (.Uni, .Biv and .Spa). This serves for illustration or for own development. For classification, we rather convert the file to a single array or matrix as explained below.

The dimensions for spatial histogramming can be changed through a parameter file. This will be explained under options for architecture (Section 4.3.1).

With program binary h2arr one can convert the hst file to a single array (Section 4.4). The histograms of different images can be collected with program collhimg to create a matrix for training classifiers.

Kolumns Kolumns are saved to a file with extension .kol, if flag --saveKol is set. The kolumns can be loaded with script LoadKolumns.m. An example for loading and displaying them is given with script exsbKolumns.m in directory /Demos.

4.2.3 Texture Maps (.txm)

The texture maps for an image are saved to a file with extension txm, if flag --saveTxm is set. They can be loaded with routine LoadTxtrMaps.m, as demonstrated in script exsbTxtrMaps.m.

```
TXM      = LoadTxtrMaps( [fipsOut '.txm'] );
```

The output variable TXM is a structure, that contains the following sets of maps:

- KNT: maps based on the count of segments per window.
- OTX: maps based on the dominant orientation present.
- SAL: saliency maps that are combinations of other maps.
- CRM: maps based on chromatic statistics.

Those sets of maps are organized as structures as follows:

Count Maps KNT

The data structure contains one map, Num, whose values correspond to the sum in a window (a simple summation filter). Another map, Blk, is a binary map whose ON-pixels signify the lack or sparseness of segments, that is fewer than a minimum number of segments.

Orientation Maps OTX

Contains 7 maps that show dominance for the following orientation angles:

.Vrt	vertical orientation
.Hor	horizontal orientation
.Dg1	diagonal 1
.Dg2	diagonal 2
.Axi	axial: vertical and horizontal orientation (co-occuring)
.Nil	null: lack of a clear orientation
.Uni	one orientation is dominant (vrt, hor, dg1 or dg2)

In regular scenes, the vertical, horizontal and null-orientation attributes are the most informative, followed by the axial attribute that shows in particular in urban scenes. In scenes or textures that appear at any orientation, the diagonal attributes and the uni-orientation attribute can be beneficial.

Saliency Maps SAL

These are maps that contain contrast information and combinations thereof, in particular with the orientation maps.

Chromatic Maps CRM

Some statistics of the red, green and blue channels.

As explained already in Section 2.4, the statistics and regions of some of these maps is summarized in the saliency file, as discussed next.

4.2.4 Saliency (.slc)

The saliency file can be loaded with routine LoadDescSalc.m; the example script exsbPlotSalc.m demonstrates how to read the variables. The loading routine returns four data structures:

```
[Txa Shp Ens Dsc] = LoadDescSalc( fipsSalc );
```

that contain the following information:

- Txa: texture information based on contour statistics as explained in Section 2.4 and 4.2.3, which are particularly useful for detecting common scene textures and small objects.
- Shp: bounding boxes of salient regions, which are taken from the space of partitioned shapes in variable ASHP.

- `Ens`: an ensemble of proposals that is combination of the contour (`Txa`) and shape information (`Shp`).
- `Dsc`: statistical information of the descriptors, both the number of descriptors as well as some of their attributes.

We elaborate on what aspects each data structure holds.

Texture Data Structure `Txa`

The texture (data) structure holds information on the 7 biases, once for the entire map in structure `Txa.Gst` and once per blob in structure `Txa.Blb`. Then there exists also a data structure `Spt` holding dense point candidates. We recall that we have the following 7 texture biases:

```
aTxtBis = {'Num' 'Blk' 'Nil' 'Vrt' 'Hor' 'Axi' 'Uni'};
```

- `Gst` The data structure holds five statistical values for each texture bias (read with `ReadMapBisStat.m`): the proportion present in the map; and the minimum, maximum, mean and standard deviation. In a scene full of texture, the numerosity value `Num` is high for its proportion, its voidness `Blk` value is low. In a scene lacking contour texture, the inverse holds: low numerosity and high voidness. The example script `exsbSalBlobs.m` shows those two cases.

- `Blb` this data structure contains the blob information, the regions outlining contour texture. It is loaded with reading routine `ReadBlobOut.m` and contains the following structures and fields:

`.Box`	bounding box values in fields `.Top`, `Bot`, `Lef`, `Rit`.
`.Typ`	texture bias, numbering 1 ('Num') to 7 ('Uni'). 8: high contrast
`.Cvg`	coverage of image, $\in [0, 1]$

There may exist multiple blobs per texture type. The bounding boxes for `Typ=1` are the most general ones and outline any blob containing contour segments: if the boxes are small, than they outline small objects in isolation, demonstrated with example script `exsbSmlObjDet.m`.

- `Spt` contains a selection of points that represent clusters of high contour count and of high contrast, relative to their immediate context. They are plotted in the section called 'Spots' (in `exsbPlotSalc.m`), with routine `p_VisSearch.m`. These points are useful if the scene contains spots of moderate cluttering that we wish to locate, ie. when searching for objects.

Shape Data Structure `Shp`

This data structure holds the bounding boxes of selected shapes and their key aspects. They are read with reading routine `ReadShpOut.m`.

`.Box`	bounding box values in fields `.Top`, `Bot`, `Lef`, `Rit`.
(`.Typ`	irrelevant (NOT texture bias))
`.Cvg`	coverage of image, $\in [0, 1]$
`.Ctr`	contrast, $\in [0, 255]$
`.Cwd`	contour crowdedness (value from texture map `'Num'`)
`.Lev`	level from which a shape was taken ($\in$ `ASHP`), $\in [0, \texttt{nLev} - 1]$
`.IxShp`	index to shape (of the level in `Lev`)
`.IxBon`	index to boundary (level in `Lev`)

Ensemble Data Structure `Ens`

The ensemble (data) structure holds a combination of the contour and shape proposals as well as an ordering of the their sizes:

`.Box`	bounding box values in fields `.Top`, `Bot`, `Lef`, `Rit`.
`.Typ`	descriptor type: 1-8 = contour texture; 10 = shape.
`.Cvg`	coverage of image, $\in [0, 1]$
`.Ctr`	contrast, $\in [0, 255]$. For contours set arbit. to 100. For shapes real value.
`.OrdGtoL`	order of indices from large to small (global-to-local)

Descriptor Data Structure `Dsc`

The statistics in this data structure are too innumerous to list them all. We here describe a few selected ones, that can be useful for active vision (Section 12.2).

- `MxRngRR`, `MxRngEg` These two arrays contain the maximum contrast (range) value of ridges and rivers (RR) and edges (Eg) per level of the image space `IS`. This information is useful for autofocusing; if the RR values are extremely low (less than 10), then we likely face a blank visual field with perhaps specular reflections; or a scene with no clear contours, such as underwater scenes, or with regions of low contrast.

- `MaxSizScl` contains the maximum size per level for each descriptor. If the one for shapes is large, ie. `max( MaxSizScl.Shp )`, then we likely have a large object in the image center (if parameter `Shp.bordTouches` $= 0$, which is default); or it can be background, that is surrounded by objects, such that the background appears as an inside shape.

- `GryMmm` is a three-value array that contains the minimum, mean and maximum gray-level value for the entire image.

4.2.5 Proposals (`.qbbx`, `.qdsc`)

The persistent proposals are saved to file if flag `--saveProp` is set, see example script `exsbProposals.m`. The loading routine returns a data structure `Bbx`

```
[BBx Nr]    = LoadDescPropBbox( [fipsOut Fixt.qbbx] );
```

that contains the bounding boxes selected from:

- `ShpGen` the entire partitioned-shape space (`ASHP`).

- `TtgGen` the entire tetragon space (`ATTG`).
- `AxVrt` partitioned shapes that appear vertical (of `ASHP`).
- `AxHor` partitioned shapes that appear horizontal (of `ASHP`).

The attributes of the selected shape and tetragon descriptors are also extracted: they are saved to the `qdsc` file, which is loaded as follows:

```
[QDSC]  = LoadDescPropAtts( [pthfips Fixt.qdsc] );
```

where structure `QDSC` contains the attributes as introduced previously. In this case they consist of quasi one level.

4.3 Options and Parameters

Options and parameters can be passed either as text file or as long options.

- text file: For the use of a text file see the directory `/Params`. The file named `PrmDesc_Example.txt` contains the most important parameters. The parameter file must start with the string `PrmDesc`.

- long option (double dash `--`): they must appear after the specification of the parameter text file, if one uses both types of passing simultaneously. Values given by long options are given preference to the ones in the text file (if one is specified).

An example of passing with both types would be as follows:

```
> dscx Imgs/img1.jpg  Desc/img1 Params/PrmDesc_Example.txt --depth 4
```

where the text file must appear as the third argument, followed by long option arguments. In this case, `--depth 4` overwrites `depth` in the text file - if it was also specified there. The naming of parameters can be slightly different for the two types occasionally.

Effect Many of the parameters will mainly regulate the number of descriptor vectors saved to the `dsc` file. For classification or matching with histograms, this typically has little effect since the histograms are created with the full set of descriptors is taken (RRE for contours, full for curve partitions); the exact choice of those parameters has lesser influence on recognition accuracy (with histograms). For manipulation with vectors however, some of these parameters can make a huge difference. In particular for identification in place recognition, one can observe substantial variations.

In the following the parameter name for the text file is listed first in `red`, if it can be specified by text file. The long option is specified with a double dash `--` and is listed in parentheses, if the same parameter can also be specified in the text file. Single letter options are not in use.

4.3.1 Architecture

`nLev` (`--nLev`): number of levels of the image space. For a pyramid, the number is calculated automatically (with downsampling factor equal 2), whereby the top level does not subceed 16 pixels for one map side. For example, for a 256x256 image a five-level pyramid is generated: 256 (original resolution), 128, 64, 32 and 16. For a scale-space, the default equals five.

`imgSpc` (`--is`): sets the type of image space `IS`. By default it is the pyramid, with value equal one. The scale space can be selected by setting to value equal two. A demo script is in directory `/Demos` called `exsbImgSpaces.m`.

`HistSpaDim`: sets the grid dimensionality for spatial histogramming. The default is 3x3. The dimensions are specified as rows and column, ie. changing to three horizontal cells that cover the entire image width, would be specified as:

```
HistSpaDim 1 3
```

Note that a grid of 5x5 generates a very high dimensionality with over 30k bins. The demo script `exsbSpaHist.m` gives an example (in directory `/Demos`).

Another parameter that can be considered part of the architecture is the window size for texture analyis. It will be mentioned further below, in Section 4.3.7.

4.3.2 Contours

`Cnt.minCtr` (`--cntMinCtr`): contrast threshold for contours. Default `=0.05`. This is the proportion, of the largest difference found in the range map of the gray-scale intensity image (for a 3x3 neighborhood).

The following two parameters - starting with `skl` - modify the output of the contour selection, the skeleton. To understand those changes see the example script `exsbContourSkel.m` in directory `/Demos`, or turn on plotting using flag `--plot`, which writes the image `ImgPyrSkel.png`.

`Skl.MinSpc` (`--sklMinSpc`): minimum spacing. Default `=0.05`. This parameter is specified as proportion of the image side length, ie. for an image side of 256 pixels it will be 13 pixels for the default value. Changes here can have a huge effect on performance, recognition accuracy in particular.

`Skl.MinLen` (`--sklMinLen`): minimum length. Default `=0.05`. This parameter is also specified as the proportion of the image side length. Changes here are less significant, in particular for large spacing values (set with `sklMinSpc`), as then only few short segments remain.

4.3.3 Regions

`depth` (`--depth`): depth of the segmentation process. Default `depth=3`. For `depth=1` no tree is grown: this corresponds to global thresholding only (with a single threshold). `depth=4` can be useful for large images, e.g. larger than 1000 pixel for one image side. A depth of five is the maximum.

`Reg.minPixNode` (`--regMinPixNode`): minimum number of pixels for a region to be segregated by the thresholding mechanism. This will affect the region count from the second segmentation map on. It will not affect the first segmentation map, as its regions are the first nodes of the tree.

