# Sehbau: The Software Suite

https://github.com/Sehbau

January 27, 2026

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 - and to apply the associative prowess of Deep Networks more specifical. 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

but the present document goes way beyond that, by introducing also texture analysis, saliency and object proposals.

Chapter 2 explains the feature description process; 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 12). The chapter on applications proposes how to deploy the suite for specific tasks (Chapter 13). Examples of how to access the structural description and how to administer the programs are given with Matlab and Python. The code notation is explained in Appendix G.3.

The software can be found on:

https://github.com/Sehbau/Haupt

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

| | |
|---|---|
| Windows 10 | `SEHBAU_win10.zip` |
| Windows 11 | `SEHBAU_win11.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 |
| /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 and kolumns, mhst, mkolL |
| /MtchTxt | matching texture info, mtxt1 |
| /MtchVec | matching descriptor vectors, mvec, motvec |
| /ShpExtr | shape extraction for a patch (image), shpx |
| /ShpMtch | shape matching, mshp |
| exsbAll.m/.py | demo script running all example scripts |
| runEss.cmd/.sh | shell script running the essential programs |

The shell script called runEss.cmd/sh (in the main folder) executes the basic programs without any need of other software; it serves to illustrate the use of the programs in bare form in case Python or Matlab is not available. To access the description in detail, there exists code for Matlab and Python. Their function routines lie in separate directories named /AdminMb and /AdminPy, respectively. Example scripts for both languages often lie in the same folder and their names contain the prefix exsb (example Sehbau). The example must be run from the respective folder, as we often use relative paths to the program binaries and sample 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, and it is therefore best to specify the path where the main folder lies, by setting variable rootSehBau in script globalsSB.m/.py.

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. The essential Matlab scripts have also been translated to Python, but the code comments have not been ported yet.

The script in the main folder, called exsbAll.m, runs all example scripts, starting with the above 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 how to setup the administrative code are given in Appendix G.

## 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 feature description for the entire image, executed with a program binary called dscx. It outputs a number of files, called image files here,

dscx → image files → mvec (mhst, mtxt1)

that in turn can be used for classification using a traditional classifier. Or we use them for identification 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. With programs mhst and mtxt1, we match histograms and texture, respectively.

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 focusing on certain parts of it. For that purpose there exist two processes, called *focus selection* and *shape description*. These processes can be regarded as (covert) attentional shifts.

**Focus Selection (Spatial/Region)** The process of focus selection extracts a (region) subset of the outputted description (from the image files), with an executable called focsel. The extraction is then saved to separate files called focus files:

image files → focsel → focus files → mvec

The focus files can then be used for matching with program `mvec`. The choice of what to select can be made based on the output from the image files or some other process; one merely specifies a bounding box. There is no further feature extraction carried out during this subset selection, it merely rearranges the targeted region of the image files to another file.

This focusing process allows to classify or identify arbitrary (rectangular) parts of an image separately, leading to more better hypotheses, which in turn can be exploited to build a cascaded (multi-stage) recognition process.

**Shape Description** Unlike focus selection, the process of shape description carries out more computation. It performs a fast 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 precise and more elaborate than the shape processing that takes place with executable `dscx`, because we utilize a color cue. The values for the color cue can be naturally drawn from the information in the image files or some other process if one desires. Shape files can be matched with program `mshp`.

One can also try to combine this shape description process with a change of camera direction (saccade). That can make sense if we are already certain of what colors to expect with the new view, and instead of running the entire description process, we directly target the outline of the desired structure.

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

- `dscx` [descriptor extraction]: the executable 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, ie. to decide when to zoom, when to perform a saccade (change of camera direction), etc.
- `mvec` [matching vectors]: matches the descriptors as outputted by the program `dscx` or `focsel`, and returns dissimilarity and similarity measurements for various types of descriptors, later refered to simply as metric measurements. This is useful for identification of structure. The use of this executable will be explained in Chapter 5.

- `focsel` [focus selection]: extracts the description of a desired, rectangular 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, which then are saved to files with extensions `.dsf` and `.hsf`, resp. The vectors can be matched with other focii with the program `mvec`. To be further detailed in Chapter 8.

- `shpx`/`mshp` [shape extraction and matching]: refines the segmentation of a shape given a color cue, and saves it to file with extension `.shp`, which then can be used for matching (Chapters 9 and 10). The targeted 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 executable `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` (or some other process). 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 attentional processes to the same image patch.

### 1.2.2 Introduction to Usage

The following examples give an idea of how to provide the arguments to the program binaries, which is also shown in the shell script `runEss.cmd/sh` in the main folder:

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 measurements, 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. Long options are denoted in violet, ie. `--prm`. Parameters provided by file are denoted in red, ie. `prm`. Some of the long option names are shorter than their names specified by file.

# 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.

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

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 mostly the feature description by default, that then can be used to run the other programs. The extracted features themselves are not saved due to their data volume. If one desires to access them for own manipulation, then one can save them by turning on corresponding flags. 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 `ISP`. A space consists of a stack of maps, also called *levels* here. Its height is specified as number of levels with parameter `nLev`:

```
ISP[l]    for  l = 0,..,nLev-1
```

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.

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 F.2).

### 2.1.3 General

An example script comparing the output of the two spaces is given in `exsbImgSpaces.m` in directory `/demos`. The image space `ISP` 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.

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 ISP. The descriptor extraction output will be therefore 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.

For each level of the space, 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 the feature extraction process and its selection parameters (Section 2.2), followed by the descriptor formation process and its selection procedures (Section 2.3).

## 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. From the regions we extract also their boundaries, which we also consider part of feature extraction.

### 2.2.1 Contours

Three types of contours are extracted, namely ridge, river and edge contours, resulting in three spaces that already reflect the intensity topology in enormous detail, see Figure H.1 for an example (in Appendix H). Due to this richness we refer to it as the contour universe:

```
CUV[l][t]    for  l = 0,..,nLev-1;   t for contour type
```

where `l` is for levels, and `t` is for the three contour types. To depict that space schematically, we use symbols . and | to express that the three spaces are 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  |||.|||||||.||.||      ||||.|.|||||||||       ||||||||||..||||
```

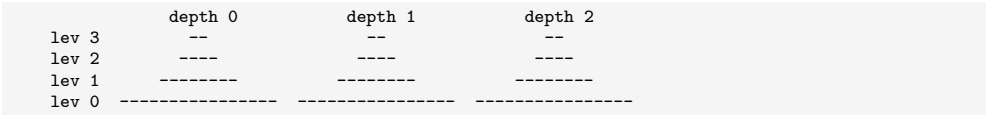
When we talk about the parameterized contours later, we abbreviate the three types as `RRE`, but the non-parameterized space of segments is called `CUV`.
The threshold parameter for accepting a pixel value as a contour pixel, is set with `Cnt.minCtr` by file or `--cntMinCtr` by long option. By default 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`. The keypoints of the universe can be saved to file, see details in Section C.1.1.

### 2.2.2 Regions and Boundaries

Regions are detected by a hierarchical, divisive thresholding process. It is applied to each level of the image space `ISP` with the same hierarchical depth. The divisive process finds any nuance in gray-shade, see how the pavement of the road scene was segmented in Fig. H.2; it finds regions of lowest contrast possible. The output is therefore vast and contains essentially all regions of slightly different color. We therefore call it the region universe, `RUV`:

```
RUV[l][d]     for  l = 0,..,nLev-1;   d = 0,..,depth-1
```

where `l` is for levels, and `d` is the dimension for depth. We depict that schematically for a pyramid architecture with four levels and depth equal three, using zero-indexing:




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.

This divisive process effortlessly segments low-contrast scenes, such as underwater scenes, night scenes, foggy scenes, etc. It also allows to deal with high-contrast scenes, often occuring when strong sunlight generates bright specular reflections in room scenes or near water surfaces. For that reason there is no contrast threshold built into this divisive process, as that readily discards potentially useful region candidates, even for the analysis of every-day scenes.

For many images, a depth value equal three is sufficient for most tasks. For images larger than $100k$ pixels, a depth of equal four could also be beneficial, in particular for low-contrast scenes. A case where depth equal five is beneficial we have never observed; it is nonetheless the maximum allowable depth possible.

If 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.

The segmentation output can be saved by turning on flag `--saveRuv`, which then generates a file with extension `ruv`. The regions can be loaded by routine `LoadRegUnv.m`, an example is shown in script `exsbRegBon.m`. More explanations can be found in Section C.1.2.

**Boundaries** For each region in `RUV` we extract its boundary pixels, resulting in what one could call the boundary universe, e.g. `BUV[l][d]`. But for practical reasons we concatenate the boundary lists across depth and thus remain only with the level variable `l`, see Figure H.3 for an example. We therefore call it the boundary space:

```
BSP[l]     for  l = 0,..,nLev-1     (dimension depth flattened)
```

The space can be saved to a file with extension `bspx`, which will include an array with the depth index. The latter allows to recreate the universe `BUV[l][d]` if desired, by segregating the list of one level (Section C.1.3).

### 2.2.3 Selection for Recognition

The vast feature output in `CUV`, `RUV` and `BSP` represents raw, unselected information, which allows to deal with any light condition or task. For tasks that analyze structural subtleties, this detailed output is necessary. For example, the analysis of material surfaces profits from the detection of low-contrast regions in later depths of `RUV`, as well as the low-contrast contours in `CUV`. Those low-contrast regions are equally useful for estimating the lighting ambience of a scene. For search scenarios, the structural detail facilitates candidate selection, ie. the mere analysis of ridge information can quickly point out salient, small spots in an image, see Fig. H.12 for an example (details to be mentioned later).

But for general recognition tasks, such as classification or identification of everyday scenes and objects, much of the detail can be irrelevant due to the near-infinite structural and apperance variability. In that case, the challenge is to select what is relevant for a given task. It then requires some sort of selection, based on parameters such as contrast, minimum size, minimum spacing, etc. As pointed out above already, this selection is difficult to perform during the feature extraction process, and risks discarding information that can be crucial. This does not mean that we do not require to readjust some parameters for a second analysis of the same viewpoint (see Figures H.7 and H.8). But it is naturally better to start out with the raw information and subselect afterwards for the desired task.

This task-oriented selection starts when we begin to characterize the boundaries, and the selection conditions will be called *entry conditions*, coming up in the next section. For contours, this entry condition is the threshold parameter `Cnt.minCtr` and had to be lowered, if one expects very low-contrast scenes. An indication for such a case, is the lack of presence of long contours (with the default value). During the feature description process, more selection will be carried out, for both contours and parameterized boundaries.

## 2.3 Descriptors

A parameterized feature is called *descriptor* and its measured parameters are called *attributes*. The description progresses gradually, starting with a simple description of contours and boundaries, followed by a more refined description that includes grouping and partitioning. We will first give an overview of this descriptor formation process (Section 2.3.1), then we overview how shape is described in general (Section 2.3.2). Then we introduce the individual descriptor types (Section 2.3.3). Following that we elaborate on the deployment of the attributes and the accompanying terminology (Section 2.3.4). Eventually we summarize how the shape description will be deployed, as it is the most complex issue in recognition (Section 2.3.2).

### 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 to parameterize 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 from `CUV` are directly parameterized resulting in the so-called `RRE` space: `RRE[l]` with `l` for level. The continued use of the term universe, ie. RRE universe, would have been more systematic, but we keep it reserved for the non-parameterized segments for clarity. Specifically, there is now one space for ridges, one for rivers and one for edges:

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

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

Boundaries from `BSP` are described in two ways. The first, a direct parameterization and is based on a variety of techniques such as a simple hull description, radial signature analysis, etc. This is called the form descriptor:

$$\texttt{boundary} \xmapsto[paramet.]{direct} \texttt{form (full)}$$

Unlike for contours, there exists no further reduction. The second boundary description carries out a curve analysis, specifically an analysis of a boundary's curvature space. This analysis partitions a boundary into a set of *curve partitions*, of which there are two types: curved and straighter segments, called *arcs* and *straighters*, respectively:

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

The result is also called the *full* set. It is then reduced to a skeleton called *gerüst* (scaffold) for the curve partitions. With the arc and straighter partitions we have a

more specific description than the form descriptor, but that will be introduced under 'Complex Descriptors'.

To clarify, this descriptor formation is carried out for each level of `CUV` or `BSP`. So is the reduction from a full set to a reduced set; it is carried out per level, not across the level dimension. The resulting reduced descriptions, such as the skeleton or gerüst, are spaces again. Reductions across space are certainly worth exploring, but have not been pursued yet for the basic descriptors.

**Entry Conditions (for Boundaries)** Unlike for contour features, not all boundaries are parameterized. There exist entry conditions that are necessary due to the vast information in `BSP`. The conditions are based on contrast and segment size and exist for admitting boundaries to the processes of direct parameterization and curve partitioning (Section 4.3.5). That is, the full sets for the descriptor types form, arc and straighter, are strictly speaking already a reduced set, but nevertheless are called as such, because they still represent a vast amount of information. They are saved to file only if a flag is set. It is only the full set for the form descriptors that is automatically saved to the `dsc` file, as its data volume 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 an 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 13).

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, coming up next.

#### Complex Descriptors and Texture Description

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

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

The curve partitions, the arcs and straighters 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 Overview Shape

We summarize the processes for shape description for a boundary that had been mentioned in the previous section. There are two principal types of shape parameter-

ization. One is the direct parameterization resulting in the form descriptor, which was defined a basic descriptor. We later refer to it also as *simple boundary description*.

The other carries out a curve partitioning process from which we obtain a spectral description and a set of curve partitions per shape. This is later referred to as the *precise boundary description*.

The curve partitions are used in multiple ways: once as individual basic descriptors as introduced previously; and once to create a partitioned-shape *descriptor*, that represents an abstraction of the arcs and straighter partitions (per shape), that was introduced as complex descriptor above.

For one shape we therefore generate the following description: the simple boundary description under 1), and the complex boundary description under 2) to 6):

**1)** a form descriptor, abbreviated `rsg`.

**2)** a spectral description (outputted by `mshp`).

**3)** a list of arc descriptors `aArc`, also called partition list. One arc descriptor is abbreviated `arc`.

**4)** a list of straighter descriptors `aStr` (the partition list). One straighter descriptor is abbreviated `str`.

**5)** a partitioned-shape descriptor, called `shp`.

**6)** a tetragon descriptor, called `ttg`. Calculated only if two loosely parallel straighter segments exist.

The two partition lists of arc and straighter descriptors are also called the *partition set*:

```
aPrt = {aArc, aStr}
```

Program binary `dscx` generates this description but will not output the spectral description (2); and of lists 3) and 4), it will output the arcs and straighters as two lists, concatenated across all partition lists of the entire level, without their correspondence to the original partition set, to be further explained below.

Program binary `shpx` generates the description for 1) to 5), which will be used by the matching executable `mshp1`. More on the deployment of this entire shape description will be given in Section 2.3.2, after we have elaborated on the individual descriptors.

As indicated, descriptors will be denoted in text color `green`. Their attributes will appear in `olive green`. For more clarification on the use of text colors see Appendix E.

### 2.3.3 Descriptor Types

We expand a little bit on the individual descriptor types and introduce those attributes that are effective in vector matching, ie. in a dissimilarity measure using the Euclidean distance metric. The first four descriptor types in the following list 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 space, denoted as RRE[l] with l for level. Based on this space, we derive three types of descriptions:

- Skeleton: a selected set of longer contours, called *skeleton* sometimes. They are drawn from the RRE space and therefore have exactly the same attributes. The skeleton is saved to the description file and appears as ACNT[l] 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.

The RRE space is *not* saved by default, due to its volume. It can be saved by turning on long option --saveRRE, in which case it is saved to a separate file with extension rre. It can be loaded with routine LoadCntRRE.m.

The reduction from the RRE space to the skeleton space 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.

We summarize the skeleton extraction process:

$$\texttt{ISP[l]} \xmapsto{\texttt{minCtr}} \texttt{CUV[l][t]} \xmapsto{paramet.} \texttt{RRE[l]} \xmapsto[global-to-local]{skeletonization} \texttt{ACNT[l]}$$

### 2) Form (rsg)

The form descriptor is abbreviated 'rsg' as it originally contained only parameters from the radial signature. Meanwhile it has been expanded by a description based on a simple hull of the boundary and other simple geometric measures.

The most effective attributes for vectorial matching are the height and width of the bounding box, followed by the region area and the orientation angle of the hull. The remaining parameters are useful for fast classification of scenes; or for the early stages of a cascaded shape recognition process.

The entry conditions for description are based on minimum contrast and minimum size, see rsgMinPix and rsgMinCtr of Section 4.3.4. More attributes to be mentioned later.

We summarize its extraction process, where entryCond. are the entry conditions for parameterization:

$$\texttt{ISP[l]} \xmapsto{\texttt{minNpx}} \texttt{RUV[l][d]} \mapsto \texttt{BSP[l]} \xmapsto[direct\ param.]{\texttt{entryCond.}} \texttt{ARSG[l]}$$

### 3) Arc (arc)

The effective arc attributes for vectorical matching are the degree of curvature, arc length and directional angle. The entry conditions for the curve partitioning process are based on contrast and size.

As mentioned before, program dscx outputs a concatenated list of all arcs. For one level this list is abbreviated ARC, also called level list; the reference to the respective partition set aPrt is irrelevant here. For a space this is denoted as AARC[l].

The full set of arc segments is reduced to a skeleton called here *gerüst* (scaffold). Only this gerüst subset is saved to the dsc file. The full set can be saved as well, by turning on long option --saveCVP (cvp for curve partition); it is then written to a separate file with extension cvpf.

In short notation the formation occurs as follows, starting with the boundary space, and skipping the inbetween formation of the partition list aArc:

$$\texttt{BSP[l]} \xmapsto[\textit{partitioning}]{\texttt{entryCond.}} \texttt{AARC[l]}\ \text{(full)} \xmapsto[\textit{global-to-local}]{\textit{skeletonization}} \texttt{AARC[l]}\ \text{(gerüst)}$$

### 4) Straighter (str)

Straighter segments lie between the arc segments in a shape. They are named in comparitive form, 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 oval's high-curvature ends (the latter extracted as arcs).

Straighter segments are parameterized by their arc length, (angular) orientation and degree of straightness; the former two are effective attributes. The entry conditions are the same as for arc description.

Analogous to arcs, the list of straighter partitions for one shape is denoted as aStr; the one for one level as STR; its space as ASTR[l].

Like arcs, only the gerüst set, ASTR[l], is saved to the dsc file. The full set is saved together with the full set of arcs to the cvpf file (if long option --saveCVP is set). They can be loaded with routine LoadCVPFull.m.

Its formation is analogous to the one for arcs.

### 5) Partitioned Shape (shp)

This descriptor is based on the segment statistics of arcs and straighters for each individual shape and is therefore more complex than the form descriptor. This descriptor 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 space. 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 discretized (binned) for the purpose of histogramming and faster matching. Those bin numbers are saved in a separate file with extension `dsb`.

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

### 2.3.4 Attribute and Descriptor Space

We specify some of the concepts that were already mentioned, with a notation that should clarify the differences. We introduce the *attribute space*, that exists once as continuous values, and once as discreted values. They are constructed for each level of the image space, which then is called the *descriptor image space*.

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

```
ATS( len , ori , str  )
```

A *descriptor instance* is then represented as a vector. By developing appropriate metrics, we can exploit this space for identification of structure, which is done in program binaries `mvec` and `mshp`. Since we here operate with the raw, measured attribute values, this is referred to as the *continuous* space.

For the vectorical matching in executables `mvec` and `mshp`, we deploy mostly the attributes as we have mentioned in the previous section. Including the full set of attributes makes sense, when a shape is cleanly segmented and does not show (or little) distortion from illumination variability.

**Attribute Space, Discrete:** When we deploy the binned attribute values as vector, the multi-dimensional space becomes *discrete*:

```
ATS[ len , ori , str  ]
```

and a descriptor instance is then expressed by a *bin-vector*. From the discrete space we can form various types of histograms suitable for fast categorization. This will be further elaborated in Chapter 3.

**Descriptor (Image) Space** Since the process of descriptor extraction is carried out for each level of the image space `ISP`, this results in a space of space. We refer to it as the *descriptor image space* `DSP`, or in short *descriptor space*:

$$\texttt{DSP[l]} \;=\; \texttt{ATS}^{\texttt{l}} \quad \texttt{for} \;\; \texttt{l = 0,..,nLev-1}$$

As already introduced in previous sections, the descriptor space for contours is called `ACNT` (or `ARdg`, `ARiv`, ...), the one for the form descriptors `ARSG`, the one for arcs `AARC`, etc.
The descriptor space with its continuous values is then a continuous space, the one with its binned values a discrete space. Program `dscx` outputs the continuous space for each descriptor type to the file with extension `dsc`, summarized in Section 4.2.1. It outputs the discrete space to the file with extension `dsb`, as mentioned previously.

**Conversion to Matrices** The attributes in those `dsc` and `dsb` files are organized per attribute, as struct-of-arrays (more explanations in Section 4.2.1). They are thus not ready for vectorial manipulation, ie. for feeding them to software that carries out clustering or classification processes, which requires a matrix format. The attribute lists can be converted to matrices with the following programs. Program binary `d2vmx` creates the matrix for the continuous vectors for one image, binary `dbn2vmx` does so for the discrete bin-vectors. Program `collvec` does that for an entire image collection. The detailed use of those programs will be explained in Section 4.5.1.

### 2.3.5 Deployment of Shape Description

The structural description of boundaries (Section 2.3.2) is intentionally quite diverse due to the endless appearance and geometric variability of boundary shapes, a variability that even exists in simple scenes. The various descriptions have advantages and disadvantages and will therefore be utilized in different ways.

The simple boundary description, expressed with the form descriptor (`rsg`), is naturally more robust to variability due to its simplicity than the precise curve description. The form descriptor is therefore more suitable for a fast classification of structure, in particular as histograms in a cascaded categorization process. It can however also be deployed for identification of larger structures, ie. scenes, using its robust attributes in an attribute space (`ARSG[l]`), as carried out with executable `mvec`. That can be exploited for a cascaded identification process.

The precise boundary description has the potential to identify better than the form descriptor, but is for that reason also more prone to geometric variability: the precision makes it challenging to develop a variability-robust metric. It is deployed in variety of ways.

In a scene with multiple boundaries, the level lists `AARC[l]` and `ASTR[l]` are matched in `mvec` with no use or reference to their origin - of what boundary they came from. These lists are matched *level-wise*. The advantage of this type of matching is the robustness to appearance variations caused by changing illuminations, such as in place recognition. Its downside is that it provides little abstraction for categories.

For abstraction, the partitioned-shape description in `ASHP[l]` (`shp`) is more suitable, but also more prone to geometric variability. Generating such a description for all boundaries leads to an excessive description that renders scene descriptions rather indistinguishable when using the identification process in `mvec`. An analogue to this situation of overinterpretation would be the phenomenon of coffee ground reading. To avoid this overinterpreation, the partitioned-shape descriptor is therefore generated only for selected boundaries (when the entry conditions are met).

**Shape Identification** When the identity of a single boundary is desired, then that can be obtained by matching the arc and straighter descriptors per shape: the two partition lists `aArc` and `aStr` of one shape, are matched with the two lists of another shape. Or to express this more concisely, we match *partition-wise* using set `aPrt`. This is carried out by program `mshp`, using the output provided by `shpx`. It also produces a measurement value for the difference of the spectra, which could be deployed for cascading.

**Matching Scenes Shape-Wise** Matching scenes by shape identification promises to be more specific than matching the arc and straighter descriptors level-wise. Such a matching process does not exist for the individual shapes as described by `dscx`. In order to pursue that, one has to extract the individual shapes with `shpx`, and then to apply partition-wise matching using `mshp` for each pair of shapes. An example will be given with the place recognition demo with wrapper function `MSHPTOSHP.m`, applied in script `plcMtcShp.m`.

Before we deploy this process of individual shape extraction, we can also exploit focus selection for generating better hypotheses, by choosing a bounding box that outlines exactly the shape under investigation. The focus selection is a subset of the descriptor space and this will include multiple levels if the shape bounding box is large enough. That in turn risks the inclusion of neighboring or inside shapes (from holes), depending on the main boundary under investigation.

## 2.4 Texture

The texture description is based on the attributes of the first level of the RRE space, that is `RRE[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, see Figures H.4-H.6 for examples. Those maps are then described globally by some simple statistics, and also by region, identifying texture *blobs*.

$$\texttt{RRE[0]} \xmapsto{histograms} \texttt{kolumns} \xmapsto[analysis]{statistical} \texttt{texture maps} \xmapsto[analysis]{stats/component} \texttt{stats,blobs}$$

The kolumns and texture maps are saved to separate files; the statistics of the texture maps and their blobs to the saliency file (`.slc`), as introduced previously. We proceed with introducing the kolumns (Section 2.4.1), followed by the texture formation (Section 2.4.2) and description (Section 2.4.3).

### 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 equals 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 parameters are determined that focus on the dominance of an orientation angle; they are also called texture *biases* (Fig. H.4):

- `Num`: total number of contours present in the window; also called the *numerosity* bias.
- `Blk`: a window is considered blank (void of texture), if only three or fewer contour segments are present; also called the *blankness* bias. 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`: lack of a dominant orientation, also called nil-dominance bias. Often corresponds to foliage in everyday scenes.

- `Vrt`: degree of vertical orientation (dominance). Often corresponds to specific texture in natural scenes, such as grass, crops, stems, etc.
- `Hor`: horizontal orientation dominance, for example uneven water surface (rippled); 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. facades in urban scenes.
- `Uni`: single (uniform) orientation dominance. This includes vertical or horizontal dominance, but excludes axial dominance as it contains two dominant orientations. 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 Description

From the texture maps, three types of descriptions are generated of increasing complexity:

- Global statistics: first-order statistics taken for the entire map.
- Texturegrams: subsampling of a map and histogramming, whose output is collectively called texturegrams.
- Blobs: analysis of the individual regions in a map.

These descriptions are calculated for most texture biases, ie. `num`, `blk`, etc. They are plotted in example script `exsbTxtrMaps.m`. We now elaborate on each description type.

#### Global Statistics

One map measure is the number of non-zero (ON) pixels present for a texture bias. For those ON pixels, the first-order statistics like the mean and standard deviation is calculated. This appears as data structure `Gst` in the saliency file.

#### Texturegrams

A texture map is subsampled to a grid of size `5x5` units. From that grid map, two types of histograms are calculated, one vertical and one horizontal, which can be thought of *band* histograms. Figure H.5 shows these descriptions.

In total, this description holds 25 cell values from the grid map, and 10 values from the bands, for each map. The description appears as structure `Grm` in the saliency file, of which the grid maps appear as structure `Grid`, and the histograms as structure `Band`, respectively.

**Blob Analysis**

A blob is a connected component in a texture map and often corresponds to an object or scene part. For each connected component we determine its bounding box 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 bounding box values appear as data structure `Blb` in the saliency file, together with a few aspects of the connected component.

The texture description can be matched with an executable called `mtxt1`, to be introduced in Section 7. This is useful for orienting during active vision (Section 13.2).

## 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 its unusual spots might exist where there is lack of texture, see Fig. H.6. If that is not the case, then we can start 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 13.4.

### 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 `DSP` (Section 2.3.4). Such proposals are more likely to correspond to an object or scene part than the presence of an isolated descriptor in space (see also Fig. H.9).

More specifically, if one descriptor type appears in the same image location in two or more adjacent levels of `DSP`, 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. 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. Proposals from contours are in the making.

# 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 concrete and is therefore suitable for building a cascaded recognition process. This was already outlined for shapes in Section 2.3.5. Here we expand to scenes or objects.

The most straightforward format to represent structure is to deploy histograms of attribute values, the bin count in discrete space. This was carried out already for kolumn formation (Section 2.4.1); histograms were also generated for texture description (Section 2.4.3). Subsequently we introduce a more elaborate version of histogramming (Section 3.1). Histograms and texturegrams 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 and texturegrams 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 `ATS` as introduced above already. In that format, a descriptor instance is represented as vector and we develop metrics to compare or discriminate different points. In continuous space, this is the most precise format and is therefore suitable for identification of structure (Section 3.2). In the discrete space, identification is less accurate, but can be carried out faster. The latter has not been fully implemented yet.

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 and texture histograms, then proceed to introduce the histograms for image and focus.

### 3.1.1 Kolumns and Texture

The kolumn histograms were formed with only the RRE contours (Section 2.4.1) since the goal was texture description. For the default 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.

Kolumn representations could be extended by histogramming also the attributes of the form descriptor. For window sizes larger than the default (16x16) this has quite some potential; for smaller window sizes it will be difficult to obtain significantly more specificity as already given be the RRE contours.

**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.

**Texturegrams** The texturegrams were formed with different bias maps. For the grid map we have 25 values, and for the band histograms 10 values. Multiplied with the number of texture biases (7) we arrive at 245 bins in total. The performance with these histograms is generally substantially better than that of the kolumn histograms for both categorization and identification. The texturegrams can be matched with executable `mtxt1`.

### 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, ie. using the binaries `h2arr` and `collhimg`.

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. The histograms are formed accumulating the attribute values of the discrete (descriptor) space `DSP`.

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. $24k$ for a 3x3 grid. The image histogram is generated with the full set of descriptors, the RRE space and the full set of curve partitions (arcs and straighters). 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 dimensionality of the spatial histograms however depends on the grid size. They can only be matched across images, if they were formed with the same gird 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`.

If one desires to take histograms for overlapping grids, then one can form those using the program for focus selection (`fochstL`), followed by concatenating them.

**Performance** Training a traditional classifier (LDA, SVM, RF, etc.) often provides good results for abstract categories, approaching the accuracies of Deep Networks that were optimized for ressources (and typically have lower recognition accuracies than full Deep Networks). 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.

**Systematic Search by Determining Quants** The choice of what bi- and trivariate histograms are formed was heuristically; some of those bins contain no values (zeros) across images. A more systematic approach to explore the usefulness of multivariate histograms is to search for unique bin-vectors in the discrete space and use those as *quants*. This is illustrated with two example scripts, that are applied to the frames of the demo for place recognition, see the scripts `plcCollVecBin.m` and `plcQntHist.m`.

The funtion `f_QntCollDty` in script `plcCollVecBin.m`, determines the unique set for a collection of bin-vectors that from this point on are called *quants*. The script also determines a hash-base for later association. In script `plcQntHist.m`, we associate the bin-vectors of one image with their corresponding quants, and then build a histogram

of quants for each image. Such a histogram would represent a multi-variate flat histogram.

### 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, form, 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 executable `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 using the continuous space. For both, the metric uses individual weights for the attributes (in most cases). More precisely, the metric uses 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 between frames of a motion sequence.

Program `mvec` matches the entire descriptor space `DSP` provided. The representation in this case can be called a space of rigid vector templates. 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. In this case we can call it a flat (rigid) vector template.

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 nu-

ances in categories, such as the subordinate categories of the Indoor-67 collection with its 67 classes of room interiors.

The 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 attentional processes of focus selection and shape matching (`focsel` and `mshp1`).

**Acceleration** Since vector matching is rather expensive, one would like to accelerate the matching process. One way is the use of a cascade identifier that starts with histogram and texturegram matching to preselect candidates (Section 1.2.1). Another is to exploit the bin-vectors and their determined quants. With the quants we can precompute their distances in order to create a lookup table, with which one can match two images quicker than using the continuous vectors. This has not been implemented yet.

This acceleration would come at the cost of decreased recognition accuracy, but would make sense in a three-stage classifier: first histogram and texturegram matching, then quant matching and eventually (continuous) vector matching.

## 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, but appears promising, since our system can identify scenes very well (see place recognition study). Since identification works so well, it is worthwile pursuing a Nearest-Neighbor analysis with the goal to connect category instances in visual 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 13.5.

## 3.4 Other Formats

We now mention other potential representation formats. One is based on exploiting the texture analysis, 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 Based

Here we exploit the texture analysis as introduced in Section 2.4. We propose techniques of increasing specificity. The simplest technique would be kolumn matching as introduced above already (Section 3.1.1). Another possibility would be to match

the texture maps as a whole (Section 4.2.3), which will be carried out later with the texturegrams, because they represent are more compact description. A more specific matching would be based on the bounding boxes of the individual blobs, which requires a metric dealing with two lists of unequal length. Both of that will be introduced in Chapter 7.

Another step more specific would be to extract the individual blobs of the 7 orientation maps in data structure `OTX` (vertical, horizontal, ...). Their boundaries could be described by a simple description such as the form descriptor `rsg`. And then we match list against list, which would be akin to vector matching of descriptors.

### 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-characteristic 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.

## 3.5 Summary

We order the implemented representation formats according to increasing complexity. The summarized dimensionality for each of those formats is for the default parameter values as explained above. If the format size is dependent on image size, then the dimensionality is given for an image size of 256x256 pixels.

**1) Texturegrams** Represents several textures biases as integrated maps and histograms, independent of image size. The total dimensionality equals 245. The description is part of the saliency file (`slc`) and can be matched with executable `mtxt1`.

**2) List of Blobs** Describes the texture blobs as bounding boxes, ca. several tens in total for an image in average. The format is independent of image size. This is also part of the saliency file (`slc`) and can be matched also with executable `mtxt1`.

**3) Kolumns** Describes the individual cell histograms of the overlapping grid for texture formation. The resulting format size depends on image size; the total dimensionality equals 14336. Kolumns are saved to the `kol` file, but only if a flag is set. They can be matched with executable `mkol`.

**4) Histogram-of-Attributes** Describes the attribute values of the descriptor types as histograms, currently mostly as univariate and bivariate histograms. The format is independent of image size, but only if the grid size for spatial histogramming is the same. The total dimensionality equals ca. $24k$ for a 3x3 spatial grid. Histograms are saved to the `hst` file and matched with executable `mhst`.

**5) Vectors** Describes structure in most specific detail as lists of vectors of selected attributes. This is partially dependent on image size, namely the number of levels `nLev` of the image space `ISP`. Vectors can be matched with executable `mvec`. They are not saved explicitly, but one can write a general vector matrix to file with executable `d2vmx`. That vector matrix contains the majority of attributes, some of which have not been exploited yet in `mvec`.

It is natural to deploy those formats in a cascaded recognition process, be it for identification or categorization. The exact pipeline of such a cascade can be task dependent; an example will be given in Section 12.2. We define formats 1 to 4 to be *statistical formats*, the vector format to be the *specific format*. The statistical formats 1), 3) and 4) are suitable for classification with a traditional classifer.

# 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 of format 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.rre | full set of ridge/river/edge attributes, used by motvec |
| img1.cvpf | full set of arc/straighter attributes |
| | |
| img1.bspx | space of boundary pixels |
| img1.cuvKpt | the keypoints of ridge, river and edge segments. |
| img1.bonBboxRaw | 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.bspx 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 F.2 for more explanations.

The options for providing paramters will be listed in Section 4.3.

## 4.2 Output Files

The directory /AdminMb/DescExtr contains Matlab function scripts to load the data files and to display the features. Example scripts will be given throughout the following sections, when we explain how to load the individual files. The directory /Examples contains some other examples for convenience:

| | |
|---|---|
| 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 |
| exsbPlotBon.m | region boundaries |
| exsbPlotBonPix.m | boundaries |
| exsbPlotSalc.m | saliency information |

### 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 space) |
| .ARSG | form 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 |

A descriptor space is 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 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_MtrxToStcArr( 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`. Routine `u_DescMxToStcOfArr.m` does this for all descriptor types, for those groups of attributes that were loaded as matrix.

The attribute labels are listed in script `o_AttsLabels.m` and are the ones for the continuous space. For the discrete (bin) space they are slightly different and are in script `o_AttsLabBin.m`.

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 `psv, psh` (deprecated `vpo, hpo`) : vertical and horizontal position. Typically the center of the feature, ie. the midpoints of contours and boundary segments; the pole for form 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.

The following attributes are used in particular for shape description, specifically the descriptor types form, partitioned-shape and tetragon.

- `Vrt,Hor,Axi`: stands for vertical, horizontal and axial. Verticality and horizontality are both considered axial. The axial attribute itself expresses the presence of both vertical and horizontal, thus structures that are tendentially square or rectangular in appearance (if in alignment with the image sides).
- `Elo` or suffix `E`: elongation, expressed as the ratio of the longer side divided by the shorter side.

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.

### Form Attributes `ReadRsgAtt.m`

The loaded data structure for form contains several sets of attributes organized into substructures. A set of attributes is read with the above mentioned routine `ReadMtrxDat.m` and the matrix is then later converted to a structure of fields with `u_MxToStcArr.m`.

The attributes for the radial signature are in substructures named `RdSig`, `BospA` and `Bglid` (first, fourth and fifth substructure). The attributes in `RdSig` express very simple statistics, such as the (scaled) radius `rds`, the elongation `elo`, the degree of concavity `cncv`, and the circularity `cir`. The attributes of the other two substructures are based on the protrusions (corners) of the radial signature. The attributes for the hull description are in substructures `RospA`, `RospB` and `RoEck`. Some of the attributes of `RospA` are particularly effective for vectorial matching:

Table 4.1: Attributes of structure `RospA` (hull-based) of which some of the bounding box (bbox) attributes are unrelated to the actual hull description.

| | |
|---|---|
| `AreN` | area of hull (normalized by image/map area). |
| `HgtN` | bbox height (normalized). unrelated to hull |
| `WthN` | bbox width (normalized). unrelated to hull |
| `HgtE` | bbox height as elongation (height/width). unrelated to hull |
| `WthE` | bbox width as elongation (width/height). unrelated to hull |
| `Hgtv` | bbox height combined with `RtAre` (below) |
| `Wthv` | bbox width combined with `RtAre` |
| `RtAre` | ratio between boundary area and hull area: voluminosity/fullness. |
| `Axial` | axiality of the hull sides. |

For scene description, five efficient attributes are the size and elongation of the bounding box in height and width (`RospA.HgtN`, `RospA.WthN`, `RospA.HgtE` and `RospA.WthE`); and the axiality (`RospA.Axial`), that is calculated with the primitive hull description. Figure H.10 demonstrates some of those attributes, generated with example script `exsbFormAxial.m`.

The loaded attribute structure also contains the index to the original boundary in field `IxBon`. If one desires to observe the geometry of description, then the plotting routine `PlotRsgSpcBon.m` (in directory `/AdminMb/DescExtr/Plot`) provides a template to do that. To observe this, one needs also to save the boundary space `BSP`.

### Arc Attributes `ReadArcAtt.m`

The geometric parameters are loaded to variable (`.Geo`) as matrix `[nArc nAtt]`. 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 also organized into several groups 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, loaded as matrix `.STR`, is a coarse description using a 8-bin histogram of the straighter orientations, which then is analyzed for various axial alignments. The second group, loaded as matrix `.SFI`, expresses the same but more refined by using a 12-bin histogram. We retrieve them as substructures named `Scors` and `Sfine`, respectively.
The data structure `Scors` has the following fields, see Table 4.2.

Table 4.2: 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 4.3).

Table 4.3: 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. Figure H.11 demonstrates how easily one can obtain useful cues about the scene layout by observing the biases.

**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:

```
[HST Nbin] = LoadDescHist([fipsOut '.hst']);
```

where substructures HST.UNF, HST.BIV and HST.SPA 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 a data structure