Default equal 6. With larger values, processing occurs more rapidly, but may not capture fine texture properly anymore.

4.3.4 Radial Shape

These are parameters determining entry conditions (Section 2.3.1):

`Rsg.minNpx` (`--rsgMinPix`) [absolute]: minimum number of boundary pixels for a radial descriptor. The number is set for the original image resolution. For higher levels of the pyramid, a correspondingly lower number is used, namely `rsgMinPix-level`. E.g. for a value of 10, the higher pyramidal levels utilize values `9, 8, 7,...`.

Default equal 5 for all levels. For larger values, ones risks loosing texture, thus if one interested in the global structure only, it can be useful to set higher values.

`Rsg.maxNpx` (`--rsgMaxPix`) [factor]: maximum number of boundary pixels for a radial descriptor. The value is multiplied with the sum of image side lengths. A value equal one means the maximal square boundary possible would correspond to the image size. Default equal 10: allows a star/asteriks silhouette covering the entire image.

`Rsg.minCtr` (`--rsgMinCtr`) [factor]: minimum boundary contrast required for radial description. A value equal zero means all boundaries will be described as radial descriptor. The specified contrast value is multiplied with the average boundary contrast value. A value equal one means that only boundaries with a contrast higher than their average are used for radial description. To turn off any radial description, one can set the value to 255.

4.3.5 Partitioning (Arcs/Straighters)

Parameter names with prefix `cvp` regulate the partitioning process and therefore affect the outcome of both arc and straighter partitions. Parameter names with prefix `arc` and `str` are specific to the respective descriptors. The following two parameters are entry conditions.

`Cvp.minPix` (`--cvpMinSiz`) [proportion]: minimum boundary length entering the boundary partitioning process. It is specified as proportion of the larger image side (`max(szI)`). Default value is equal 0.05, ie. 13 pixels for an image side length of 256 pixels. A value of 0.02 for a [1024 x 2048] image will set the minimum size to 41 pixels. The calculated value cannot be below 12 pixels, as that is the minimum number of pixels entering the partitioning process.

For small values this will seemingly regulate the number of straighter segments only, as those are rarer. For higher values, it will also start omitting arc segments.

`Cvp.minCtr` (`--cvpMinCtr`) [proportion]: minimum boundary contrast entering the boundary partitioning process. This is a proportion of the largest boundary contrast found in boundaries. Default equal 0.05.

Arcs and Straighters

`Cvp.prpMinLenArc` (`--arcMinLen`): minimum arc length to be described. This is relative to image side length. Default equal 0.08. This is a better way of directly regulating the number of arcs than entry parameter `Cvp.minPix`.

`--strMinGer`: minimum straightness value for a straighter segment to be accepted. This is a fixed threshold $\in [0..1]$ based on the measure chord length divided by segment arc length. Default equal 0.8.

`Cvp.inclBord` (`--cvpInclBord`) [flag]: includes partitions at borders. The default is OFF, meaning that partitions at image borders are ignored. Turning them ON (by listing this option) can improve place recognition significantly.

The following are skeletonization parameters for gerust formation (`Gst`). Reducing the number of segments for finer scales often improves recognition - and also reduces matching duration. But I suspect that for smaller images (or higher levels of the pyramid; ie. smaller 100 pixels per side), that the selection might have less effect.

`Gst.minSmoArc` (`--arcGstMinSmo`): minimum smoothness for arcs set for entire pyramid. Default approximately 0.20. This parameter is intended to eliminate segments resulting from segregation of luminance gradients, that typically show high irregularity in their spatial course. Images of scenes taken at night tend to have those in particular.

`Gst.minSpcArc` (`--arcGstMinSpc`): minimum spacing for arcs set for entire pyramid. Default approximately 0.05.

`--arcGstOff`: turns off any selection by setting all parameters values to zero (for smoothness, spacing and length). This then takes the full set of extracted arcs.

`Gst.minSpcStr` (`--strGstMinSpc`): minimum spacing for straighters set for entire pyramid. Default approximately 0.05.

`--strGstOff`: turns off any selection by setting all values to zero. This then takes the full set of extracted straighters.

4.3.6 Partitioned Shape (Arcs & Strs)

`Shp.bordTouches`: number of border touches permitted for a partitioned shape. Default equals zero, meaning only inside shapes are described. Values 1 to 4 permit the corresponding number of border touches for a shape.

For more parameters consult the parameter file `PrmDesc_Example.txt` in directory `/DescExtr/Params/`.

(`Shp.minCtr`: minimum contrast for a shape. Not in use yet. Currently depends on the boundary contrasts measured.)

4.3.7 Texture

`Txt.winSiz` (`--txws`): side length of the rectangular window. The default is 16 and serves well for image sizes of 256x256 depicting regular scenes. For larger image sizes, an increase might make sense. For the search of small objects, a smaller size can be beneficial, see also the example `exsbSmlObjDet.m`.

4.3.8 Utility

`--prms` [flag]: displays the parameter values used. Default OFF.

`--saveBbox` [flag]: turns on saving of bounding boxes (`.Bbox` file) and contour endpoints (`.CntEpt`). Default OFF.

`--saveBon` [flag]: turns on saving of boundary information (files `.BonBbox`, `.BonAsp` and `.BonPix`). Default OFF.

`--noBin` [flag]: turns off saving of descriptor bins (file `.dsb`). Default ON.

`--plot` [flag]: plots contours and region boundaries for the entire pyramid; not available for scale space (image space `is=2`). Default OFF. The following images will be written:

- `-Icnt.png`: contours plotted onto the color image for the original resolution.
- `-ImgPyrBonOnly.png`: boundaries of the entire pyramid and for different depths.
- `-ImgPyrCntOnly.png`: contours of the entire pyramid and for different levels.
- `-ImgPyrSkel.png`: the selected contours (skeleton).

`--verbose` [flag]: for illustration or for tracking errors. Default OFF.

4.4 Collecting Histograms [h2arr, collhimg]

The histogram file (hst) can be converted to a text file, that contains all histogram values as a single array, by applying program binary h2arr ('histogram to array'). The outputted array can then be applied directly to a classifier.

The program takes a histogram file as first argument and an output file stem as second argument, for example:

```
> h2arr Desc/img1.hsf Vect/img1
```

This will append the extension hari and write the file named `img1.hari` to the directory `/Vect`. It can be loaded as shown in `LoadHistImgArr.m`. A full code example is given with script `exsbH2arr.m`.

The histogram values represent the raw count; no scaling was carried out. The total dimensionality of the histogram can be over 24 thousand for spatial histogramming with a 3x3 grid (the default). The example script `exsbSpaHist.m` in directoy `/Demos` displays the histograms for different grid dimensions.

The program can also be used to convert a focus histogram to an array, more in Section 7.1.

With binary collhimg we can collect the histograms for multiple images. For `nImg` images, the program then generates a matrix of size, `nImg x nBin`, namely number of image histograms times number of bins. It can be deployed for training traditional classifiers such as LDA, SVM, Random Forests, etc. The same output can be achieved by calling h2arr individually and then concatenating the outputted histograms.

The program takes two arguments. The first argument is a text file containing a list of histogram files. The second argument is an output name for the matrix:

```
> collhimg ListHists.txt COLLHST
```

This will append the extension hstc to the output name, and write the matrix into the file `COLLHST.hstc`, where letter 'c' in the file extension stands for collection. The matrix is written to file in binary format (not as text). It can be loaded with routine `LoadCollHist.m`. An example script is in directory `/MtchHst`, called `exsbCollHist.m`.

The second output variable of `LoadCollHist.m`, here called `Nbin`, contains the bin numbers of the individual histograms. The 4-element array `Nbin.Tot` holds the bin numbers for the four types of histograms: flat univariate, flat bivariate and the two spatial versions.

4.5 Generating Vector Files

The attribute values can be collected to form vectors with binaries d2vmx and collvec, analogous to the binaries h2arr and collhimg collecting histograms. Program binary d2vmx does so for one image (description to vector matrix). Program collvec does so for a list of images. The outputted matrix has the size number of descriptors (rows) times number of attributes (columns): `ntDsc x nAtt`, where the total number of descriptors `ntDsc` is for either one or several images. A corresponding label array is written as well, that contains level and image index.

4.5.1 One Image Description [d2vmx] (.vecCnt, .vecRsg, ...)

Program binary d2vmx generates a vector file for each descriptor type. It takes as input a description file and requires the specification of an output file path, which in our examples is saved to the folder /Vect:

```
> d2vmx Desc/img1.dsc Vect/img1
```

This call then generates a separate vector file for each descriptor type with extension vec[Dsc], ie. img1.vecCnt, img1.vecRsg, img1.vecArc, etc. The vectors are written rowwise, the columns therefore represent the attribute types. The output file path must contain a slash sign.

Level Array For each descriptor type, the program also generates a label array that contains the level index (of the image space) where the descriptor instance comes from. The indices are written with zero-indexing. The file extension is called lev, standing for 'vector level'. The full file extension are therefore called .levCnt, .levRsg, etc.

The files can be loaded by routines LoadDescVect.m and LoadDescVectLev.m, respectively. An example script of how to run the program and read the output files is given in directory /DescExtr, called exsbD2vmx.m.

The script o_AttsLabels.m provides the attribute labels. Each descriptor contains two columns for the position information, vpo and hpo, for the vertical and horizontal position, respectively. The chromatic attributes are labeled red, grn and blu.

4.5.2 List of Image Descriptions [collvec]

This program allows to concatenate the vectors from multiple image descriptions into a single matrix. In simpler words, it is the list version for program d2vmx, but we call it collvec in analogy with program collhimg for histograms. The same result can be achieved by concatenating the output from d2vmx.

While d2vmx generates the vector file for each descriptor type automatically, here we run the program for *one* descriptor type only, specified with the second input argument. Thus, the program takes three arguments. The first argument is a text file containing a list of description (dsc) files. The second argument specifies the descriptor type, ie. 'skl', 'rsg', etc. The third argument is an output name for the matrix, called COLL here:

```
> collvec ListVec.txt skl COLL
```

This will append the suffix VEC_[Dsc] to the output name and write the matrix into the text file named COLLVEC_skl.txt in this example. The matrix can be loaded with script LoadCollVec.m.

The program also writes a text file with labels to a file named COLLLab_skl.txt, where the suffix is Lab_[Dsc]. The label array is of size ntDsc x 3, where the first column is the index of the level from which the descriptor was taken; the second column the image index; the third column is not used presently.

It can be loaded with LoadCollVecLab.m. An example script is given in directory /DescExtr, called exsbCollVec.m.

Chapter 5

Matching Vectors [/MtchVec]

Program binary mvec matches the descriptor vectors of two images (.dsc files) as generated by the descriptor extraction program (dscx), or two focus files (.dsf) as generated by focsel. It can be used for any two structures expressed by the vectors, be it a scene, an object, a shape or a texture. The directories in the folder contain the following:

/Desc	description or focus files (as outputted by dscx/focsel)
/Imgs	sample images for immediate probing
/Mes	results of matching metrics
/Params	contains example files for setting parameters
/Regist	text files containing a list of filenames of description files

The program mvec matches the two lists of descriptors using pairwise measurements and choosing the nearest neighbor. Two metric measures are available, a dissimilarity and a similarity value, abbreviated dis and sim, or sometimes also abbreviated as dist and simi, resp.:

- dis (dist) returns the Euclidean distance.
- sim (simi) returns the proportion of matches that are below a fixed threshold value, set with option [dsc]TolMtc or tolMtc for all descriptor types.

The program can be applied to two description files (dsc) as outputted by dscx, or to two description focus files (dsf) as outputted by focsel. The combination of a dsc and dsf file is not possible yet. The description files are required to have the same number of levels for the descriptor space DS, otherwise the program returns no results (more flexibility to be included in the future).

The program comes in two variants:

- mvec1 : matches one pair of images (or focii): one versus one. Its output is very elaborate, for example it generates nearest-neighbor information suitable for learning category-characteristic descriptors. The Matlab script exsbFrames.m demonstrates how to deploy the program and read its output. This program is useful for exploring parameter settings for different levels of the descriptor space.
- mvecL : matches a list of images, or a list of focii: one versus multiple. Its use is demonstrated in Matlab script exsbMvecLimg.m. It outputs the measurements per image only and not for the entire descriptor space (as in mvec1). It is useful for

matching at large scale. The demo for place recognition gives an applied example with scripts `plcDscx.m` and `plcMtcImg.m`.

Then there exists also the binary `motvec`, that calculates the motion vectors between nearest neighbors. It is similar to `mvec1`, but returns the vectors only, useful for estimating motion flow, introduced in Section 5.3.

5.1 Program Use [mvec1, mvecL]

We firstly explain the use of `mvec1`, the matching of two description files. Their file paths are given as arguments. For example for two description files from images (`.dsc` from `dscx`) we write:

```
> mvec1 Desc/img1.dsc Desc/img2.dsc
```

Or for two focus file (`.dsf` from `focdsc1`):

```
> mvec1 Desc/foc1.dsf Desc/foc2.dsf
```

The description files are required to have the same number of levels, ie. generated by similarly sized images, otherwise it returns no results. The output will be further explained in Section 5.2.

For the use of the program `mvecL`, we specify a file path as well as a text file that contains the file paths for a list of files to be matched with. We prefer to keep those text files in a folder called `/Regist` (for register):

```
> mvecL Desc/img1.dsc Regist/FinasImg.txt
```

This will write the metric measurements into a file named `Vec.txt` in directory `/Mes`. One can specify a different file name by providing a third argument, ie.:

```
> mvecL Desc/img1.dsc Regist/FinasImg.txt Mes/ImgVec.txt
```

By default, the program matches all descriptor types for the entire pyramid. The options allow to select descriptors and levels, as well as to set attribute weight values.

5.1.1 Options and Parameters

Options and parameters can be passed by text file or by long options (as in case of `dscx`). If options are passed by file, then the filename must start with the string `PrmMtch` and appear as the third argument, before the filename of the measurement files - which in that case is specified as fourth argument, for example:

```
> mvecL Desc/img1.dsc Regist/FinasImg.txt Params/PrmMtch_PosTol.txt Mes/ImgVec.txt
```

In the following the use of the long options is explained. They are specified at the end of all previous arguments.

The first list of options, under 'General' below, set a parameter value to the same value for all descriptor types. This can be unspecific in some cases, but is convenient for coarse tuning. The second list of options, under 'Individual', allows to adjust parameters of individual descriptor types for fine tuning.

General (All Descriptor Types)

The following options set values for all descriptor types simultaneously and are useful for coarse tuning. Default values are given in approximate values only, as they are often individual to the descriptor type:

`--tolMtc` [similarity metric]: sets matching tolerance to a fixed value, arcoss all pyramidal levels (and across all descriptor types). This is for the similarity metric only (section `Simi` in output). Default: ca. 0.05.

`--wgtRGB`: sets the weight value for the RGB difference. Set this parameter to zero if chromatic information is irrelevant, for example when places are to be recognized at either day or night. Keep in mind that three difference values are taken (R,G,B) and that this weight parameter therefore has more influence than the others. Default: ca. 1.0.

`--wgtPos`: sets the weight value for the position parameter for each descriptor type. A value of 0 turns off the influence. Default equal ca. 1.0.

Individual (Per Descriptor Type)

In the following, the strings `{Dsc}` and `{dsc}` denote one of the descriptor types, ie. `Cnt` and `cnt`; `Rsg` and `rsg`, etc. The directory `/MtchVec/Params/` contains examples of how to set the parameters by file.

`{Dsc}.tolMtc` (`--{dsc}TolMtc`): tolerance for matches with the similarity metric. Fixed value for all levels. Default: around 0.025, but varies for type.

`{Dsc}.tolPos`: tolerance of position value. When the difference is larger, the match is ignored.

`{Dsc}.wPos`: weight of position value.

`{Dsc}.wRgb`: weight of chromatic values. See parameter file `PrmMtch_RgbOff.txt` for an example.

`{Dsc}.wOri`: weight of orientation angle.

Contours

For contours we can also manipulate the weights of the length and straightness influence, with `Cnt.wLen` and `Cnt.wStr`, respectively.

Utility

--prms: displays the parameter values used.

5.2 Output

5.2.1 Program mvec1

Two types of results are returned. One type are the measurements of the list-matching metrics, written to stdout. The second type are the nearest neighbor indices (Section 5.2.1). See the example exsbFrames.m for the upcoming explanations.

Descriptor List

The list-matching measurements (appearing in stdout) are returned once for the entire pyramid, that is for each level (and each descriptor); and once integrated, for each descriptor type; as well as for the total, called image measure. The values per descriptor type and per image look as follows:

```
----- desctypes -----
dty  dis       sim
skl  1.357370  0.117928
rsg  1.480588  0.098934
arc  1.489363  0.009344
str  1.351547  0.054755
shp  5.827399  0.048974
eodty.
----- img -----
dis 23.574306
sim 0.000000
eoim.
```

The strings `----- desctypes -----` and `----- img -----` help locating the beginning of the respective measurements sets; as well as the 'end-of' strings `eodty` and `eoim`. This is carried out with the Matlab script pso_Mvec1Sections. The actual measurement values are read by routine pso_Mvec1Vals.

Nearest Neighbors

The nearest-neighbor indices are written to files in directory /Mes with prefix NNspc and suffix 12, the latter indicating direction of comparison. The indices are kept separate for each descriptor type, ie. NNspcCnt12, NNspcShp12, etc. They are loaded with scripts LoadNNDspace and ReadDescNNs. The section 'Correspondence' in script exsbFrames.m establishes a visual correspondence for illustration.

5.2.2 Program mvecL

The results are written to three files into directory /Mes, called Vec.txt, MesDtyDis.txt and MesDtySim.txt. Each row corresponds to one pair of matching.

- Vec.txt contains the metric measures for the ensemble of descriptor types. It consists of four columns, where the first is the dissimilarity value, the second the similarity value. The third and fourth column are empty (zero) for the moment. The Matlab routine LoadMtchMes.m shows how to load the file.

- MesDtyDis.txt contains the dissimilarity value for each descriptor type, organized column-wise corresponding to contour, radial shape, arc, straighter, partitioned shape, tetragon and bundles of contours.

- MesDtySim.txt contains the similarity value for each descriptor. Its organization is the same as for dissimilarity values.

With the output of files MesDtyDis.txt and MesDtySim.txt one can develop individual ensemble measures.

5.3 Motion Vectors [motvec]

The program motvec takes as input two description files and the arguments are therefore specified analogous to program mvec1:

```
> motvec Desc/frm1.dsc Desc/frm2.dsc
```

This will output the vectors into directory /Mes to a file named A.motvec. It can be read as demonstrated with script LoadMotVec.m. A complete example is given with exsbMotVec.m.

The output structure contains fields for the endpoints (Ep1, Ep2), the magnitude (Mag) and the angle (Dir). The field Dis contains the dissimilarity value of the matched descriptors. One could attempt to improve the flow estimate by excluding those that are very dissimilar. This is exemplified with in the place recognition experiment, see plcMotEgo.m, upcoming in Section 11.6.

The vectors from the different descriptor types were concatenated in the following order: contours, radial, arc and straighter. Their count is given in the respective fields, nRdg, nRiv, nEdg, nSkl, nRsg, nArc and nStr, resp. One can therefore separately access the motion values for the different types.

By default the program uses the skeleton contours as provided in those description files (structure ACNT). The counts for nRdg, nRiv and nEdg appear with value -1. The program will also search for the presence of the dscRRE files in the same directory (in our example /Desc); if they are present, then the RRE vectors are taken for motion calculation, and the skeleton descriptors are ignored.

Since the program dscx does not save the RRE set by default, this has to be triggered by the corresponding flag. Example script exsbMotVec.m demonstrates how to do that, ie. by setting OptK.saveRRE = 1;.

Chapter 6

Matching Histograms [/MtchHst]

There are two program binaries matching histograms. One for the Histogram-of-Attributes, and one for the texture kolumns, called mhstL and mkolL, respectively. Their use is analogous and we therefore firstly explain the usage of the former, and then mention the latter (Sections 6.1 and 6.2, respectively). The use of both is exemplified with the script plcMtcHstKol.m as part of the demo for place recognition. Individual demo scripts exist as well, coming up.

6.1 Histogram-of-Attributes [mhstL]

The program mhstL matches an attribute histogram against a list of other attribute histograms. This coarse comparison allows to identify a selection of candidates, that are then applied to a classifier, or that are immediately used for more accurate matching with mvec.

The program takes as input a histogram file (.hst file) as generated by the descriptor extraction program dscx, or as generated by focus selection program, fochst1 (or fochstL). The directories in the folder contain the following:

/Desc	histogram files (as outputted by dscx/focsel)
/Mes	results of distance measurements
/Regist	text files containing a list of filenames of description files

The program mhstL calculates the Hamming distance between histograms and outputs the array of measurements to a file. Its input arguments are analogous to those of the program for vector matching (mvecL).

6.1.1 Program Use

Analogously to the program mvecL, we specify a histogram file (.hst) and a text file containing the file paths for a list of histogram files to be matched with:

```
> mhstL Desc/img1.hst Regist/FinasImg.txt
```

This will write the metric measurements into a file named HstLst.txt into directory /Mes. One can specify a different file name by giving a third argument, ie.:

```
> mhstL Desc/img1.hst Regist/FinasImg.txt Mes/MtcImg.txt
```

The histogram files need to originate from images of (almost) similar size, in order to properly match the spatial histograms.

By default, the program matches all descriptor types, ie. contours, arc segments, straigher segments, etc. as that typically yields best results. Matlab script `exsbMhstL.m` (in the main folder), gives an example of how to run the process.

The program can also take focus histograms (.`hsf`) as input:

```
> mhstL Desc/img1_f1.hsf FinasFoc.txt
```

in which case the list of filenames in `FinasFoc.txt` need to be `hsf` files as well. For the focus histograms their original size is irrelevant, as no spatial histograms are involved; the idea of focus histograms is to create individually sized spatial histograms. Thus, the user must determine, whether it makes sense to compare two focii of different size.

6.1.2 Output

The measurement values are written twice to file, once ordered and once unordered, both in text format. The ordered output, written to `Mes/Hst.txt`, contains two columns, one containing the index of order in zero-indexing format, and the other the distance values. The ordered file, named `Mes/HstUor.txt`, contains the measurement values in unsorted order without any other information.

6.2 Kolumns [`mkolL`]

As mentioned already, the usage of this binary is analogous to the use of `mhst`. We specify a kolumn file (.`kol`) and a text file containing the file paths for a list of kolumn files to be matched with:

```
> mkolL Desc/img1.kol Regist/FinasKol.txt
```

This will write the metric measurements to a file named `KolLst.txt` into directory /`Mes`.

Kolumns are generated for the entire image only. There is no focus selection. An example of how to use this binary is given with script `plcMtcHstKol.m` as part of the place recognition demo.

Chapter 7

Focus Selection [/FocSel]

The goal of focus selection is to enable to apply a classifier or identification process to a specific (rectangular) region in order to accelerate visual browsing by maximally exploiting the descriptor output. In other words, before we move the camera direction (in active vision) or apply the shape-extraction program `shpx` (Chapter 8), which requires more computation, we make full use of the description file (`dsc`) that allows us to make better choices of where to look next. Focus selection identifies the descriptors of a specified rectangular region, the *focus*, and saves them to the *focus file* for further analysis, ie. to be matched with `mvec` or `mhst`. The program takes as input a description file (`dsc`) and a region specified as bounding box. The selection process comes in three variants:

- `focdsc1` : extracts the subset of the descriptor spaces for one focus, and saves them to a file with extension `dsf`. That file can be loaded by `mvec` for full vector-by-vector matching in order to identify structure.
- `fochst1` : generates an attribute histogram for one focus, and saves it to a file with extension `hsf`, useful for rapid classification and for matching with `mhst`.
- `fochstL` : generates attribute histograms for a list of focii as specified in a text file.