```
[SLC HedSlc] = LoadDescSalc( fipsSalc );
```

that in turn contains four data structures, Txa, Shp, Ens and Dsc, that contain the following information:

- Txa: texture description based on statistics of the texture maps as explained in Section 2.4 and 4.2.3. The description is 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 the global map statistics in substructure `Txa.Gst`, the texturegrams in substructure `Txa.Grm`, the blob information in substructure `Txa.Blb`. In addition, there exists a substructure `Spt` holding dense point candidates. And another one for the degree of colorfulness in substructure `Bnt`. 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.

**-** `Grm` The grid map appears under substructure `Grid`; the vertical and horizontal band histograms under `Bvrt` and `Bhor`. They can be plotted as shown in `exsbTxtrMaps.m`.

**-** `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` (see also Fig. H.8).

**-** `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.

**-** `Bnt`, `mxBnt` contains a selection of points that represent clusters of high color richness, `bnt` standing for *bunt* meaning colorful. The value `mxBnt` represents the total degree of colorfulness for the entire texture in the image (not for the regions).

#### 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 13.2).

**- `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 `ISP`. 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 of the intensity distribution in image space `ISP`.

### 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( [ fips 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. The are written in red below.

- 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 (with mhst), this typically has little effect since the histograms are created with the full set of descriptors (RRE for contours, full for curve partitions). For manipulation with vectors however (mvec), some of these parameters can make a huge difference. In particular for identification in place recognition, one can observe substantial variations.

The parameter name for the long option is given in parentheses, if it 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 ISP. 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 Form

These are parameters determining entry conditions (Section 2.3.1):

`Rsg.minNpx` (`--rsgMinPix`) [absolute]: minimum number of boundary pixels for the form 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 form 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 form description. A value equal zero means all boundaries will be described as form 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 form description. To turn off any form description, one can set the value to 255.

`Rsg.bordTouch`: number of border touches permitted for form description. Default equals four, meaning all shapes are described. A value equal zero excludes any boundaries touching a border. Values 1 to 4 permit the corresponding number of border touches.

### 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 number of boundary pixels entering the 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 will not subceed 12 pixels, as that is the minimum number of pixels required for the partitioning process. Thus setting `Cvp.minPix` to zero means we take all boundaries, and those shorter than 12 pixels will be only described by the form descriptor (if `Rsg.maxNpx` is set to a value smaller 12).

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 ∈ [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 gerüst 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 square 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.

`--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.

**Parameter Intervals** The program `dscx` 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.

## 4.4 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 8.1.

With executable 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 size ntDsc x nAtt: total number of descriptors times number of attributes. 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 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 ISP, where the descriptor instance originates from. The indices are written with zero-indexing. The file extension is called lev, the complete extensions 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, psv and psh, for the vertical and horizontal position, respectively. The chromatic attributes are labeled red, grn and blu.

#### Bin-Vectors

Bin-vectors are generated with executable dbn2vmx. Its handling is analogous to executable d2vmx. It takes a dsb file as input

```
> dbn2vmx Desc/img1.dsb Vect/img1
```

which produces matrix files with extension vbn[Dsc], ie. img1.vbnCnt, img1.vbnRsg. An example is given with script exsbDbn2vmx.m.

The corresponding labels are in script o_AttsLabBin.m; they are slightly different from the ones of o_AttsLabels.m because not all attribute bins are written.

The program does not write any level arrays, since the outputted matrices are of the same size as the ones generated with d2vmx. For the position values, one would retrieve them from the (continuous) vector files vec[Dsc] (img1.vecCnt,...).

### 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`.

**Bin-Vectors** For the discrete space, there does not exist an executable yet. The example script `plcCollVecBin.m` gives a blueprint to develop one.

# 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 types of 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. The description files are required to have the same number of levels for the descriptor space DSP, 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 executable `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 for a `dsc` file, as well as a text file that contains the file paths for a list of `dsc` 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
```

which must contain the string `Mes` to be recognized as measurement file, either as directory or as file stem.

By default, the program matches all descriptor types for the entire descriptor space. 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 approximate (coarse) tuning. The second list of options, under 'Individual', allows to adjust parameters of individual descriptor types for accurate (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 measure]: sets matching tolerance to a fixed value, arcoss all levels of `DSP` (and across all descriptor types). This is for the similarity measure 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 measure. Fixed value for all levels of `DSP`. Default: around 0.025, but varies for type.

`{Dsc}.tolPos`: tolerance of position value. When the measured position difference exceeds the tolerance, 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 columm-wise corresponding to contour, form, 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 the place recognition demo, see plcMotEgo.m, upcoming in Section 12.6.

The vectors from the different descriptor types were concatenated in the following order: contours, form, 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 rre 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 space 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 | registers: 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 executable 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 executable is given with script `plcMtcHstKol.m` as part of the place recognition demo.

# Chapter 7

# Matching Texture [/MtchTxt]

Program binary `mtxt1` matches the texture information that was saved in the saliency file (`slc`, Section 4.2.4). We recall that we have several texture biases and for each one we have a grid map, two histograms, and the bounding boxes of the blobs' connected components (Section 2.4.3). These descriptions are matched individually and the dissimilarity values are outputted separately. The program takes two saliency files as input:

```
> mtxt1 Desc/frm1.slc  Desc/frm2.slc  Mes/Txt
```

This will write the dissimilarity measurements to three files with the following values:

- `TxtGrid.txt` 7 values corresponding to the first 7 entries of `o_TxtrLabels`: `num`, `blk`, `nil`, `vrt`, ...
- `TxtBand.txt` 7 values (as above).
- `TxtBlob.txt` 9 values, for all labels in `o_TxtrLabels`.

The distance values can be combined as desired. Since the numerosity and blankness bias values (`num` and `blk`) contain higher values than the remaining biases, a weighted sum performs generally better, see the organizational routine `o_WgtTxtrMtch` for an example. Script `exsbMtcTxt.m` gives an example for three frames.

A program binary that matches one-versus-many does not exist yet. For the moment, this has to be programmed explicitly, see `f_MtxtL.m` for inspiration. The demo script `plcMtcTxt.m` uses that script for the frames of the Place Recognition demo.

**Correspondence for Blobs** Executable `mtxt1` also returns the nearest-neighbor (NN) information for the individual blobs' bounding boxes. This is written to four separate files into directory `/Mes`:

- `Txt1NNMsRow.txt` contains the NN measurements for the first image versus the second. A value equal 99 signifies a no-match situation, a bounding box that has no spatial correspondence in the other image.
- `Txt1NNIxRow.txt` contains the NN indices for the blobs of the the second image. The index is irrelevant when the corresponding measurement value in file `Txt1NNMsRow.txt` has a value equal 99 (no match).

- Txt1NNMsCol.txt is the inverse to Txt1NNMsRow.txt: contains the NN measurements for the second image versus the first.

- Txt1NNIxCol.txt contains the NN indices for the first image.

Script exsbMtcTxt.m plots the correspondence between the first and the second frame.

**Performance** The matching is useful for generating candidates. It is roughly equal in performance with mhst for place recognition and has lower complexity. It is included as a preselection strategy in the cascade identifier CASCIDF.m (Section 12.2).

# Chapter 8

# 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 9), which carries out 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 descriptor space. 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 8.1 and 8.2, respectively.

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

Executable 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 stem, 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 executable 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.

## 8.2 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 9

# 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 11. This process achieves a finer segmentation than that was obtained with dscx. Program shpx then describes the shape as explained in Section 2.3.2, namely by descriptors rsg, arc and str, whose attribute values will be more precise 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 12.5). In the following we firstly explain the use of the shape extraction program (Section 9.1), then the one for patch extraction (Section 9.2). In practice, we deploy the latter program first, but it is here mentioned second, as it is rather a utility program.

## 9.1 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 additional, 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.

### 9.1.1 Parameters and Options

Setting parameters and options occurs 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).

### 9.1.2 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 11).

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.

## 9.2 Program Patch Extraction [ptchxL]

The executable `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 executable `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, and a 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 10

# Shape Matching [/ShpMtch]

The shape matching program mshp correlates shape descriptions as generated with program shpx, see also Section 2.3.2 again. For a pair of shapes, three groups of measures are calculated.

1) descriptor distances: one distance for the arc descriptor, one for the straighter descriptor and one for form rsg. The arc distance is calculated from the partition list aArc, the straighter distance from aStr. The form distance is calculated only between two vectors.

   The metric includes the angle and position attribute by default, with reference to itself (the pole of the partitions), 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. Since the spectra do not include any information on angles or position, this distance is therefore independent of those two attributes.

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.

## 10.1 Program Use and Output [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.

**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 form 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 11

# 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.

## 11.1 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.cvps` | 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 11.2. In the following it is explained what options are available.

### 11.1.1 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.

## 11.2 Output

The `.cvps` 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 shown with the example script.

# Chapter 12

# 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 12.1) and explain how to setup a cascade identifier (Section 12.2). Then we move toward the use of focus selection (Section 12.3): 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 12.4). We take that a step further and also extract and match shapes (Section 12.5) 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 12.6).

**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.

## 12.1 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'.

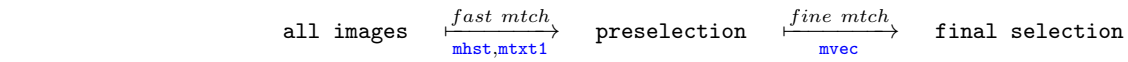
Script plcMtcHstKol performs histogram and kolumn matching, whereby here the first frame is compared to itself and the remaining four, hence the labels '0-0' to '0-6'.

**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.

## 12.2 Cascade Identification (Whole Image)

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




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 using any one of the statistical representation formats (Section 3.5), or a combination of them. The terms *fine matching* or *detailed matching* are used when we deploy vector matching, see also terminology in Appendix D.

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. In the following we elaborate on some of the routine's specifics.

The routine CASCIDF.m contains a switch for fast matching choosing between preselection with histograms, texturegrams or kolumns. Histograms and texturegrams perform better in place identification, but a more systematic evaluation might favor also kolumns; ensembles could be tested as well. After fast matching was 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 chose 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** The suggested cascade identifier can be regarded as an example. There are many variations one can test, for example running and combining the distances of all fast matching options. We tried so with the option `ens` for ensemble. Another variation would be to use a three-stage cascade, in which we start with texturegram matching to preselect candidates, followed by histogram matching to subselect candidates for detailed matching.

## 12.3 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 executable `mvec1`. The other occurs by matching an entire list, executable `mvecL`.

**Matching 1-on-1** Script `plcMtcZon1o1` matches the zones using executable `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.

## 12.4 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 executable `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 measure, one for the similarity measure, 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 13.5. But for the development of an active vision system that explores an environment, this is an optimal starting point.

## 12.5 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 executable `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 executable `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.

## 12.6 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 (`rre`) 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.

## 12.7 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 subselect by matching the bounding boxes, of either shapes or contour blobs, or both, indicated here as `mtch bboxes`.

`CASCIDF` → `mtch bboxes` → `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 immediately improve the identification of scene sets that contain similar structure but different textures, a combination with the methodology of Local Features is suggestive, discussed later in Section 13.5.1.

# Chapter 13

# 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 13.1), active vision (Section 13.2), object recognition (Section 13.3) and some other tasks (Section 13.4). Finally, we discuss possibilites of a fusion with other methodologies (Section 13.5).

## 13.1 Recognition for Navigation

For an efficient navigation through an environment, it is beneficial to detect vertical and converging structures. In our description, 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 space, which can be loaded from the `rre` 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 form 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 form `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 or regions, available in files `bspx` and `ruv` (Appendix C).

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 `rre` 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.

## 13.2 Active Vision

We have already introduced two possibilities for selecting a different focus, namely with programs `focsel` and `shpx`, the former rather for any type of structure, the latter for shape outlines in particular. These are also called *covert attentional shifts*. As part of those one can also regard the spatial histogramming of the attributes, be it for the image histogram or for texture, parameters `HistSpaDim` and `Txt.winSiz` (`--txws`), respectively. They cannot be calculated separately yet: for the moment one had to rerun `dscx`.

In the following we rather discuss *overt attentional shifts* that involve a change of camera direction, also called *saccade*. We firstly specify the challenge.

**Views in Visual Space** 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 humans interact with our surround, we place our 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. Hence, 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 process we continuously try to categorize and identify the streamed input.

To efficiently mimick this browsing process, we essentially deploy the cascade recognition process. Whereas a cascaded recognition process acts on one view, we here deal with several views in rapid succession. It is impractical to carry out a complete cascade process to each view. Rather we remain mostly in the early stages of the cascade and complete it only occasionally. The entire process is rather fluid in its execution and its exact pipeline also depends on the specific task. But for simplicity we distinguish between the *orienting* and the *confirmation* phase, Sections 13.2.1 and 13.2.2, respectively. Eventually, we discuss the process of acquiring description and representation of a novel environment (Section 13.2.3).

### 13.2.1 Orienting

The phase of orienting can include a number of processes. For action planning we may be interested in estimating the angles of certain features, in which case we observe specific attribute values in the description or proposal file, `dsc` and `qdsc`, respectively. This may not require any particular type of view recognition and much of perception-for-action can be considered orienting.

For search of an object or specific scene view, we may require also zooming and, depending on the task, also a discrimination between presence and absence of texture. That is, it is best to start with a triage that observes the texture statistics in particular. If a scene is scarce of structure, then we need to fish for locations of potential interest. Then we proceed to operate with the statistical representation formats (texturegrams, Histogram-of-Attributes, etc., Section 3.5).

#### Triage

The triage conducts a separation between texture and lack of texture. When no texture is present, then that may correspond to a 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:

```
[SLC HedSlc] = LoadDescSalc( fipaSalc );
```

To now discriminate between presence and abscense of texture, we deploy data structures `SLC.Txa` and `SLC.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.

Based on this information we may decide to adjust the zoom. If the view displays texture only, we may adjust the window size for texture formation, parameter `Txt.winSiz` (`--txws`), and recompute descriptor extraction. It would make sense to do this only for contours, but that flexibility will be provided in the future.

For a scene full of structure, the next step would be rather preselection. For a scene scarce of structure, we try to locate any structure, or structure relevant to the goal.

#### Fishing

Examples of scenes that are scarce of structure are: an unfurnished room with only a plug (power outlet) present; a landscape scene with hardly any objects, ie. the horizon at sea, etc. In such scenes, the presence of any blob or shape is of potential interest and we then scour the saliency or proposal file for candidates. Example scripts `exsbProposals.m`, `exsbPlotSalc.m` and `exsbSmlObjDet.m` demonstrate how to access that information, the latter upcoming below.

#### Preselection

The triage 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 the numerosity value `nums` is high (as retrieved in code above); 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. This would already be a step toward confirmation.