Example scripts explaining how to deploy those programs contain the prefix `esxbFoc`, ie. `exsbFocDsc1.m`, `exsbFocHst1.m`, `exsbFocHstL.m` and `exsbFocDscFew.m`. They call the corresponding wrapper functions named `RennFocDsc1.m`, `RennFocHst1.m` and `RennFocHstL.m`. If one prefers to dive directly into an application, then the demo for place recognition offers a number of scripts, see the summary script `plcAll.m`.
The folder /`FocSel` contains the following directories:

/`Desc`	description (`dsc`) files as generated by `dscx`
/`Focii`	output directory for focal selections, the `.dsf` files

The process extracts the corresponding part of the original feature pyramid. To illustrate that, we refer to the scheme in Section 2.2. For example selecting from a quarter region of the image (with four levels), the corresponding subset of the list is organized as follows,

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

and starts with level 0. It reaches only level 2, as we extract a subset of the pyramid. The extracted focus pyramid has therefore three levels, nLevFoc=3.

The programs calculate the number of pyramid levels automatically for the selected region. If no descriptors are found in that subspace, then no output file will be generated.

We firstly introduce the generation of histograms, then the one for attribute description, Sections 7.1 and 7.2, respectively.

7.1 Program Use and Output [fochst1, fochstL] (.hsf1)

Binary fochst1 requires both the dsc and the dsb (bin) file as input, but we specify only the dsc file, and expect the dsb file to be present in the same directory as the dsc file. The program arguments are then as follows:

1) a description file as generated by dscx. The file name *must* include extension dsc.
2) bounding box parameters specified as `top bottom left right`.
3) [optional] an output file name *without* extension, where the selected descriptors are written to.

For example, we wish to extract a `40x40` region (height x width) from the upper left part of an image:

```
> fochst1 Desc/img1.dsc 10 50 10 50 Focii/foc1
```

It generates the focus histogram, which is written to a file with extension hsf1. The program also writes to standard output, namely the number of levels, that was calculated automatically, `nLevFoc`, and the total number of descriptors detected in the focus, `ntDsc`. The output can look as follows:

```
nLevFoc 3
ntDsc 27
```

If no descriptors are found, the standard output returns `ntDsc 0` and no file is generated. This may happen if we specify a very small bounding box, or a region lacking any other descriptor, ie. a specular reflection. An example script is given with exsbFocHst1.m. It also demonstrates how to use binary h2arr to convert the histogram file to a single array. It will append the extension harf to the output filename.

The program fochstL does the same as fochst1 but for a list of focii specified in a text file, named BboxFocii.txt. For example we specify:

```
> fochstL Desc/img1.dsc BboxFocii.txt FOCII1
```

for which the bounding boxes are given rowwise. The first line of the file must contain the number of bounding boxes to be read. The output is written to file `FOCII1` in the example.

7.2 Program Use and Output [focdsc1] (.dsf)

Program focdsc1 is deployed with the same arguments as program fochst1, ie. we specify:

```
> focdsc1 Desc/img1.dsc 10 50 10 50 Focii/foc1
```

The selected descriptors are then saved to directory /Focii with extension dsf. It also inserts a suffix to the file stem (base file name), ie. foc1_lev2.dsf, that holds the number of levels of the focus description (2 in this example). This is done to allow to organize the outputted files more quickly, ie. using system functions for finding lists of files in a directory. An alternative to obtain that information is to load the header of a focus file and extract its size from it. We need this level information to ensure that we match the corresponding descriptions, because program mvec matches only descriptor spaces of same height, ie. the same number of levels.

The dsf file is similar to the dsc file (generated by dscx). It is loaded as demonstrated in script LoadFocDesc.m, located in directory /AdminMb/FocSel/. The example script exsbFoc1.m plots both the description of the entire image, and those of the focus (selection). The plotting routines (PlotCntSpc.m, PlotRsgSpc.m, ...) are found in directory /Plot of the folder for descriptor extraction (/DescExtr).

Chapter 8

Shape Extraction [/ShpExtr]

The goal of shape extraction is to obtain a more accurate description for a certain shape silhouette, than it was obtained with dscx; a more precise term would be 'shape descriptor extraction'. To achieve this, we specify the RGB triplet of a region (silhouette) under investigation, for example taken from the partitioned-shape descriptor shp of the description file. That region can represent anything of interest, a simple shape itself, an object silhouette, an object part, or scene part.

Using the specified RGB triplet, program shpx then carries out a simple color-segregation process, whose output can be studied with the demo program sgrRGB, to be introduced in Chapter 10. This process achieves a finer segmentation than that was obtained with dscx. Program shpx then describes the shape as was done in dscx, namely by descriptors rsg, arc and str, whose attribute values will be more accurate due to the finer outline. In addition to those descriptors, shpx also returns the spectra of the local-global space. The entire description is then saved to a file with extension shp, that can be used with the shape-matching program mshp1 (next chapter).

An applied example for this exists in the demo for place recognition (Section 11.5). In the following we firstly explain the use of the shape extraction program (Section 8.1), then the one for patch extraction (Section 8.2). In practice, we deploy the latter program first, but it is here mentioned second, as it is rather a utility program.

8.1 Program Shape Extraction [shpx]

The input to the program is an image patch or an entire image. Taking an entire image may make sense, when the shape is large and covers almost the entire image width or height. If the shape under investigation is smaller, than it is of advantage to crop the image to its expected size and save it as an image patch, ie. extracting the individual letters of some text (in the wild). Cropping helps eliminating any distracting region. Cropping should not be too tight, as that makes proper curve description difficult; a border of at least one pixel should be included. By specifying an RGB triplet, we ensure that we deal with only that one shape under investigation. For example, we provide the following arguments:

```
> shpx Ptch/img2_ptch1.jpg 150 100 185 Desc/ptch1
```

where

Ptch/img2_ptch1.jpg is the image patch (first argument),

`150, 100, 185` is the color triplet (arguments two to four),
`Desc/ptch1` is the output file stem (fifth argument).

This will write the file named `ptch1.shp` to directory `/Desc`. The example script `exsbShpx.m` runs several patches. If a parameter file is provided, it must follow the output file, before any long options are specified.

Depending on the complexity of the image patch, it may well be that the segmentation output identifies other, smaller regions of similar color in the image patch than the one aimed at. The matching program will take the largest one for matching and ignore the smaller ones.

It is also possible that no boundaries are detected, which can happen if we specify a RGB triplet that is unable to segment the patch into two groups. In that case, no shape file is generated. This will be communicated in standard output by displaying `nBon 0`, explained in more detail below.

8.1.1 Parameters and Options

Setting parameters and options is analogous to program `dscx`. An example of how to set parameter values is given with the file `Params/PrmShpx_Example.txt`. The parameter filename must contain the string `PrmShpx`. Flag `--prms` displays the parameter values used (default OFF); flag `--prmchg` displays the parameter values read by the parameter file (if provided).

8.1.2 Output (`.shp`)

The shape file with extension `shp` is the only file output. For other types of file output, one can deploy program `sgrRGB` (Chapter 10).

The program also writes standard output that informs what features were extracted, ie. as follows:

```
nBon 1
nCrv 1
umf 1.245
```

where `nBon` is the number of boundaries detected, `nCrv` is the number of curve partitions detected, and `umf` is the ratio between the boundary perimeter and the patch size (circumference) for the largest detected boundary.

This information allows the user to judge the success and result of the process. As pointed out above already, if no boundaries are detected, then this will appear as `nBon 0`; consequently no shape file is saved, and we may want to readjust the RGB triplet and rerun the process. If multiple boundaries are detected, we may deal with a texture. If value `umf` is larger one, then the shape is indented (concave), such as in a star shape or 'H' shape.

8.2 Program Patch Extraction [`ptchxL`]

The binary `ptchxL` helps extracting patches of appropriate size from the image. Because bounding boxes from proposals are tight, ie. from `slc` or `qbbx` files, we need to enlarge them for better results. This is done automatically with binary `ptchxL`, where for each specified bounding box, a margin proportional to its size is added. By default the proportion value is 0.1 (10 percent), but can be changed.

The program takes an image as first argument, an list of bounding boxes as second argument, specified in a text file called `PtchBbx.txt` here

```
> ptchxL Imgs/img2.jpg PtchBbx.txt Ptch/P 0.2
```

The third argument specifies the basename of the output patches. In our example it will write the patches to directory `/Ptch`, named `P` using zero-indexing, ie. `P0.png`, `P1.png`, etc. As implied, the patches are written in *png* format.

The fourth argument (optional) specifies the margin proportion, that will be added to the bounding boxes, a value of 0.2 in this example.

If a bounding box lies near the image border within the specified margin value, the program will crop the patch at the image border, that is no extra margin is generated if there is none available.

The example script `exsbPtchx` demonstrates how to apply the program.

Chapter 9

Shape Matching [/ShpMtch]

The shape matching program mshp correlates shape descriptions as generated with program shpx (as introduced in the previous chapter). For a pair of shapes, three groups of measures are calculated.

1) descriptor distances: arc, straighter and radial shape. The metric includes the angle and position attribute by default, but can be turned off by providing a parameter file containing prefered weight values.
2) spectral differences: one for the bowness spectrum and one for the straighter spectrum. This information is independent of orientation and position.
3) ratio of sizes: perimeter, height of bounding box, width of bounding box.

The measures will be outputted separately and the user can combine them to an ensemble as desired.

For the moment there exists only the one-on-one version, mshp1, whose usage is analogous to the vector-matching program mvec1.

9.1 Program Use [mshp]

Analogous to the vector matching program mvec1, we specify two files, in this case shape files with extension shp:

```
> mshp1 Desc/A.shp Desc/B.shp
```

A parameter file can be specified, whose filename must contain the string PrmMshp. The example script exsbMshp1.m gives a simple demonstration of how to provide the parameter file and how to read the measures from the standard output.

9.2 Output

There exists only standard output. It displays the three groups of measures in three separate lines, ie.

```
mes 1.681 4.189 0.008
rts 0.847 1.033 0.969
spk 0.104 0.104
```

In detail, the values are as follows:

- `mes`: the three values correspond to the arc distance, the straighter distance and the radial-shape distance. They can be combined to an ensemble as desired. Multiplication is a safe and uncomplicated way to obtain a reasonable prediction accuracy. A weighted sum may perform better, if the appropriate weights can be found.
- `rts`: the three values correspond to the ratios of the perimeter, height and width.
- `spk`: the two values correspond to difference of the bowness and straighter spectrum.

Chapter 10

Demo Segregation RGB
[/DemoSgrRGB]

The program `sgrRGB` segragates a color image into two groups according to a specified RGB triplet. Its purpose is to obtain a more precise outline of an object or scene part, than obtained by the region segmentation process (in `dscx`) that occurred in gray only. For example, we have detected a specific shape in the descriptor output, ie. in `rsg`, `shp`, `ttg`, etc. Since we know the approximate color of that shape, we can use it to cue the color-segregation process, and we obtain a more precise shape outline. The target color is assumed to represent the foreground; then, the segregation process automatically calculates a distractor target, assumed to be background. The process then performs a 2-Means clustering with a single pass (iteration). The segmented shapes of the foreground regions are then partitioned and described as it is done in program `dscx`, but only for the original resolution (no pyramid is generated, no divisive segmentation takes place).

To improve and accelerate the segregation process, the image should be cropped to the approximate size of the target object (or scene part). That is, the process is rather applied to only part of an image, called *patch* hereafter. The program will determine a distractor triplet using the image border pixels and for that there are different initializations possible.

The example script `exsbSgrPtch1.m` demonstrates how to run the program and how to load the output files. The script `exsbSgrPtchInit.m` compares the output for the different initialization techniques.

10.1 Program Use [`sgrRGB`]

The program `sgrRGB` takes a (color) patch as input and a RGB triplet, ie. {150, 100, 185}:

```
> sgrRGB Imgs/ptchX.jpg 150 100 185
```

This will generate the following files (there is no option for specifying a filename yet):

`BonFore.bonPix`	boundary pixels for the target/foreground
`CrvPrt.cvp`	curvature partitions (arcs and straigther segments)
`Mlab.mpu`	black-white map, with white the foreground
`Ifore.png`	the foreground regions in average color
`BonBack.bonPix`	boundary pixels for the distractor/background

How those are loaded will be explained in Section 10.2. In the following it is explained what options are available.

10.1.1 Options

The following long option is available, to be specified with a double dash '--'; single letter option are not in use.

`--init`: the initialization technique that calculates the distractor triplet, specified as digit 0, 1 or 2:

- **-0** (default): uses the (image) patch borders to calculate the average RGB value. The complexity corresponds to the number of border pixels: $O($`nPxBorder`$)$.
- **-1**: determines the farthest distance between target and each border pixel. The complexity is the same as for `init=0`: $O($`nPxBorder`$)$.
- **-2**: measures the distance of the target triplet to black $\{0,0,0\}$ and white $\{255,255,255\}$, and takes the farther of either as the distractor. This has lowest complexity as it calculates only two distances: $O(2)$.

For small patches with homogeneous regions, such as a character shape in text, there is little difference between the different types of initializations. The larger the patch and the more heterogeneous the colors, the greater the differences. Since the example script runs the program on an entire image, one can observe substantial differences between different initializations.

10.2 Output

The `.cvp` file is loaded by routine `LoadCrvPrt.m` in `/DescExtr/Curve`.

The `.mpu` file holds the size of the map in the first two integer values, height and width. The remaining values are of type `unsigned char` and describe the map values. It is loaded with Matlab script `LoadMapUch.m` (in `/AdminMb/Util/FileIO/`). The `BonPix` files can be loaded as explained in the Appendix C.

Chapter 11

Demo Place Recognition

[/DemoPlcRec]

The following demo starts with an example of place recognition, that we then modify to use focii and shape extraction, which so gradually becomes an approach for general scene recognition. Firstly, we perform matching with the whole image (Section 11.1) and explain how to setup a cascade identifier (Section 11.2). Then we move toward the use of focus selection (Section 11.3): this is done first by using generic image partitions called *zones* here. Then we provide scripts for the use of the object proposals in the saliency file (Section 11.4). We take that a step further and also extract and match shapes (Section 11.5) using programs `shpx` and `mshp1`. Finally, we exploit the description files also to measure the motion between frames in order to obtain an estimate of ego motion (Section 11.6).

Material and Procedure The demo for place recognition uses frames from the Living Room collection. The original set contains 32 frames from a roaming robot, once taken at night and once taken during the day; the original frames have high resolution. We use only five frames of that set from the day condition, downsampled to lower resolution for simplicity: the first four and the seventh frame (image names 000000**0** to 000000**3**, and 000000**6**). Thus, comparing the first frame (no. 0) to the last (no. 6) should show the largest difference; comparing the first frame to the second - or any other interframe distance -, should result in the smallest difference. This is demonstrated in the subsequent Sections with various matching approaches. Since we compare the frames within one run (of one condition), our matching process is less an actual place-recognition test (as carried out in benchmarking), but rather one that reflects loop closure detection in robotics.

The entire demo can be run through script `plcAll.m`, `plcAll.py` respectively; both scripts are located in folder `/DemoPlcRec`.

Script `plcDscx` generates the descriptor files for the five frames. It uses the parameter file `PrmDesc_Gerust.txt`, that contains those parameter values, with which we obtained the best place recognition performance in our studies. The parameters that were used for the four collections for benchmarking are located in folder `/PlcRec` of `/DescExtr/Params` and `/MtchVec/Params`. Since those parameter values were determined ad hoc, there are chances that one can obtain better recognition accuracies by a systematic search, ie. placing the system in a loop that tests gradual parameter changes.

11.1 Whole Image

Script `plcMtchImg` performs vector matching using `mvec1`. The comparisons are made between the first frame (0) and the other four, as well as between the second (1) and the last (6). The labels therefore read '0-1', '0-2', '0-3' and '0-6' for the first four comparisons, and '1-6' for the last comparison. As expected, the maximal distance peaks at '0-6'; correspondingly, the lowest similarity is at '0-6'.

Script `plcMtcHstKol` performs histogram and kolumn matching, whereby here the first frame is compared to itself and the remaining four, hence the labels '0-0' to '0-6'.

Image Resolution The matching results with vectors (`mvec1`) are tendentially better for lower resolutions, around 300x300 pixels. This was observed with the current parameter settings. Again, a systematic testing of parameters can perhaps find values, that result in better accuracies for higher resolution.

An alternative is to deploy the descriptor extraction to partitions of the image, akin to spatial histogramming, and then match the corresponding partitions. This has worked well for some databases, in particular when the camera uses a fisheye lens. Ideally, we would implement this using focus selection or shape description, in order to accelerate the recognition process and improve its accuracy. That is the goal of the following sections, but first we explain how to form the cascade.

11.2 Cascade Identification (Whole Image)

The identification process can be accelerated by preselecting candidates with histogram matching followed by matching vectors of those candidates.

$$\texttt{all images} \xmapsto[\texttt{mhst}]{\textit{fast mtch}} \texttt{preselection} \xmapsto[\texttt{mvec}]{\textit{fine mtch}} \texttt{final selection}$$

This is a multi-stage identification process, we call *cascade identifier*. We use the term *fast matching* for any type of preselection that identifies candidates for vector matching; we could also match kolumns for example. Vector matching is then called *fine matching* in this context. Such a two-stage classifier not only accelerates identification, it can also improve accuracy. The cascade identifier is implemented with routine `CASCIDF.m` (in directory `/AdminMb/Cascade`). An example is given with `plcCascIdf.m`.

The routine `CASCIDF.m` contains a switch for fast matching choosing between preselection with histograms and with kolumns. Only the former has been tested so far. After fast matching is carried out with `f_CmndExec`, we load the sorted array and select the number of candidates `nPre` from the ordered indices in `OrdHis`, shown here for the case of histogram matching:

```
[OrdHis DisHisOrd] = LoadSortFltTxt( fpMesHst, nImg );

nPre    = round ( nImg * Prm.prpPre );
OrdPre = OrdHis( 1:nPre );
```

Parameter `Prm.prpPre` is the proportion of images we preselect. Typically a value of `0.5` works well for small environments. With the preselected indices we then preselect the corresponding file names from the description files (`dsc`), here `Rgst.aVec( OrdPre )`, and save them to the text file to be read for fine matching with `mvecL`.

```
SaveFipaLstPrependPath( Rgst.aVec( OrdPre ), Rgst.pth, fpaRgstVec );
```

Eventually we reorder the fine matching indices according to the preselected indices, carried out with routine `uu_CascSortReorder`.

Variations There are a number of variations one can test. One is to use kolumn matching for preselection, as it is setup with the switch `strcmp( Prm.stgy,'kolm1st' )`. Another is to combine the matching results from histogram and kolumn matching. Another variation would be to use a three-stage cascade, in which we start with kolumn matching to preselect candidates, followed by histogram matching to subselect candidates, that we eventually use for vector matching. More variations can be thought of.

11.3 Focus Selection with Zones

As explained earlier, the goal of focus selection is to make optimal use of the rich information in the description file (`dsc`). We do this here for image partitions of same size, called zones here, akin to spatial histogramming.

Script `plcFocZon` extracts focus descriptions for a grid of zones. This is carried out with the binaries `fochst1` and `focdsc1`. We then instantiate two slightly different versions of focus matching. One occurs by matching the focii one-on-one with binary `mvec1`. The other occurs by matching an entire list, binary `mvecL`.

Matching 1-on-1 Script `plcMtcZon1o1` matches the zones using binary `mvec1`, whereby we match only the corresponding zones, ie. most left with most left only. For the histogram difference, the Matlab script uses an improvised histogram difference. For the distance measure, the '0-6' combination is the largest as anticipated. For the similarity measure however, the '0-6' does not provide the distinctness as expected.

Matching Lists Script `plcMtcZonLst` matches the zones using binaries `mvecL` and `mhstL`, whereby in this implementation, the zones are matched pairwise for each image, simulating the case that the (exact) spatial location is unknown. We therefore create a distance and similarity matrix (`DMhst`, `DMvec` and `SMvec`), from which we then determine the nearest neighbor for each focus of the other image, e.g. for histograms:

```
DisHstNN(c,:) = min( DMhst, [], 2 );
```

The nearest neighbor measures are then combined to an image-to-image measure, for instance by summing across the zones:

```
DisSumHst   = sum( DisHstNN, 2 );
```

In this script, we include a comparison of the testing image with itself, labeled '0-0'. For the distance measure the '0-6' pair is correctly recognized as the two most dissimilar scenes. The similarity measure is plotted on a logarithmic scale. The similarity value for self-matching ('0-0') is non-zero due to the challenge of defining a similarity measure when no descriptors (of one type) are present. Again, the dissimilarity for '0-6' is lower than expected.

That the dissimilarity measure is less accurate for focus matching than for whole-image matching, highlights the challenges for this type of matching - if the goal is accurate self-localization: it requires further consideration into an efficient focus-to-focus metric.

Note that in this code example, the zones are all of same size and can therefore be matched mutually by `mvec`. If the zones were of different sizes, in particular the number of levels extracted (`nLev`), then we can match only the corresponding sizes, because `mvec` takes only descriptions of similar size. This is demonstrated in the upcoming section.

11.4 Focus Selection with Proposals

The goal here is to replace the zones (of the previous Section) with proposals from the saliency file (`slc`) or the space-consistent proposals from the proposal file (`qbbx`). Again, one could apply binary `dscx` to those proposals and perform matching on the output of those image partitions, and that is certainly worth trying. Here we deploy the focus selection binaries in order to make optimal use of the rich description of the whole image (the `dsc` file).

Selection The script `plcFocProp.m` demonstrates how to deploy the proposals for focus selection. It selects as proposals the general shapes from the proposal file (`qbbx`). But we could also use those from the `slc` file, or even combine them; or include proposals from contours, ie. `ACNT`.

Matching Then we match those focii with script `plcMtcProp.m`, which is an adaptation of `plcMtcZonLst.m` (previous Section). In a first step we match the histogram files of the focii and place the individual matches into matrix `DMhst`, which is as in `plcMtcZonLst.m`, except that here this is carried out in a separate loop. Then we carry out vector matching, which now requires matching the focii of corresponding levels.

This is currently implemented as follows. We loop over the levels and match the two lists of focii (for one level) in a routine called `MFOCTOFOC.m` (matching focii to focii). Routine `MFOCTOFOC.m` returns a single measure as opposed to all nearest neigbor measures (as was done in `plcMtcZonLst.m`): one measure for the distance metric, one for the similarity metric, and one for the spatial relation of the bounding boxes. Those are collected in arrays `BoxLev`, `DisLev` and `SimLev`, respectively, in the main script, which then are combined to a single image-to-image measure.

Routine `MFOCTOFOC.m` uses the following spatial preselection. It firstly determines the spatial relation of a pair of bounding boxes. If the pair is near, then we match the two lists of vectors. To obtain the spatial location of the bounding boxes, it requires reading only the header of the focus file. That header info is provided to routine `f_FocToFoc.m`, in which now the spatial relation is determined, by measuring congruence and spacing of the two bounding boxes. If their combination shows a reasonable value, then we match the two lists of vectors with `mvec1`. From the measurement matrices, we obtain various level-metric measurements with function `f_MtrFromMM.m`. Those level measurements are fed to the level arrays (in the main script), `DisLev` and `SimLev`, respectively.