For the purpose of locking in to a certain scene view, we would exploit the statistical representation formats, specifically we apply executables `mtxt1`, `mhst` and `mkol` for generating candidates. We keep browsing until the scene candidates appear reasonable.

Preselection could also be carried out by matching vectors of the coarser levels of the descriptor space. This will be offered in the future.

### 13.2.2 Confirmation

After preselection has identified reasonable candidates for our desired view, we continue the process of view identification with detailed matching by using executable `mvec` (Section 12.2). Or we can apply scene classifiers on the `hst` file for basic-level categories.

As part of the confirmation we can also regard shape-wise scene matching as discussed in Sections 2.3.5 and 12.5.

### 13.2.3 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 the statistical representation formats for preselection, be it either for self-localization or view reidentification. Only for the purpose of confirmation and action, one would deploy vector matching (`mvec`) to exploit the precision of the description. The precise description of those views, that are used

rarely, would be moved into the background, perhaps even eliminated if memory is limited.

## 13.3 Object Recognition

### 13.3.1 General

General 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, in particular with the scripts for proposal and shape matching, `plcFocProp/plcMtcProp` and `plcShpExtr/plcMtcShp`, respectively. To adapt that scheme to categorization one would apply first histogram classification instead of matching. In the code section on histogram matching of proposals (`plcMtcProp`), we would then first apply executable `h2arr` to the histogram file,

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

whose output, `fpHsf1ArrTmp` + `harf`, is then fed to a trained classifier. This classification 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 latter is likely to provide better hypotheses.

### 13.3.2 Search; Small Object Recognition

Trying to find distant, barely visible objects, can be done by analysing ridge and river contours. An example is given with Fig. H.12, that shows how red spots can be easily detected by analysing the chromatic values of ridge contours. For example we load the full RRE space with `LoadCntRRE.m` and access the RGB values of the first level of space `RRE.ARDG`. If the orientation of the object is known, then we can exploit the calculated orientation angle of the contours to subselect the pool of candidates.

When the target object appears larger, it is more practical to search for clusters of ridge, river and edge contours, which is carried out with the texture analysis (Section 2.4), see also Fig. H.13.

The results of the texture analysis are loaded with the saliency file (Section 4.2.4). The data structure in `SLC.Txa.Blb`, 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 appears 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 form 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 descriptor output, see Section 13.5.

## 13.4 Other

### 13.4.1 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 11) 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 RRE space (if it 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 form 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.

### 13.4.2 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 script exsbRegBon.m.

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.

## 13.5 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.

### 13.5.1 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`).

### 13.5.2 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 Executables and Demos

## A.1 Executables

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`.
- `mtxt1`: matches texture information of a pair of images, specifically the bounding boxes of the various texture biases in the saliency file (`slc`).
- `shpx`: carries out a refined segmentation for a target region using a color cue, and then describes the obtained region silhouette using the form 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 executable 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.
- dbn2vmx: generates the vector matrices for the bin-vectors of the discrete space. Analogous to program d2vmx.
- 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.

- /Demos: contains various scripts demonstrating in particular parameter changes (script name contains prefix exsb).
- /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 an executable called sgrRGB demonstrating the color-segregation process used for shape extraction (Chapter 11). 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.
- /AdminMb/DescExtr/Examples: contains various scripts demonstrating how to plot descriptors and load feature files. These were the early example scripts, whose content is mostly covered in other scripts.

# Appendix B

# Image Filtering

Some options for simple image filtering are provided. This filtering is carried out only for the original image resolution, before the image space `ISP` is generated; it is not applied on higher levels of the image space.

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

| | |
|---|---|
| `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. Image filtering has little effect when using image histograms for fast matching. It is rather for matching vectors where it can make some difference.

# Appendix C

# Feature Files

These files contain the segment pixels of the feature spaces (Section C.1), as well as more information about the boundaries (Section C.2).

## C.1 Segment Pixels

### C.1.1 Contours Universe (.cuvKpt)

Of the contour universe, only the keypoints of the segments are written to file. The file is generated if long option `--saveCuvKpt` is set. It is a binary file with extension `cuvKpt`. The example script `exsbCntUnvKpt.m` demonstrates how to load the points.

The keypoints of a segment are its two endpoints as well as its midpoint, the pixel that lies halfway on the curve (not the geometric average between its endpoints). The keypoints are written per level, per contour type and per point type.

The first value holds the number of levels. Then each level of the space 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]
...
```

The segment coordinates are absolute values corresponding to the map size of the

image space. Thus, for a pyramid space, they need to be upsampled to match to the original image resolution if they are used as object/part proposals.

### C.1.2 Region Universe (.ruv)

The region pixels are saved to file if long option `--saveRuv` is set, in which case they are saved to a file with extension `ruv`. The regions can be loaded by routine `LoadRegUnv.m`, an example is shown in script `exsbRegBon.m`.

The region pixels are specified as linear map index, in variable `IxLin`, see reading routine `ReadRegPixBlok`. They were written as a single array. Their starting indices are in variable `Anf` (Anfang=beginning). To address the pixels in the map, it requires the map dimensions, which are given with variable `SzM`.

### C.1.3 Boundary Space (.bspx)

The boundary pixels are saved to file if long option `--saveBsp` is set, which then generates a file with extension `bspx`. It can be loaded with routine `LoadBonPixSpc.m` for a space of boundaries, or with `LoadBonPix.m` with the boundaries from RGB segregation.

Routine `ReadBonPixSegw.m` reads one level and returns the segments and their corresponding region index in struct `Org`. Variable `Org.Dth` holds the depth index, variable `Org.CC` the region index. With those two indices we can then retrieve the boundary's region pixels. Script `exsbShape` gives an example.

## C.2 More Boundary Information

These files contain in particular bounding boxes in various formats. Some of them were originally developed to make a comparison with object proposal studys.

### C.2.1 Bounding Boxes (.bonBboxRaw)

The bounding boxes for regions can be saved by setting the (long option) flag `--saveBonBboxRaw`. They are written in text format to a file with extension `bonBboxRaw`. The bounding boxes are in original map resolution. The information in this file is minimal.

The first two integer values of the file hold the number of levels `nLev` and segmentation depth `depth`. 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.2.2.

### C.2.2 Aspects (.bonBbox, .bonAsp)

If the flag --saveBonMore is set, then more boundary information is saved to two separate files:

- bonBbox: bounding box information, similar to the bonBboxRaw file as introduced above, but in different format. It contains the bounding boxes upscaled to original image resolution - if the architecture is a pyramid.
- bonAsp: holds more boundary aspects.

The two scripts exsbPlotBon.m and exsbPlotBonPix.m demonstrate how to load those data files.

The bonBbox file contains the following:

- 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 bonAsp file 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.2.1 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.

# Appendix D

# Terminology, Definitions

To clarify the usage of some of the terms we provide here a summary. For the different type of scales or axes, see Concrete/Abstract, Fine/Coarse and Local/Global.

**Aspect** Expresses general description, from which often more specific description is derived, then called attribute. The term is used in particular with the parameters of bounding boxes, connected component, etc.

**Attention, Attentional Shift** Refined analysis at a specific location. If that occurs with camera direction fixed, then that is called *covert* attentional shift, ie. as carried out by programs `focsel` and `shpx`. If that occurs with a saccade, then that is called an *ouvert* attentional shift. Program `shpx` can also be deployed in combination with a saccade, in which case its use becomes an ouvert shift.

**Attribute** The parameter name for a specific aspect of a descriptor or description, deployed in recognition (categorization, classification). Its abbreviation appears in color `olive green`.

**Attribute Space** The multi-dimensional space spanned by the attributes of one descriptor type, usually for one level of the image space.

**Bias** Is similar to the term aspect: it expresses general description that is not specific enough to be considered an attribute; nevertheless, its abbrevation appears in color `olive green` like an attribute. The term is used for texture description in particular.

**Cascade Recognition** Is a multi-stage process that gradually approaches the recognition goal by generating an increasingly reduced set of candidates that are then used for detailed matching. We use the term *fast matching* to refer to the early stages, and the terms *fine matching* or *detailed matching* to refer to the later stages. The word *fine* is used for reason of textural compactness; it does *not* refer to the fine/coarse scale of the image space.

While the Viola/Jones face-detection algorithm is weeding through a huge set of features, in this context the cascade is rather weeding through the multi-dimensional description space.

**Concrete/Abstract Spectrum** Expresses the degree of abstraction of a structure or category obtained from a parametric description. The concrete degree is suitable for identification, the abstract degree for categorization. Abstraction can occur anywhere on the fine/coarse scale. For that reason we refrain from using the terms fine or coarse for abstraction.

**Descriptor** Expresses a structure by a number of attributes (Section 2.3). This structure can be a single feature, a group of features or a group of descriptors. The abbreviation of a descriptor appears in color `green`.

**Descriptor Instance** The attribute values of a single descriptor (of one descriptor type).

**Descriptor Space** The list (space) of attribute spaces for one descriptor type as obtained from an image space. The list represents a fine/coarse scale, where the coarse scale can be used for generating candidates to be matched with the full space to accelerate recognition.

**Descriptor Type** A genre of descriptor or description, such as a contour, arc, straighter, form, shape, etc.

**Feature** A feature is a segment of pixels (Section 2.2), ie. a contour segment or a region segment. It can be considered local or global depending on its size. A parameterized feature is called descriptor (see definition).

**Fine/Coarse Scale** Expresses the granularity of the image (or map) resolution. Lower levels of the image space `ISP` possess finer granularity, higher levels have coarser granularity. The scale space expresses a fine-coarse scale where the resolution does not change (Section 2.1.2); for the pyramid the resolution decreases toward the top level.