With the current parameters, the matching process cannot meaningfully compare between image no. 0 and no. 6 as they are so different. We therefore assign an arbitrarily high value to the distance measure, and zero to the similarity measure, called `valNoMatch` in `f_MtrFromMM.m`.

Whether this type of matching can provide the same position accuracy as whole-image matching remains to be evaluated, see also the suggestions on methodological fusion in Section 12.4. But for the development of an active vision system that explores an environment, this is an optimal starting point.

11.5 Shape Extraction and Matching

Now we carry out shape extraction and matching based on shape proposals using the binaries `shpx` and `mshp1`. This runs analogous to the focus-matching scheme of the previous section.

Extraction The script `plcShpExtr.m` demonstrates how to deploy the proposals for shape extraction. It firstly extracts the patches using binary `ptchxL`, for which we save the bounding boxes first. Both the file with the bounding boxes and the patches will be overwritten with each new processed image.

Then we carry out the shape description with `shpx`. Since the binary `shpx` requires an RGB triplet as color cue, we also load the attributes

```
[QDsc] = LoadDescPropAtts( fpDsc );
```

and retrieve the corresponding colors

```
RGBchn = QDsc.ShpGen.RGB;
```

Matching Matching is carried out in script `plcMtcShp.m`. Routine `MSHPTOSHP.m` matches the two lists of shapes using `mshp1`; the procedure is analogous to routine `MFOCTOFOC.m` (above). The output of the shape matching program (`mshp1`) is combined by multiplication:

```
dis     = prod( [Msv; MxRts; MxSpk] );  % ensemble measure
```

that is all 8 values are used to form an ensemble decision.

In this matching example, no position information is utilized at all; an optimization into that direction would certainly make sense.

11.6 Ego Motion

Since we generated the description files for each frame, we can also conveniently link their descriptions to motion vectors using `motvec`. The example script `plcMotEgo.m` computes those vectors, that then are immediately loaded to Matlab, to cell `AVec`:

```
AVec{i}     = LoadMotVec( lfp );
```

Since the descriptors were extracted without saving the full set of contours (`dscRRE`) in `plcDscx.m`, the motion estimates are carried out with the skeleton set only. Better estimates might be obtained by including the full set, in which case one needs to run the descriptor extraction process with option `--saveRRE`, see again Section 5.3.

11.7 Recognition Continued

The demo scripts can be understood as code and algorithm templates for further development, in particular the focus- and shape-matching routines `MFOCTOFOC` and `MSHPTOSHP`, respectively. To build a complete recognition process, one would kick off the evolvement with the cascade identifier `CASCIDF`. Then we could further sbuselect by matching the bounding boxes, of either shapes or contour blobs, or both, which has not been implemented yet.

`CASCIDF` $\rightarrow$ `mtch bboxes` $\rightarrow$ `MFOCTOFOC/MSHPTOSHP`

With those subselected candidates, we would then start matching focii and shapes. This could well bypass the lack of actual texture recognition. If one however wanted to improve the identification of scenes sets that contain similar structure but different textures, a combination with the methodology of Local Features is suggestive, discussed later in Section 12.4.1.

Chapter 12

Applications

The demo for place recognition has explained the possibilites of constructing a cascade identification process and a scene recognition process. Analogously one could create such a cascade for categorization. In the following, we explain how we can deploy the suite for navigation (Section 12.1), active vision (Section 12.2) and some other tasks (Section 12.3). Finally, we discuss possibilites of a fusion with other methodologies (Section 12.4).

12.1 Recognition for Navigation

For an efficient navigation through an environment, it is beneficial to detect vertical and converging structures. Contour information offers a fast access to distant and thin structures, region information often holds converging lines better than contours.

Vertical Structure In road scenes, there exist often thin structures at distance, ie. lamp posts, traffic sign post, poles, etc. Those appear as ridge, river or edge contours, whereby ridge and river contours conveniently delineate a structure's axis. A fast way to obtain hypotheses is to deploy the contour skeleton `ACNT` of the vector file and integrate across the pyramid to find strong hypotheses. Since the skeleton is already a reduced contour set, there exists the risk of misses. A more thorough way would be to use the full RRE set, which can be loaded from the `dscRRE` file.

In indoor scenes, structures are nearer and have larger width, ie. walls, door frames, furniture, etc. In that case, we firstly focus on region boundaries and their descriptions, such as radial descriptors `ARSG`, partitioned-shape descriptors `ASHP` and tetragon descriptors `ATTG`. In particular the latter gives us an immediate understanding of the geometry of our surround; many of the attributes were designed for that, many more could be developed. The proposals generated with the proposal file (`qbbx`, `qdsc`) are a first step toward that direction, specifically the bounding boxes in `ShpVrt` (Section 4.2.5).

Converging Structure Converging lines often exist as boundary segments of regions, that represent scene parts. In indoor scenes, those segments are often part of the shapes as described in list `ASHP`. As explained, `ASHP` contains by default only the inside shapes, ie. those not touching any image borders. When no inside shapes are present, as is the case in a wide-open road scene, then we include those touching the border by increasing the parameter value bordTouches. Should this still not provide candidate segments, we then analyse the full set of straighter segments, loaded with routine LoadCVPfull.m (turn ON saving for full set with long option --saveCVP).

Road Surface The region segmentation process easily returns the regions corresponding to the pavement. The segmentation process is so sensitive, that it will also return any pavement patch with slightly different color, ie. originating from road repairs or from frequent traffic such as the rills from truck wheels. The latter is often described by ridge or river contours as well.

The challenge with this sensitive output is to distinguish between task-relevant and task-irrelevant regions. For an analysis of road surface with respect to potential damage, any region of the surface might be relevant; for driving, we want to discriminate between non-obstructive patches and those, that pose a potential danger such as a slippery surface, spilled oil, etc. Since the descriptors are generated for higher contrasts by default, it is likely that some low-contrast regions are not described and therefore not available as radial shape rsg or partitioned shape shp. And since those descriptors probably are not sufficient for an exhaustive characterization, it is best to analyze the whole set of boundaries, available in file BonPix (Section C.4).

If the scene is a dirt road, whose tracks (from previous vehicles) appear nearly equal to its surround, and therefore lacking border candidates from regions, then it is useful to focus on ridge and river contours, ie. using the full set in the dscRRE file. In this case, the use of the scale space might be better suited (Section 2.1.2); this facilitates finding contiguous ridge and river contours, albeit at the risk of obtaining contours that connect too much structure - as is the case for edge detection for different scales.

12.2 Active Vision

In active vision we are confronted with a much larger range (or space) of visual input than that is present in photographs, because photographs are only selected snap-shots of an endless visual continuum; they represent only a small subset of our entire visual surround. When we take a photograph, we move the camera such, that the motif lies in the center of the image, and then adjust the zoom to obtain sharp contours. Photographs mostly depict structure in easily recognizable views, so-called *canonical* views. Many image collections depict mostly canonical views and were carefully prepared. Views, that take longer to be recognized, are called *non-canonical*.

Analogous, when we interact with our surround, we place the focus on objects in order to process the visual structure according to our goals. We browse our environment to lock in on canonical views for recognition and for perception for action, that is, most of the time we wade through non-canonical views until we find those canonical views we feel familiar with for interaction. During this browsing we continously try to classify and identify.

To efficiently mimick this browsing behavior, we start with a triage during which we observe the texture statistics in particular (Section 12.2.1). Then we proceed to operate with blobs or shapes that in turn can lead us to canonical views (Section 12.2.2).

12.2.1 Triage

The triage conducts a separation between texture and non-texture. A non-texture is most often a scene, but may also show a blank field, such as the texture-free wall of an indoor environment, the open entrance to a dark room, or an empty sky outdoors, etc. The saliency file serves well as starting point for this triage (Section 4.2.4). Its content can be used to gauge whether we have the appropriate zoom and if there is an object or scene part present in the image. We recall that we can load the `slc` file with:

```
[Txa Shp Ens Dsc] = LoadDescSalc( fipaSalc );
```

To discriminate between blank view and 'some texture', we deploy data structures `Txa` and `Dsc`, in particular `Txa.Gst` and `Dsc.MxRngRR`. We observe the ridge-river contrast value for the first level

```
Dsc.MxRngRR(1)    (or Dsc.MxRngRR[0] for zero-indexing as in C or Python)
```

in order to adjust the zoom. Simultaneously we observe the numerosity and blankness bias:

```
nums = Txa.Gst.PrpPres(1)    % numerosity
blnk = Txa.Gst.PrpPres(2)    % blankness
```

The demo script `exsbSalBlobs.m` displays those values for two different images, one full of texture, the other void of it.

That texture information can be taken to deploy an appropriate classifier on the histogram file (`hst`), e.g. one trained solely for textures, which will be deployed when value `nums` is high; another classifier for scenes void of it, which will be deployed when value `blnk` is high; another classifier for indoor/outdoor classification if neither value is high, etc.

Based on this information we may decide to perform a change of view, a saccade, in particular when no or little structure is present. Or we may zoom in or out, if the camera faces texture only.

12.2.2 Scenes

When a view contains shapes and blobs of certain size, we can apply scene classifiers on the `hst` file; or we may attempt to identify the scene as in the cascade identifier (Section 11.2). If that does not result in a clear assignment, we plan the next saccade by analyzing the statistics of descriptor occurrence, ie. we observe the spatial histograms of the `hst` file. Or we analyze the ensemble information in data structure `Ens`, which is a mixture of the shape and blob information in data structures `Txa.Blb` and `Shp`. For example, we can determine their eccentricity from their bounding boxes, see the code lines above for small object recognition for how to retrieve the boxes (Section 12.3.2). Toward that goal, one could also analyze the proposals as suggested with the proposal file, see example script `exsbProposals.m`.

Apprehending a Novel Environment In case we let a robot explore a novel environment, one could store multiple views for each spatial location. That would result in a massive amount of visual description that is too large for efficient recognition and could easily reach memory limits. During later revisitation, one would deploy histogram matching (mhst, mkolL) for fast candidate identification, be it either for self-localization or view reidentification. Only for the purpose of verification and action, one would deploy vector matching (mvec) to exploit the precision of the description. The precise description of views, that is used rarely, would be moved into the background, perhaps even eliminated if memory is limited.

12.3 Other

12.3.1 General Object Recognition

Object recognition is naturally carried out with the attention processes focus selection and shape description (Section 1.2.1). An example was essentially given with the demo for place recognition, scripts plcFocProp/plcMtcProp and plcShpExtr/plcMtcShp, respectively, that carry out identification of structure. To adapt that scheme to categorization one would apply first histogram classification instead of matching. In the section on histogram matching in plcFocProp we would then first apply binary h2arr to the histogram file,

```
        cmnd    = [ FipaExe.h2arr ' ' hsf1 ' ' fpHsf1ArrTmp ];
```

whose output, `fpHsf1ArrTmp` + harf, is then fed to a trained classifier. From the posteriors we then select candidates for vector matching.

The classification stage can be carried out with a single classifier trained on annotations of arbitrary sizes, or on several classifiers trained on annotations of (roughly) corresponding sizes, ie. using `nLev` classifiers. The identification stage however must be carried out with annotations of corresponding size, exemplified with a loop over `nLev` levels.

12.3.2 Small Object Recognition

Small objects in isolation are easily detected by searching for clusters of ridge, river and edge contours, which is carried out with the texture analysis (Section 2.4). The results of the texture analysis are loaded with the saliency file (Section 4.2.4). The first output argument, named Txa in our examples, contains a data structure called `Txa.Blb`, which in turn holds the statistics for the various texture biases. In particular the bias for numerosity, Num, and the one for high contrast, often outline small objects in isolation. The corresponding candidates can be identified using the type variable `Typ`:

```
    Bnum     = Txa.Blb.Typ==1;         % numerous
    Benk     = Txa.Blb.Typ==8;         % high-contrast
```

which can be used to access the bounding boxes in `Txa.Blb.Box`, ie.

```
   BboxNum = Txa.Blb.Box( Bnum, : );
```

The example script exsbPlotSalc.m had already demonstrated this. Directory /`Demos` holds an example script exsbSmlObjDet.m, that demonstrates this for two displays made of small targets. The example makes use of the parameter adjusting the window size for texture integration (--txws).