**Local/Global Scale** Expresses the size of the window under investigation or the direction of processing. For example, the region-segmentation process starts with global thresholding, which then becomes increasingly local by focusing on the segmented regions, therefore called *global-to-local* to indicate the direction of evolvment. Or the selection of descriptors starts with the largest (most global) one, and then proceeds by selecting the smaller ones. Or the curve analysis occurs with a range of window sizes, that corresponds to a local/global scale. This local/global analysis can occur anywhere on the fine/coarse scale.

**Metric** We use the term loosely (in comparison to mathematics), for reason of textual compactness. It stands for proximity measure, more specifically for any similarity or dissimilarity measure, or their measurement value. A metric consists of one or more rules and parameters.

**Saccade** Is a change in camera direction for the purpose of acquiring more visual input. Also known as *ouvert* attentional shift, see term attention.

# Appendix E

# Color Code (of Text)

The text colors denote the following information:

`Executables` `dscx`, `mvec1`, etc.

`File Extensions` `dsc`, `hst`, `slc`, etc.

`Measurement File` file holding measurement values, ie. dissimilarity values of a matching process

`Description` such as descriptors and texture description.

`Attributes, Biases` such as descriptor attributes or texture biases; or other specific variables of description.

`Parameters` individual parameter names, or a file holding parameter values.

`--Long Options` parameters specified as command arguments to an executable.

**Process** identification, categorization, classification, etc.

`Example Script` usually contains prefix `exsb`, ie. `exsbDscx`. For the place recognition demo the prefix is `plc`.

# Appendix F

# Distributions, Implementation

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

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

| | |
|---|---|
| Windows 10 | SEHBAU_win10.zip |
| Windows 11 | SEHBAU_win11.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 G). In the following we discuss some performance aspects.

## F.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.

## F.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 (`imgSpc = 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 (`imgSpc = 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.

## F.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, Python** For issues with Matlab or Python itself, we refer to the next appendix (G) on administrative code.

# Appendix G

# 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 more elaborate than that for Python and contains the most comments. The Python code contains the essential functionality, but lacks often comments, for which one has to consult the corresonding Matlab routine for the moment.

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).

- `exsb{Script}`: these scripts explain how to run an executable. 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. In Python, one needs to select the filename of the executable, ie. if it is called `python3`, then select the corresponding string in `exsbAll.py`.
- `plc{Script}`: 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`.

**Windows/Linux** I tried to account for all cases in the early, simple scripts. In later scripts, the call for executables under linux may lack the path-indicating dot-dash prefix `./` and thus produce no results; or if verification has taken place, it may appear with the error message `program did not execute somehow`.

Should there be other 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. More specific explanations for the individual languages follow now.

## G.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 `system` (or `dos`) function. There could be two reasons. Under linux it may lack the path-indicating dot-dash prefix as pointed out above already. Or it is caused by 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`.

## G.2 Python

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

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

where the first line adds the relative 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`.

We apply mostly the modern way of handling file names by using module `pathlib`, in particular in example scripts, which is perhaps too advanced already for earlier Python versions, see also paragraph 'Issues' below. But for a number of functions we still utilize the earlier way of using module `os` and string types, as we prefer concatenating long options as a single string, and not as individual path objects.

**Issues** There are a number of early obstacles that you might encounter:

- `Python not found` when running `exsbAll.py` or `plcAll.py`: occurs when your executable is called python3, in which case you have to change the to the appropriate filename in script `exsbAll.py`. Not done yet in `plcAll.py`.

- `Module numpy not found`: numpy is a basic python module. You need to install. It is required to load the description files.

- {program} did not execute properly somehow: perhaps the arguments were not properly concatenated, see also paragraph 'Version' below. In that case, you have to concatenate them the old-fashioned way, as shown in exsbDscxFull.py.

**Inconveniences** As already mentioned, 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 more specific types such as `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 argument format when one calls function `subprocess.run`.

## G.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 quasi syllables of three to four letters, aligned from abstract to more concrete.
- `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`.
- `Renn{Prog}`: runs an executable from Matlab/Python using the system call. It is a wrapper routine facilitating the use of the executable with its options. For example function script `RennDscx.m` runs program `dscx`.
- `pso_Prog`: parses the standard output obtained from an executable, e.g. script `pso_Mvec1.m` parses the output of `mvec1`.
- `exsb{Script}`: 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 `Renn{Prog}`.

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

# Appendix H

# Figures

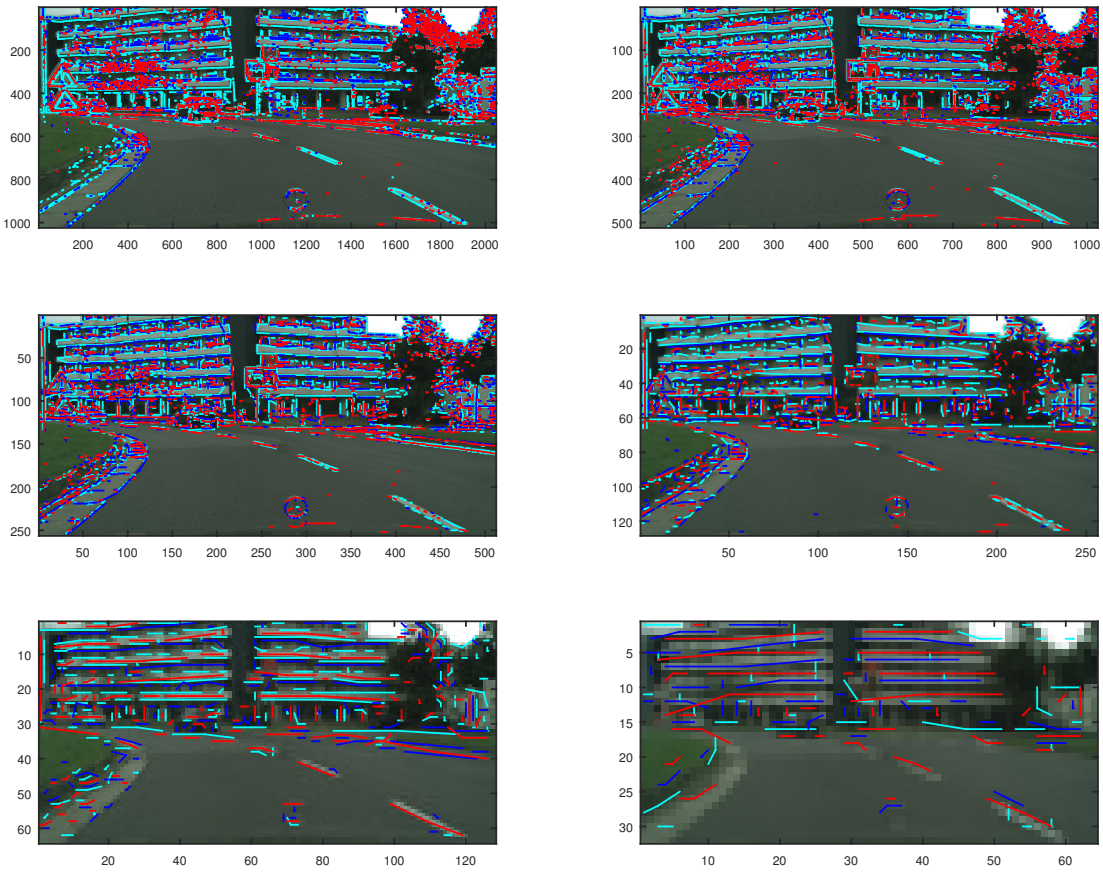