The less isolated an object occurs in a scene, such as a drone flying near a tree silhouette, the more we need to know about its characteristics to discriminate it from its surround. We then build a classifier that operates on contour information, as a first step, that will help to eliminate unlikely candidates. We increase the specificity of the classifier by including also information from radial descriptors and perhaps even partitioned-shape descriptors (`shp`), from focii, from shape extraction (`shpx`), or applying the entire descriptor extraction process to solely that image part.

If an object is heavily occluded - quasi camouflaged -, such as a traffic sign covered by leaves, then we need a Deep (Network) Detector, that excels at integrating, dispersed local information. Applying a Deep Detector to the entire image is of course expensive, and also struggles with low-contrast situations, but the methodology introduced so far enables to apply the detector more specifically and therefore to accelerate its use. Ideally, one would combine the Deep Network methodology with the presented feature output, see Section 12.4.

12.3.3 Anomaly and Change Detection

Anomaly Detection The segmentation process is predestined for anomaly detection, because it segments anything, irrespective of contrast and shape. A transparent, plastic bag floating across the road pavement is easily detected, as well as different pavement colors as pointed out already above (Road Surface). The curve partitions (arcs and straighters) also allow to understand the structure of the anomaly. If the location of the anomaly can be determined, it makes sense to apply the segregation process `sgrRGB` (Section 10) or to apply shape extraction (`shpx`).

Change Detection The entire feature extraction output is predestined for change detection due to its thorough topological analysis. For short-term changes, the 'tracking' of ridge and river contours may help, in which case one can utilize program `motvec` as it matches the full RRE set (if the RRE set was saved). For long-term changes, we seek a robustness to luminance changes, for which then the matching program `mvec1` is more appropriate (Section 5.1). For either time scale, the use of focii and shape analysis could be beneficial (`shpx` and `mshp`).

Text Recognition in the Wild Letter shapes will appear as boundaries and can be discriminated from other shapes (non-letters) using the radial-shape descriptor (`rsg`) and the partitioned-shape descriptor `shp`, in a first stage. In a second stage, we can deploy shape extraction and matching (`shpx` and `mshp`) to refine the discrimination between letters and non-letters. But deploying an OCR network directly after the first stage, might be sufficient for fast recognition.

12.3.4 Collecting Annotations

The output of the region segmentation process (Section 2.2) can be deployed to collect objects with sharp boundaries. To understand that, it is best to run the demo programs in directory /`DemoBaum`, or the demo script `exsbPlotRegPix.m`.

Directory /`DemoBaum` contains two binaries: one for gray-level segmentation, called `baumgrau`, and another for chromatic segmentation `baumfarb`. The two program binaries can be run through example scripts `exsbBaum1grau.m` and `exsbBaum1rgb.m`, respectively.

Since many objects correspond to one or few regions, it is much easier to collect annotated material by clicking on regions, instead of outlining them with multiple clicks. The explicit specification of a complete bounding box is only necessary for colorful objects that appear in front of complex backgrounds, a situation that is equivalent to camouflage. Even in such situations, we gain sharp part and object boundaries. Using the program binary `sgrRGB`, one can obtain the sharpest region silhouettes, because it focuses on one RGB triplet by specification.

12.4 Methodological Fusion

In certain scenarios, other methodologies outperform the present methodology with its current parameter settings. For fast success, it therefore makes sense to combine them, as was suggested already previously. Since structure is well expressed and identified with the present methodology, the other methods can be deployed in a much more specific manner. Here we summarize potential approaches.

12.4.1 Local Features

Local features, such as histograms of gradient values, excel at identifying scenes in very large environments, ie. place recognition in the world. Those features are often sampled randomly from an image. With the contour and region features provided here, one can test a more directed approach to apply them, ie. taken at proposals as provided by the saliency file (`slc`).

For example, one could take histograms of gradients at specific locations in the image, such as the regions provided in the saliency file (`.slc`), or of the contour skeleton (`ACNT`).

12.4.2 Deep Networks/Learning

We firstly discuss the use of convolutional neural networks (CNN), that typically take pixels as input, also termed end-to-end learned. Then we mention the use of those networks that take processed input. Finally, there are possibilities to deploy Graph CNNs.

Pixels as Input (CNNs) CNNs are particularly good in two situations: for discriminating complex (multi-region) objects, or subtly different categories; and for the detection of heavily occluded objects, such as a traffic sign covered by tree branches, a situation also termed "camouflaged". Since CNNs require much ressources, it makes sense to provide candidates with our structural approach in order to accelerate the recognition process. This means that one would train the CNNs according to the confusions of the structural classification. For the case of object detection, one would provide candidate locations, where the object detector is applied specifically, in order to reduce the frequency of its deployment.

Processed Input For networks that take vectors as input, such as transformers, there exists of course many possibilites to test our multi-dimensional attribute spaces. For networks that take tabular information or words as input, there exists equally many possibilities to generate them.

Graph CNNs Graph CNNs are useful for discriminating distributions in low-dimensional space. They could be applied to boundaries in order to discriminate subtly different shape silhouettes.

Appendix A

List of Program Binaries and Demos

A.1 Binaries

The program binaries are divided into the categories computation, learning and utility.

A.1.1 Computation

The following lists those binaries, that carry out computational processes.

- `dscx`: carries out feature extraction and description. It is the main program (Chapter 4).
- `mvec`: matches the descriptor vectors as outputted by `dscx`, and returns dissimilarity and similarity measurements for each level of the space and each descriptor type. It can be applied to any structure: shape, object or scene. There are two instantiations of vector-matching:
  - `mvec1`: matches one pair of image descriptions
  - `mvecL`: matches one versus many ('L' for list)
- `mhst`: matches descriptor histograms and outputs a dissimilarity measurement. This can be used to identify candidates for vector matching (with `mvec`) in a cascade classifier.
- `mkol`: matches kolumn histograms and operates analogous to `mhst`.
- `shpx`: carries out a refined segmentation for a target region using a color cue, and then describes the obtained region silhouette using the radial description (`rsg`), the arc description (`arc`) and straighter description (`str`).
- `mshp`: matches the description as produced by `shpx`.
- `motvec`: computes the motion vectors between descriptors of two frames (Section 5.3). This is essentially the same as `mvec1`, but outputs in particular the motion vectors between nearest neighbors. That allows 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 (space) level, then preselects images and progresses toward finer levels. Early experiments have shown that this strategy not only speeds up the search, but also improves the sorting. For the moment one can use the combination of mhst and mvec to accelerate recognition.

A.1.2 Learning

A learning process will be provided that determines category-characteristic descriptors by individual matching of vectors. In principle one can carry out such a learning procedure with the matching binaries mvec, but it is of course more convenient to have binaries that provide more automation.

- kkcan [planned]: searches for nearest neighbors across images of one category to find candidate descriptors. This is based on mvec1 as introduced above.
- kkgrp [planned]: refines the search and selects a final set of category-characteristic descriptors using some clustering technique.
- kkmtc [planned]: matches the category-characteristic descriptors against a new image and outputs the degree of dis-/similarity per category-characteristic descriptor, similar to mvecL.

A.1.3 Utility

The utility programs rearrange the description output for specific purposes.

- focsel: selects descriptors from a region (focus), specified by the user as bounding box. Program fochst1 extracts the histogram for one focus, program fochstL does so for multiple focii. Histograms are written to files with extention hsf. Progam focdsc1 extracts the descriptor spaces from one focus; they are placed into a file with extension dsf. The dsf or hsf files are loaded by matching programs such as mvec, mhst or knnv.
- h2arr: generates a single histogram array from the histogram file (as outputted by dscx), thus suitable for feeding directly to a traditional classifier (LDA, SVM, RF, etc.), introduced in Section 4.4.
- collhimg: same as binary h2arr, but taking a list of histogram files as input. It outputs a single matrix of size **[nImg nBins]**: number of images times number of bins. This matrix can then be used to train traditional classifiers.
- d2vmx: generates the vector matrices for the individual descriptor types to the format **[nDsc nAtt]**: number of descriptors times number of attributes. This matrix can then be deployed for clustering, ie. word formation. Introduced in Section 4.5.
- collvec: concatenates the vector matrices for a list of (image) descriptions to a single matrix of size, **[ntDsc nAtt]**, that is total number of descriptors (for all files) times number of attributes of a descriptor (Section 4.5).
- ptchxL: Extracts rectangular patches of any size from one image using a list of specified bounding boxes. This serves to prepare the patches for shape extraction with shpx.

A.2 Demonstrations

The following mentions the demonstration code that has its own directories.

- /DemoBaum contains binaries that demonstrate the segmentation process, one for gray-scale analysis (as used in dscx), and one for a chromatic analysis using the RGB channels.
- /DemoPlcRec: contains demo scripts that carry out a simple place recognition experiment using programs dscx, mvec and focsel. The script plcAll runs the complete sequence.
- /DemoSgrRGB: contains a binary called sgrRGB demonstrating the color-segregation process used for shape extraction (Chapter 10). It 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. It also outputs the arcs and straighter descriptors for the foreground regions.
- /Demos: contains various scripts demonstrating in particular parameter changes (script name contains prefix exsb).

Appendix B

Image Filtering

Some options for simple image filtering are provided. This filtering occurs only for the original image resolution, before the image space `IS` is generated.

Three degrees are available, specified by number with long option `--if` or with string `imgFlt` by file:

`imgFlt 1` gaussian filter with sigma = 0.5

`imgFlt 2` gaussian filter with sigma = 0.75

`imgFlt 3` gaussian filter with sigma = 1.0

Demo script `exsbImgFilt` shows how to deploy them as long option.

Appendix C

Feature Files

These files contain information about the features, their pixels and their bounding boxes. Some of them were originally developed to make a comparison with object proposal studys.

The .regPix file contains the region pixels. The .Bbox file holds the bounding boxes of regions in original map resolution. The .BonBbox contains the bounding boxes upscaled to original image resolution. The .BonAsp file contains some boundary aspects. The .BonPix file contains the boundary pixels.

C.1 Region Pixels (.regPix)

The region pixels are saved to file if long option --saveRpx is set, in which case they are saved to a file with extension regPix. The regions can be loaded by routine LoadRegPix.m, an example is shown in script exsbPlotRegPix.m.

The region pixels are specified as linear map index, in variable **IxLin**, see reading routine ReadRegPixBlok. They were written as a single array. To obtain their starting indices we have variable **Anf**. To address the pixels in the map, it requires the map dimensions, which are given with variable **SzM**.

C.2 Bounding Boxes (.Bbox)

The bounding boxes for regions can be saved by setting the (long option) flag --saveBbox. They are written in text format to a file with extension .Bbox. The list contains all bounding boxes. This allows to subselect according to the desired tasks.

The information in this file is minimal. More information is provided in the .BonXXX files (upcoming in Section C.4). But here we introduce the organization of the bounding boxes as well as the border values.

The first two integer values of the file hold the number of levels **nLev** and segmentation depth **depth** applied in the run. The following numbers hold the region count for each segmentation map, **nBbox**, saved looping levels as the outer loop and looping depth as the inner loop. The example below shows that for two levels and depth equal 3 (using zero-indexing).

```
nLev
depth
```

```
nBbox_Lev0_Depth0
nBbox_Lev0_Depth1
nBbox_Lev0_Depth2
nBbox_Lev1_Depth0
nBbox_Lev1_Depth1
nBbox_Lev1_Depth2
```

Then the bounding boxes follow. They are organized analogously to the above inner/outer loop: first all bounding boxes of [lev=0,depth=0], then those of [lev=0,depth=1], etc. A bounding box contains 6 parameters.

```
top, bottom, left, right, area, border       (for lev=0, depth=0)
top, bottom, left, right, area, border
...
top, bottom, left, right, area, border       (for lev=1, depth=2)
```

The parameters describe:

- `top, bottom, left, right`: absolute coordinates that correspond to the map size of the pyramidal level. Thus one needs to upsample them by multiplying with the corresponding factor (2, 4, 8, ...).

-`area`: size of bounding box, calculated with the first four parameters.

-`border`: number of touches with the four image sides. The values are:

```
0        no touches: off border
1-4      at one border, directions NESW (top, rite, bot, left)
11-14    at two borders, directions NE,ES,SW,NW (topright, ...)
15,16    "   "    "    , NS axis, WE axis
101-3    at three borders
200      touching all borders
```

The bounding box sizes are typically slightly too small in comparison to annotations in datasets, partly due to the segmentation procedure and partly due to downsampling. Adding margins achieves better annotation correspondence, ie. margin values that correspond to the pyramid level.

Of course one can perform better selections with more information such as boundary contrast and perhaps region attributes, which is included in the `.BonXXX` files, upcoming in Section C.4.

C.3 Contour Endpoints (`.CntEpt`)

The contour endpoints are written to file if the long option `--saveBbox` is set - it is the same option as for the file generating the bounding boxes introduced above.

The points for the contour segments consist of the two endpoints as well as their midpoint. The points are written per level, per contour type and per point type. They are saved in binary format with extension `CntEpt`.

The first value holds the number of levels. Then each level of the pyramid is written separately with firstly the points of the ridge contours, then those of the river contours and eventually those of the edge contours. The points are written blockwise (and not rowwise as in case of the bounding boxes). The first value holds the number of descriptors. Then follow first all coordinates of the first endpoint (for that level); then all coordinates of the second endpoint; followed by all coordinates for the midpoint. The coordinates `coords` are saved as row/column pairs, per point.

```
nLev
nRdg   (# of ridge contours for lev=0)
[ridge coords of 1st endpoint for lev=0]
[ridge coords of 2nd endpoint for lev=0]
[ridge coords of midpoint     for lev=0]
nRiv   (# of river contours for lev=0)
[river coords of 1st endpoint for lev=0]
[river coords of 2nd endpoint for lev=0]
[river coords of midpoint     for lev=0]
nEdg   (# of edge contours for lev=0)
[edge  coords of 1st endpoint for lev=0]
[edge  coords of 2nd endpoint for lev=0]
[edge  coords of midpoint     for lev=0]
nRdg   (# of ridge contours for lev=1)
[ridge coords of 1st endpoint for lev=1]
[ridge coords of 2nd endpoint for lev=1]
[ridge coords of midpoint     for lev=1]
...
```

As with bounding boxes, the segment coordinates are absolute values corresponding to the map size of the pyramidal level. They need to be upsampled to match the original image resolution if they are used as object/part proposals, and given some spatial width by adding some corresponding value.

C.4 Boundary Information (`.BonBbox`, `.BonAsp`, `.BonPix`)

If the flag `--saveBon` is set, then the full boundary information is saved to three separate files.

- `BonBbox`: bounding box information, similar to `Bbox` (as introduced above), but in different format.
- `BonAsp`: more boundary aspects.
- `BonPix`: boundary pixels.

The two scripts `exsbPlotBon.m` and `exsbPlotBonPix.m` demonstrate how to load those data files. The file with extension `BonBbox` contains essentially the same information as the `Bbox` file (described above under Section C.2), but in slightly different format and with additional information. The differences are:

- the bounding boxes are concatenated across depth, but the depth information is still available as the 6th parameter.
- the box coordinates are scaled to original size already.
- the contrast value for the boundaries is given as 5th parameter.

The file with extension `BonAsp` lists some additional boundary aspects:

- chromatic values red, green, blue for the pixels along the boundary, not the region inside. Thus for small regions it may not be an optimal chromatic representation.
- coverage of the boundary area, as proportion of the image/map
- border values as introduced under Section C.2 already.
- perimeter, which is given as absolute value.
- area in pixels for the connected component (not the boundary itself), thus excluding holes
- area of boundary and thus including holes.

The `BonPix` file is loaded with routine `LoadBonPix.m`.

Appendix D

Distributions, Implementation

The main package (`SEHBAU`) is available on:
`https://github.com/Sehbau/Haupt`

for the following systems, all 64 bit (x86):

Windows	`SEHBAU_win.zip`
Ubuntu, 22.04.4 LTS	`SEHBAU_ubu.tar.gz`, compiled under WSL2
Debian	`SEHBAU_deb.tar.gz`, compiled under WSL2
Fedora	`SEHBAU_fed.tar.gz`, compiled under WSL2

After downloading it, it is best to strip the suffix denoting the distribution, for example rename folder `SEHBAU_win` to `SEHBAU`. To run the administration software, we set a global path variable (Appendix E). In the following we discuss some performance aspects.

D.1 Fast Binaries

The package on `Sehbau/Haupt` provides binaries that were compiled without any optimizations. Fast binaries, that were compiled with optimizations for speed, can be found under the following site:
`https://github.com/Sehbau/BinsFast`

for the following distributions:

Windows	`bins_fast_win.zip`
Ubuntu, 22.04.4 LTS	`bins_fast_ubu.tar.gz`
Debian	`bins_fast_deb.tar.gz`
Fedora	`bins_fast_fed.tar.gz`

The compressed files contain the optimized binaries for descriptor extraction and matching (`dscx`, `mvecL`, `mhstL`). They need only to be copied into the respective folders.

D.2 Memory Limitations

Descriptor extraction can be executed for any image size and any ratio in principle. As mentioned previously (Section 4.1), for image sizes larger than ca. 320 x 320, it is expected that the image represents a regular scene. The reason is that we operate with constant memory allocation for practicality.

Specifically we assume that the image is not made of dots, which in case of an 3000x4000-pixel image would mean allocating the attributes for 3 million contour segments, thereby easily reaching memory limits, in particular when we include memory allocation for boundaries (ie. taken for a tree with depth equal 4). For large images made of fine-grained texture, the program might therefore exceed the allocated memory and crash, in particular if we use a scale space as architecture (`is = 2`).

For the databases tested so far, no image has produced a feature output that comes close to the maximally allocated memory. For example, for the CityScape collection with its 1024x2048 pixel images, the program allocates by far sufficient memory for a pyramid architecture (`is = 1`). The program also works for images of size 3000x4000 pixels, such as an image taken by a cell phone. Larger sizes have not been tested yet.

D.3 Issues

The following issues might appear:

NaN, Not-Quite-A-Number NaN entries are utilized for attributes where its definition is not applicable, ie. the orientation angle for a circular shape is set to NaN. They are however sometimes written as Not-Quite-a-Number to file in C, that I have troubles reading to Matlab or Python with the function `fscanf`. When they are present, ie. when reading the matrix values with `LoadDescVect.m`, the file is first read as text, and later converted to float values using `sscanf`.

Matlab Dos Function in Windows Matlab in Windows might not execute a program binary with its `dos` or `system` function, see subsequent Appendix E.1.

Appendix E

Administrative Code

After decompressing the package (`SEHBAU`) do the following:

- strip the suffix denoting the distribution, for example rename folder `SEHBAU_win` to `SEHBAU`.
- open script `globalsSB.m/.py` in directory /`AdminMb/Py`/) and specify the full path of the main folder /`SEHBAU` in variable `rootSehBau`.

The administrative code for Matlab is the most elaborate and contains the most comments. The code development for Python is less elaborate and often lacks comments, but contains the essential functionality.

There are two sets of example scripts. Those whose filename starts with the four characters `exsb` (example sehbau), and those starting with prefix `plc` (place recognition).

- `exsbXXX`: these scripts explain how to run a binary. The scripts in directory /`Demos` show in particular the effects of parameter variations. The entire list of these scripts is summarized (and run) in script `exsbAll.m/py`, starting with two simple scripts. For some of the scripts the order matters, e.g. one script does descriptor extraction, another plots the descriptors.
- `plcXXX`: these scripts show the application of the suite in a mock application focusing on place recognition. They are all listed in the script `plcAll.m/py`.

Should there be any difficulties with paths and global variables, it is best to work through the script that contains all the example scripts in one sequence, `exsbAll.m`, `exsbAll.py`, respectively. The early scripts in that sequence rely less on paths and global variables. Later scripts rely more on convenience wrappers and utilities. Their order of appearance matters, because we did not carry out descriptor extraction in each script.

E.1 Matlab

In Matlab we add all the paths to the search path:

```
addpath( genpath( [rootSehBau 'AdminMb'] ) );
```

The Matlab version under which the code was developed is over a decade old. No particular toolboxes are deployed as far as I can remember. A student version should be able to manage the code.

Issues Matlab in Windows might not execute a program binary with its `dos` (or `system`) function, due to the complex interaction of DLLs (dynamic link libraries). In that case try consulting:

https://de.mathworks.com/matlabcentral/answers/316233-can-t-run-external-program

or perhaps try running Matlab without the desktop, ie. matlab -nodesktop.

dos/unix/system Since the administrative code was originally developed under Windows, it (still) might use occasionally the `dos` function for system calls. Under Unix one would use function `unix`. But a better choice would be the use the system-independent function `system`.

E.2 Python

To import the package to a script, the following two lines are used:

```
sys.path.insert(0, '..')           # points to folder SEHBAU
import AdminPy as sb
```

where the first line adds the path for the main folder (/`SEHBAU`), that is usually one directory backing out in most example scripts. Then we import the tree /`AdminPy` as `sb`.

Inconveniences The Python code often lacks code comments, because we developed the Matlab code first, but did not port the comments to Python yet. Thus, for comments one would consult the corresponding Matlab routine.

The first code version made excessive use of the general type `class`, where types `dataclass` or `dict` would be more appropriate. This is being corrected.

Version The version under which the code was developed is 3.11.9. Older versions might require a different format for running a subprocess, ie. different arguments when one calls function `subprocess.run`.

E.3 Notation

Our code notation is leaned toward the Java notation that uses concatenated, capitalized words and syllables, also called camel case if the first letter starts with a small case. The underscore sign '_' is not used for variables but for function names, where a prefix denotes the type (or class) of a function or routine. For example a prefix made of a single letter, such as f_, i_, u_, stands for computation, initialization and utility, respectively. But also prefixes of multiple letters are used. Only few routines are named without any underscore, that then contain a verb (or syllable of a verb) to express 'action' (ie. `Load`, `Save`, `Read`, ...) in order to distinguish themselves from variables.

- `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.
- `o_Func`: organizational routines, such as data structures for labels, file handling, argument passing, etc.
- `p_Func`: plotting routines.
- `v_Func`: verifies data structures, command execution, etc.
- `LoadX`: loads from file with path being specified as function argument. Such functions typically call Read routines, as explained below.
- `SaveX`: saves data to file with path being specified as function argument. Such functions typically call Write routines, as explained below.
- `ReadX`: reads from file with filepointer given as function argument. They are usually called from a loading script `LoadX` (see above).
- `WriteX`: writes to file with filepointer given as function argument. They are usually called from a saving script `SaveX` (see above).
- `PlotX`: comprises a longer list of plotting instructions, calling usually plotting routines `p_Func`.
- `RennProg`: runs a program 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. script `RennDscx.m` runs program `dscx`.
- `pso_Prog`: parses the standard output obtained from a program binary/executable, e.g. script `pso_Mvec1.m` parses the output of `mvec1`.
- `exsbX`: is a demonstration script showing how to apply a program, e.g. `exsbDscx.m` runs program `dscx`. Some of these scripts call the corresponding wrapper routine `RennX`.

The following notation is used for variables:

`lowercase`	scalar values, ie. parameters
`Capitalized`	arrays, matrices, structs
`UPPERCASE`	structs of arrays/matrices/structs
`aX | AX`	array (list) of X, ie. `ALev` for levels
`nX`	number of X, e.g. `nLev`, `nRow`, `nCol`
`szX`	size of X, e.g. `szV`, `szH`, vertical, horizontal size

For more explanations on the notation see also my Computer Vision overview:
https://www.researchgate.net/publication/336460083