Figure H.1: Contour universe for 6 levels of a pyramid. Red=ridge, blue=river, cyan=edge. Figure generated by `exsbCntUnvKpt.m`. This does not show the actual curve pixels, but only a three-point outline, namely two straight lines that connect a segment's endpoints with its midpoint (the latter a quasi vertex).

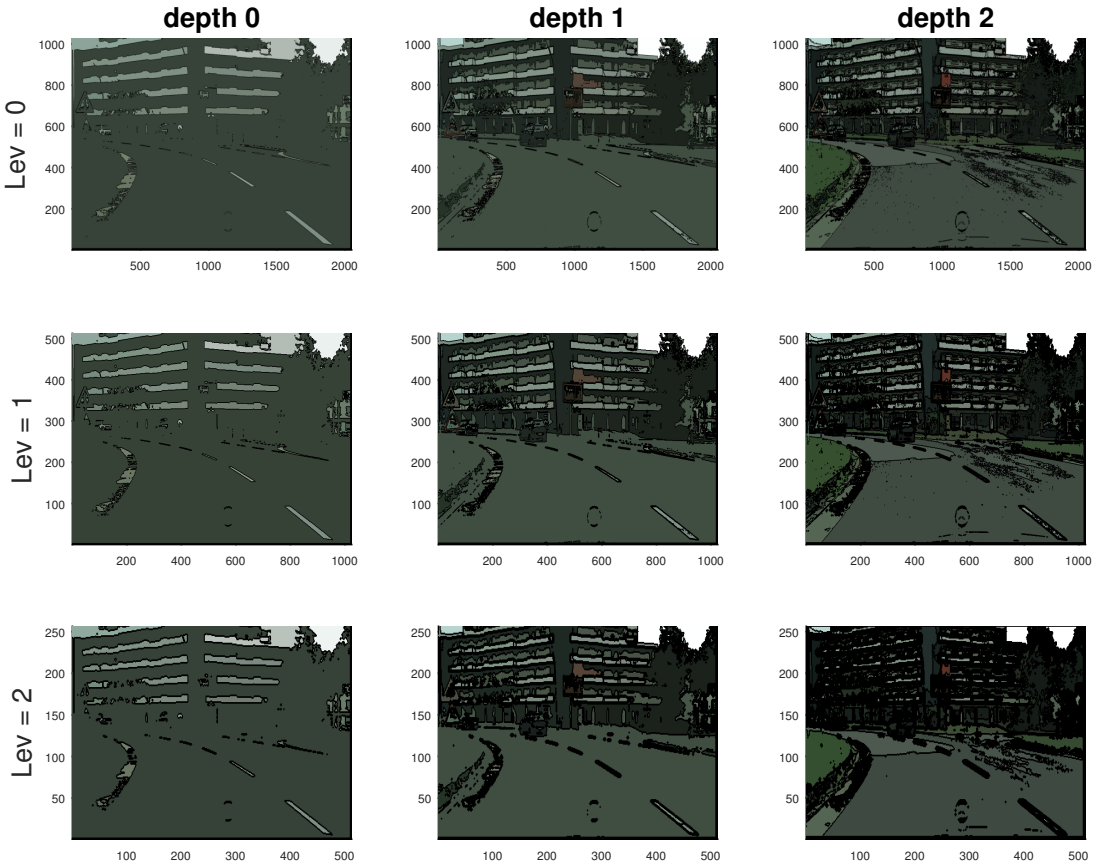



Figure H.2: Region universe as obtained with the (divisive) local-to-global segmentation procedure. Figure generated by script `exsbRegBon.m` (directory /`Demos`).
**Rows**: level of image space (pyramid in this case).
**Columns**: depth of segmentation tree.
Note, that the car is fairly well segmented into its parts at lev=3 and depth=2 (bottom, center plot). We can obtain shaper region boundaries by using executable `shpx` applied to those regions (performing thus a quasi-attentional shift).

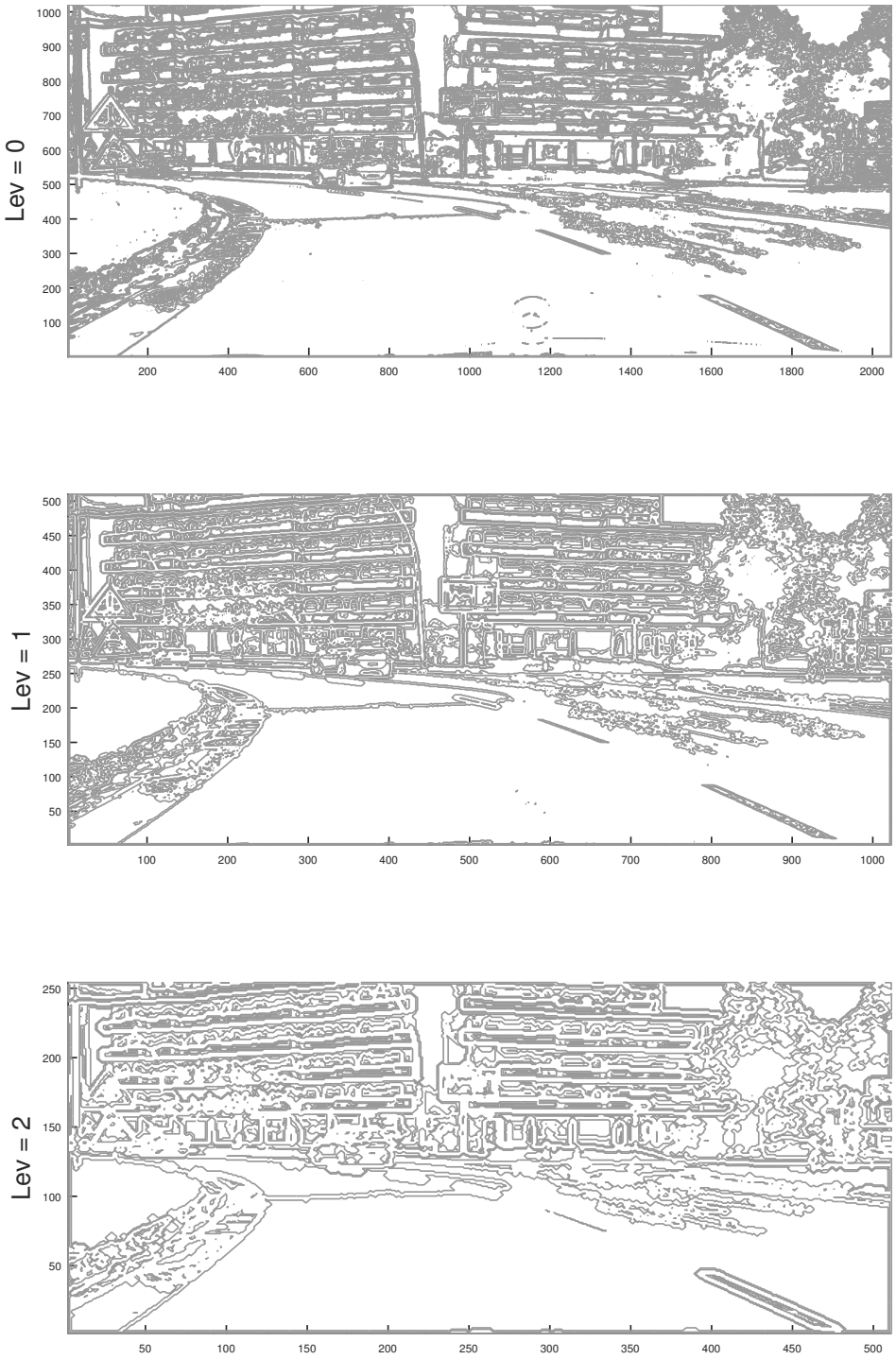



Figure H.3: Boundary space `BSP`. The boundaries were flattened per level, that is one (level) plot contains all boundaries from the different depths (per level). Figure generated by script `exsbRegBon.m`.

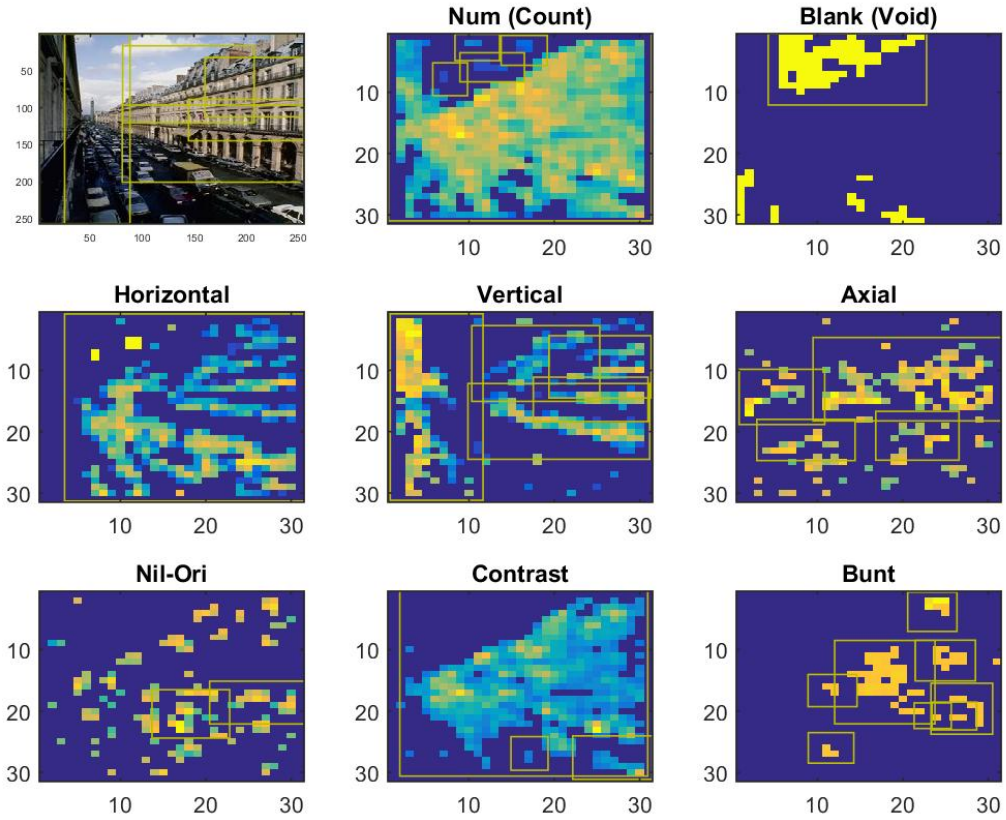



Figure H.4: Texture maps as loaded by the `txm` file and the corresponding bounding boxes, the latter loaded from the `slc` file (`exsbTxtrMaps.m`, directory `/Demos`).
**Num**: total contour count (numerosity) per window.
**Blank**: void of texture (logical map): not sufficient contour segments detected to qualify as texture. For example the contour count in the sky region did not qualify as texture (compare with map `Num`).
**Horizontal**: windows with a horizontal orientation bias of its segments.
**Vertical**: vertical orientation bias present. Note that the colonades are segmented properly.
**Axial**: both horizontal and vertical bias present.
**Nil-Ori**: no dominant orientation angle detected. In this scene, the nil-bias can be regarded as noise, but becomes relevant for scenes that contain natural elements, in particular foliage (see next figure).
**Contrast**: difference in intensity multiplied with count (Num).
**Bunt**: degree of colorfulness (relative to the entire image, specifically value `Txa.mxBnt` available in the `slc` file).

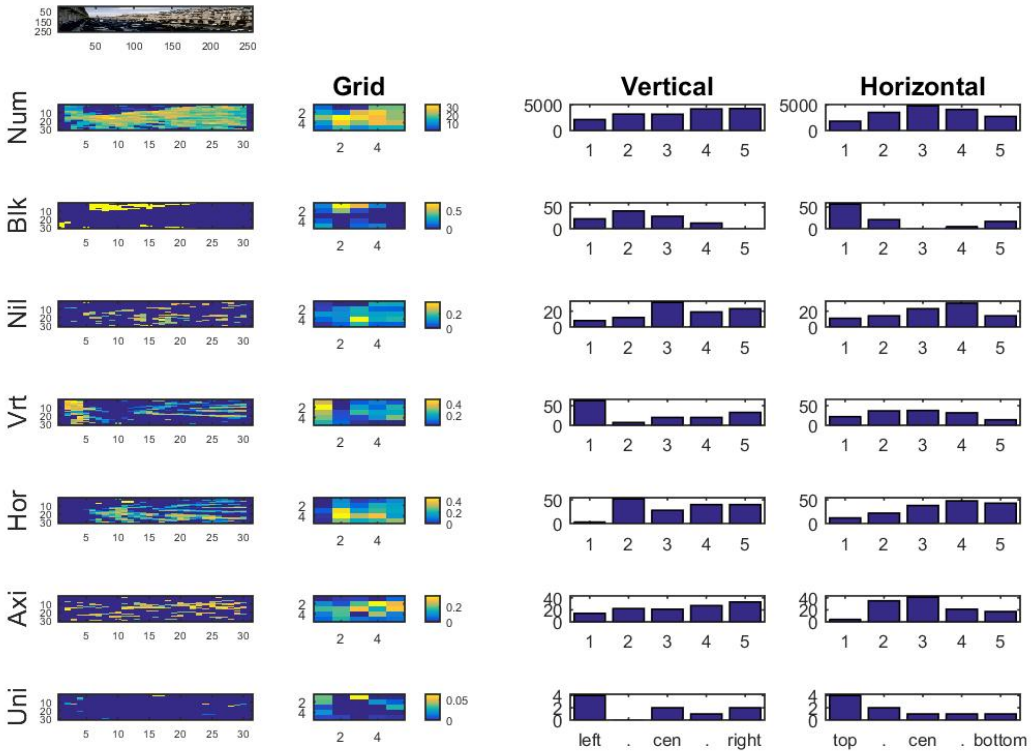



Figure H.5: Texture description by subsampling the texture map to a *grid map*, followed by histogramming the grid vertically and horizontally, called *band histograms* . Collectively call texturegrams. Same image as previous figure. Generated with `exsbTxtrMaps.m` of directory `/Demos`.
**Leftmost column:** texture maps as shown in previous figure.
**2nd column:** the subsampled grid map.
**3rd column:** vertical histograms of the grid.
**Rightmost column:** horizontal histograms.

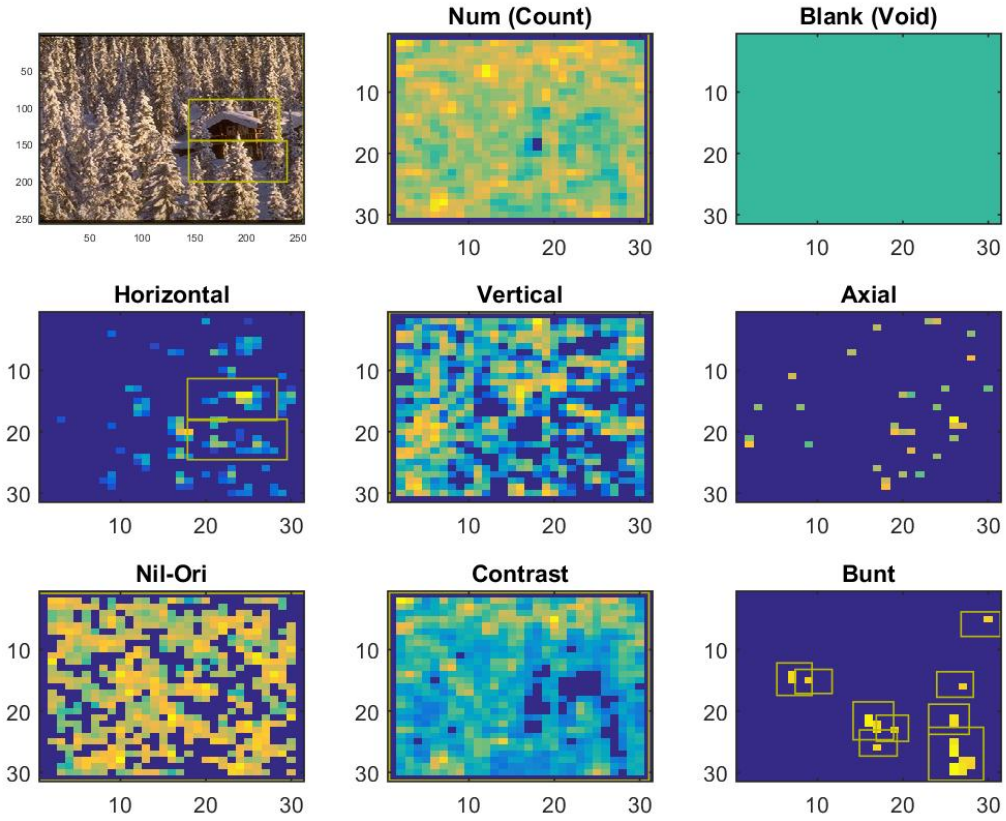



Figure H.6: Texture maps for a forest scene.
**Num**: the lowest count is around the house.
**Blank**: there are no blank windows detected: everything is considered texture.
**Horizontal**: detected around the house.
**Vertical**: much of the texture is vertically oriented.
**Axial**: only few windows show both horizontal and vertical bias.
**Nil-Ori**: largely present.
**Contrast**: low around the house.
**Bunt**: its presence may seem exagerated here, but as pointed out before, this is relative to the entire image.

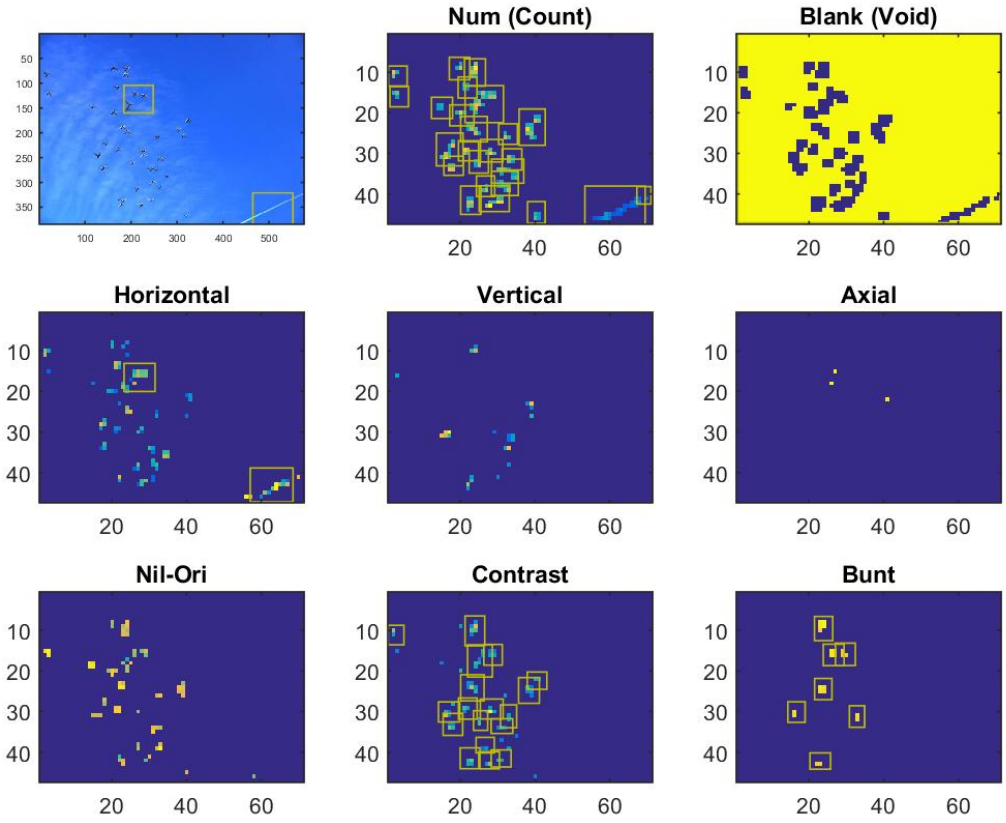



Figure H.7: Texture maps for a sky scene with birds (CamSDD collection).
**Num**: only few windows show texture.
**Blank**: most of the sky is blank.
Mainly horizontal and nil-ori texture is present. Note that some of the bounding boxes appear as clusters of three to four birds, for example the cluster just right of the image center represents three birds. In order to refine the segmentation, we rerun this image with a smaller window size (parameter `txws`), see result in next figure.

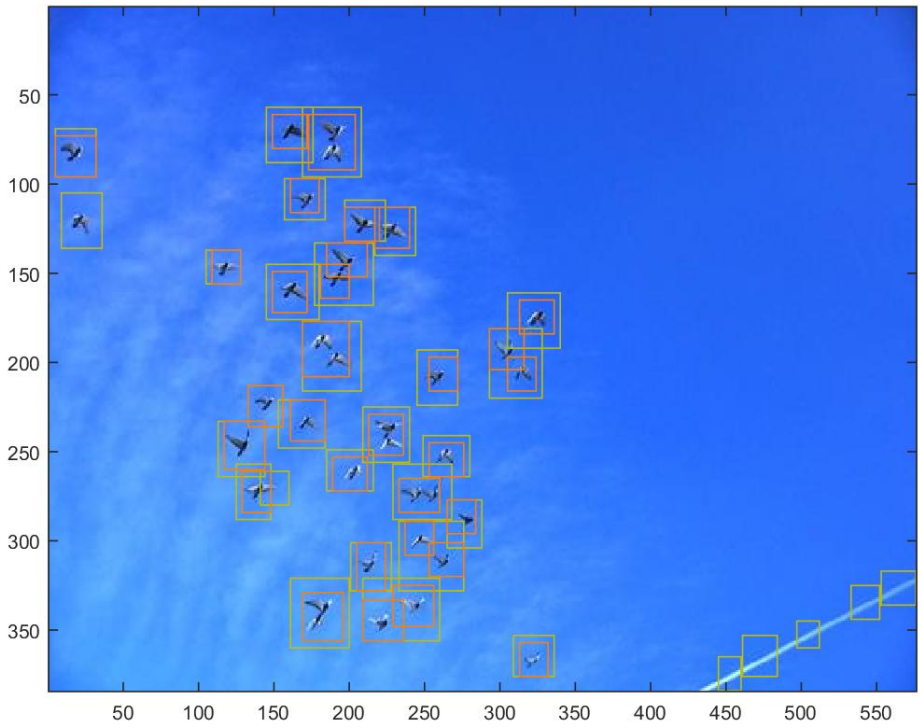



Figure H.8: Small object detection by `exsbSmlObjDet.m` (directory /`Demos`). The texture maps for this scene were shown in the previous figure, where it was run with the default size for the texture window. To obtain a finer bird detection, the image is rerun with a smaller value for parameter `txws`, resulting in the shown bounding boxes. The birds just right of the image center are now properly segregated. In total, there remain only four clusters of two birds.

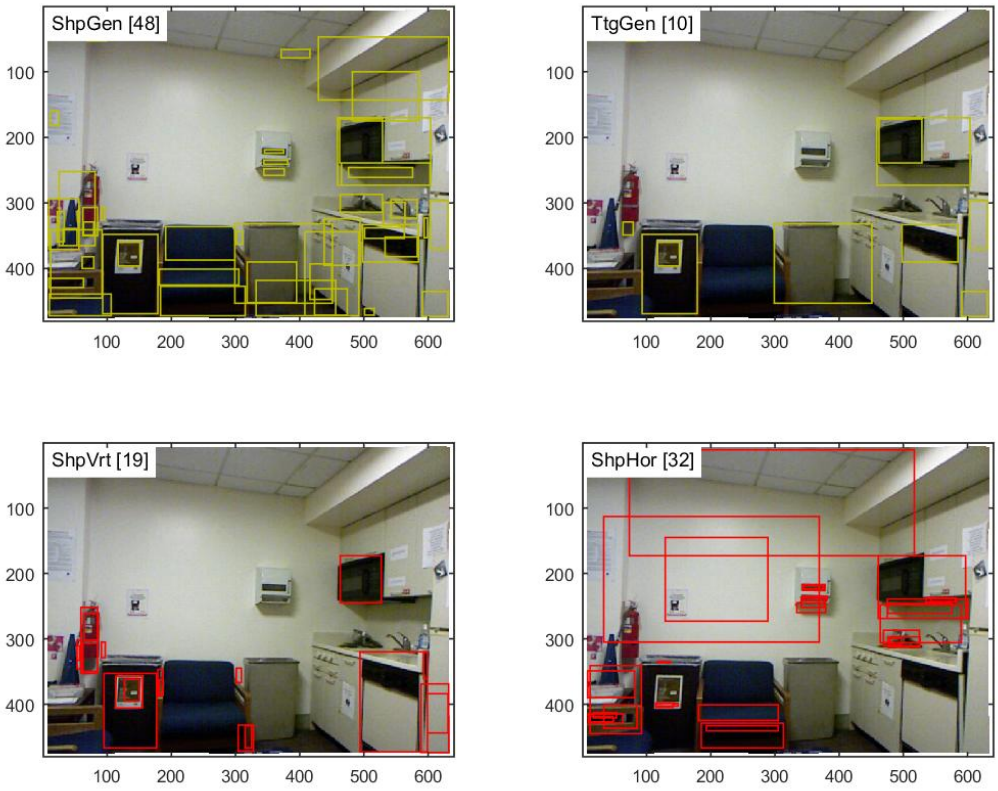



Figure H.9: Proposals as generated by script `exsbProposals.m` (directory `/Demos`).
**Upper Left**: bounding boxes for a selection of general shapes.
**Upper Right**: bounding boxes for some tetragon candidates.
**Lower Left**: shapes with vertical orientation bias (bounding boxes).
**Lower Right**: shapes with horizontal bias.

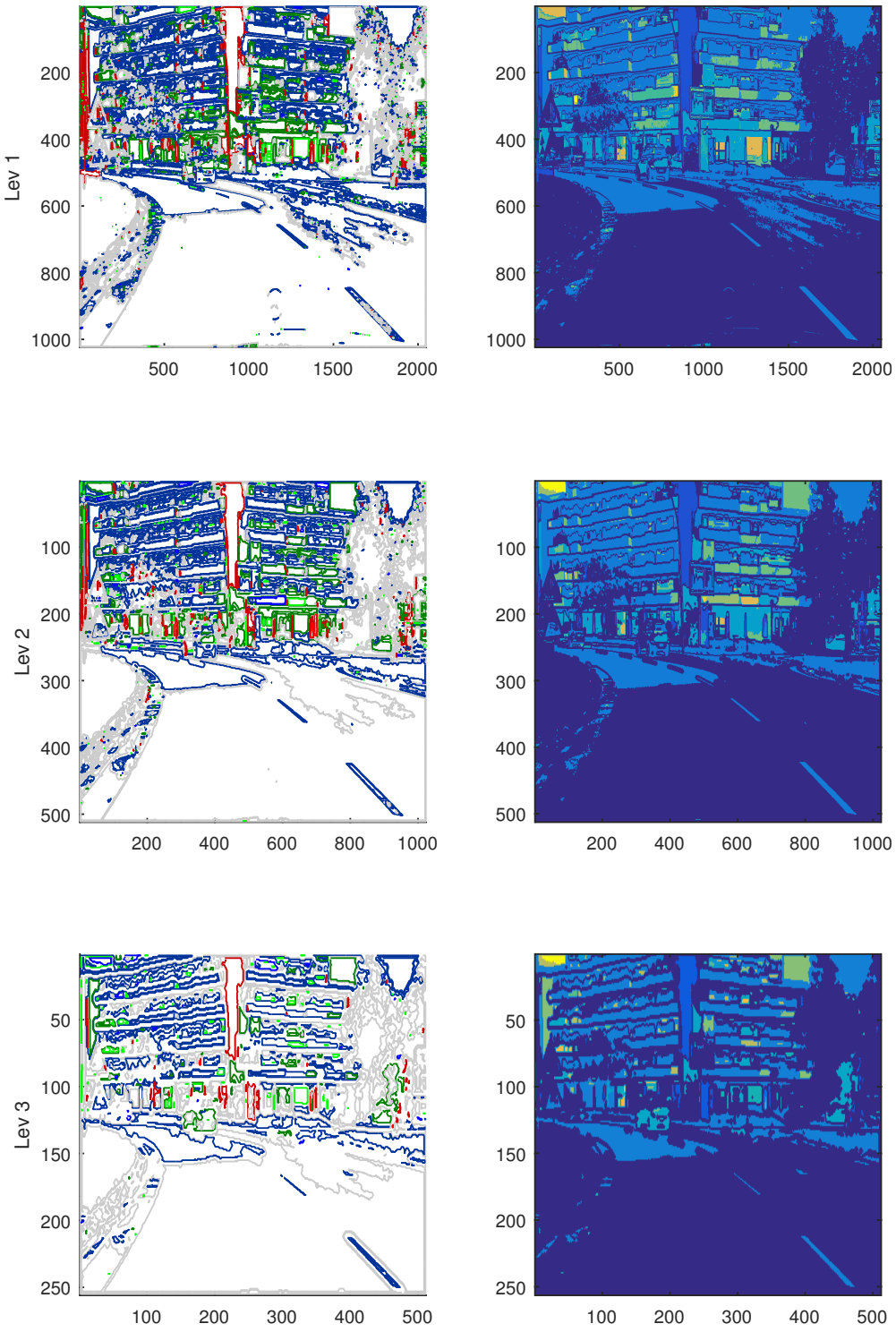



Figure H.10: Parameterization of boundaries using the simple boundary description for a pyramid of three levels (script `exsbFormAxial.m` in directory `/Demos`). In gray are shown all boundaries. In red tone are shown boundaries whose bounding boxes are high; in blue tone those that are wide; in green tone those that are axial, the latter determined with the simple hull description by the form descriptor (`rsg`).
**Left Column**: In bright red/blue/green are shown boundaries with high convexity (of corresponding geometric bias).
**Right Column**: the corresponding region interior for those boundaries (as an alternative illustration).

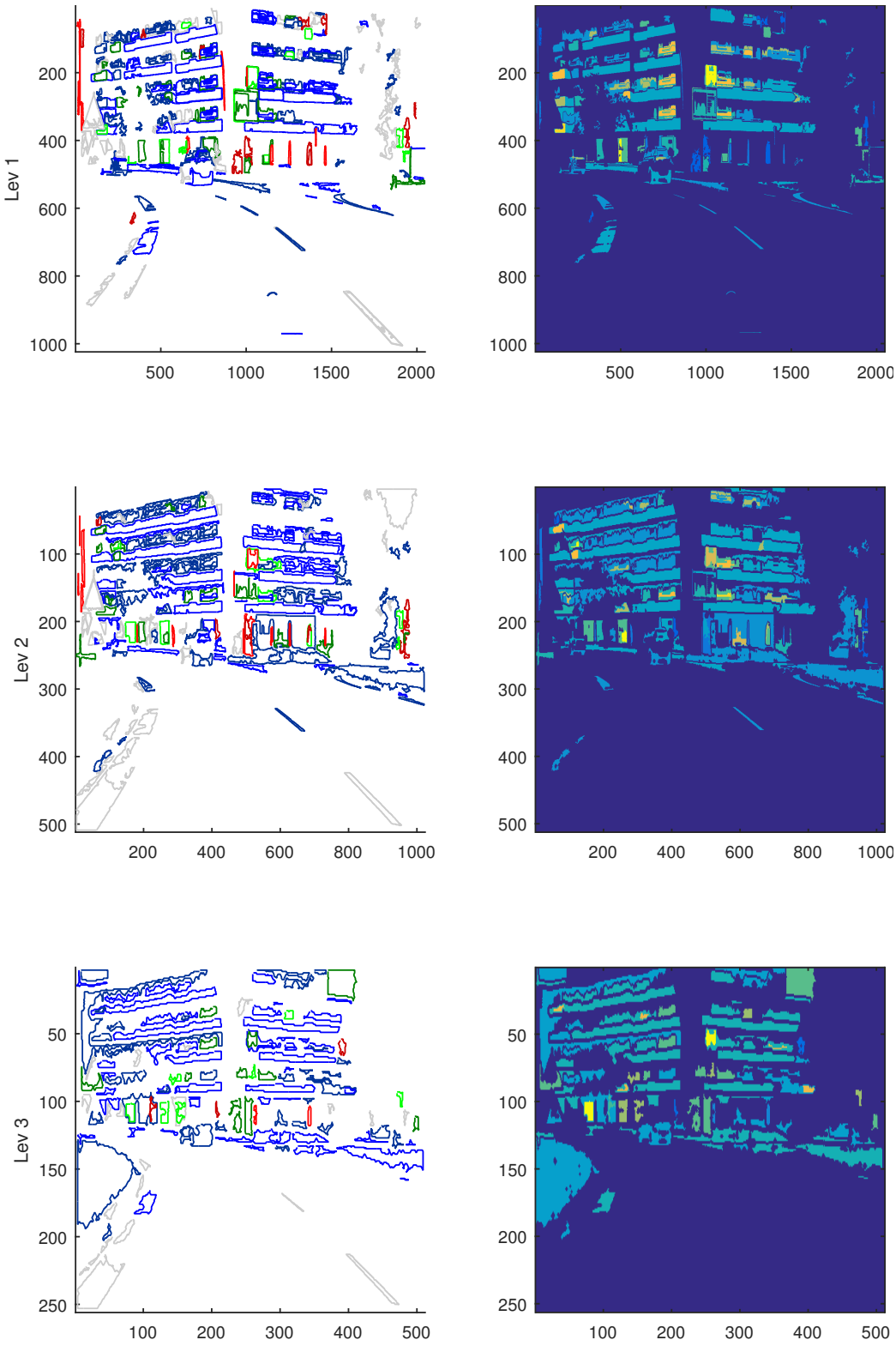



Figure H.11: Axial shapes as identified with the partitioned-shape descriptor `shp` (`exsbShape.m`).
**Left Column**: Shapes with vertical bias (red), horizontal bias (blue) and both biases, called axial (green).
**Right Column**: the corresponding region interior for those shapes.


Figure H.12: Salient red spots in a road scene obtained by analysing the chromatic values of the ridge contours, visualized as search with starting point marked by a red asterisk. The most red-salient spots are the cars' taillights, followed by the information board in red color. For this result, no image filtering was carried out, merely the structural analysis as carried out by this system. Figure generated with `exsbSalRedSpots.m`.

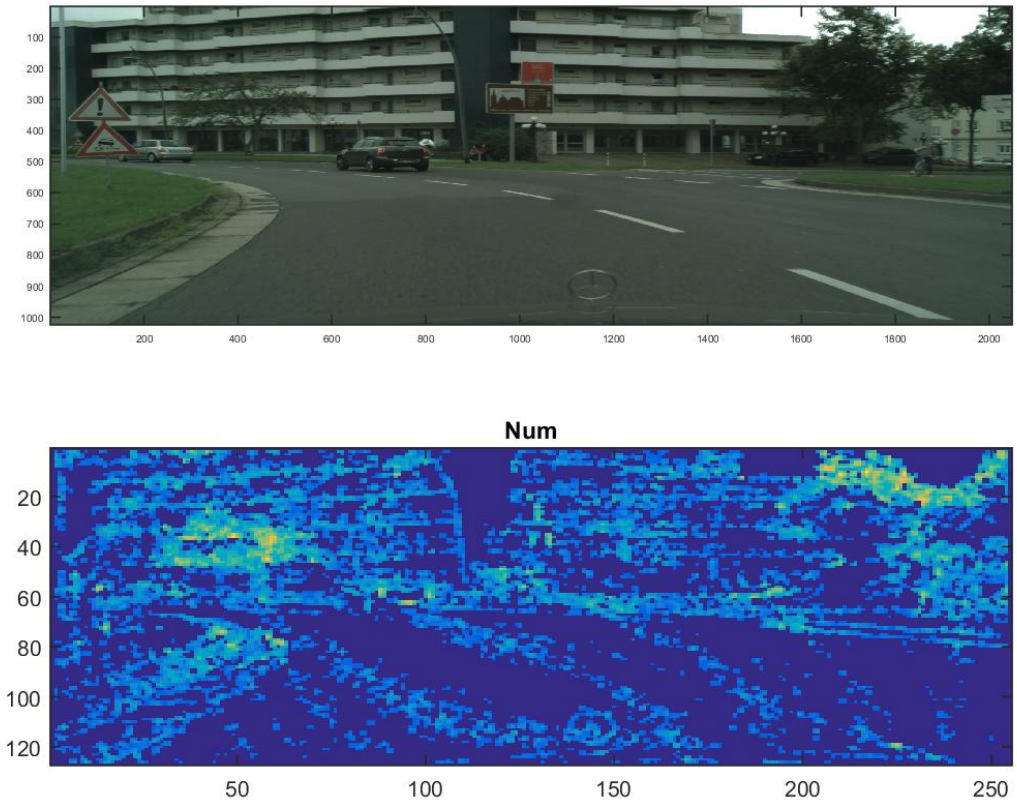



Figure H.13: Numerosity map of the road scene (from texture analysis). Since this scene has a number of low-contrast objects, we reduced the minimum contrast for contours to `0.02` (default `0.05`), to ensure we have a large pool of candidates to start with, while still ignoring much of the visual space.