Learning Occlusion Regions

by

Ahmad Humayun

Submitted to the Department of Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science

Computer Graphics, Vision and Imaging

at

University College London

September 2010

Supervisor:

Gabriel J. Brostow

Disclaimer

This report is submitted as part requirement for the MSc. Degree in Computer Graphics, Vision and Imaging at University College London. It is substantially the result of my own work except where explicitly indicated in the text.

Learning Occlusion Regions

by

Ahmad Humayun

Submitted to the Department of Computer Science on Septemeber 7^{th} 2010
in partial fulfillment of the requirements for the degree of
Master of Science
Computer Graphics, Vision and Imaging

Abstract

Pixels occluded from one frame to the next pose a significant problem for algorithms computing motion, depth, or temporal segmentation. Finding these occlusions is non-trivial especially in scenes lacking texture or in sequences undergoing large motion.

This thesis develops a supervised learning method to identify regions of occlusion in a two frame sequence. The algorithm's main contribution is a set of features that correlate with occlusion regions, and a flexible classification framework to use these features. The thesis offers a review of relevant literature, a detailed description of the algorithm, and analysis on both synthetic and natural sequences in comparison to competing algorithms.

Thesis Supervisor: Gabriel J. Brostow

Acknowledgements

I would like to acknowledge the following, without whom this thesis would have not been possible:

- To God who has been most Gracious, most Merciful. بسم الله الرحمن الرحيم
- To my parents, Asaf and Naheed Humayun for their love, patience and constant encouragement for all my pursuits.
- Gabriel Brostow, my advisor, who helped me learn more about Computer Vision and introduced me to the interesting field of Machine Learning.
- Oisin Mac Aodha, who allowed me the time and critical thoughts to learn more about my thesis. He went the extra mile to proofread my thesis and collect the datasets herein.
- Sohaib Khan, without whom I might never have endeavored in the field of Computer Vision and academic research.
- My sister and brother-in-law who accepted me whole-heartedly in their abode at London.
- All my friends at UCL who have helped me make this thesis better through their advice and discussion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Occlusion resolution has long been a subject in cognition and visual perception. Researchers have explored both the idea of completion of shapes behind occlusions [42, 30, 60] and the identification of occlusion regions [20]. In the field of Computational Vision, methods have been proposed to compute occlusions with depth maps [56, 53] and flow fields [52, 38].

One may ask why is it even important to locate these regions? Finding these regions accurately has benefits across many techniques in vision. In flow computations, algorithms which lack knowledge of occlusions perform particularly badly near such regions. In some cases they try to compress regions incorrectly to make room for pixels which were actually occluded. Hence, not finding occlusions not only hurts the performance of flow algorithms on regions of occlusion, but also on the nearby pixels. This is especially true when occlusions occur on textureless surfaces.

For computation of depth for stereo, the need for locating occlusions is apparent. Not finding them can lead to incorrect assumptions of disparities in pixels. Like flow, the performance hit on nearby pixels can occur during regularization. On the other hand, finding occlusions in a stereo framework can help derive some parts of the scene structure.

Even in motion segmentation frameworks the knowledge of occlusions can be critical. One way of computing layer ordering of pixels is to find the occluded pixels and then reason on the ordering of adjacent layers. Finding the correct relationships between these regions can help establish the ordinality of object layers (what went on top of what). A few methods, like Ogale et al. [38], solely reason on occlusions to help establish ordinality of layers.

Given that techniques for flow, segmentation, stereo, shape and depth estimation are all entangled in a "chicken-and-egg" relationship, Ogale and Aloimonos [36] argues that these early vision methods can only succeed if regions of occlusion are identified.

Having established that finding occlusion regions is important, we propose a method in this thesis for classifying pixels as occluded or not in a learning framework. The key contribution is the development of simple features to train a classifier in a *supervised* manner. Since we are finding occlusions, where temporal reasoning is critical, one of the key concerns discussed in our thesis is what to use to develop such reasoning. The natural method would be to opt for a particular optical flow algorithm and build features using it. This technique might not be ideal since flow on occlusions is an under-constrained problem, and depending on a single algorithm will make it the weakest link in our framework. To counter this, we use a set of candidate flow algorithms to build features with temporal reasoning over the scene (see Section 3.2.2).

The other key-point to discuss in a supervised learning framework is the training dataset. Although there is a paucity of occlusion training sets, we ensure that our training samples are representative of both synthetic and natural sequences (see Section 4.2).

Before we overview our algorithm (Section 1.5), we will discuss the goals of this thesis and develop an understanding of the related key concepts (flow 1.2 and occlusions 1.3). We will also discuss what is a supervised learning method in general in Section 1.4.

1.1 Goals

The goal of this research is to explore a new framework for detecting occlusion regions and how current techniques in this domain may be improved in a learning-based approach. We hypothesize that given the right set of simple features for detecting occlusions, a classifier can be trained to give results more accurate than the ones produced by any single feature in the set. We will develop features which work on image properties and on the flow of pixels - even though we use a framework which is open ended for practitioners looking to add new features. The attractiveness of learning methods lie in the ease provided in extending the input feature vector. The thesis uses a supervised learning scheme (see 1.4), one which intelligently applies the features provided as need be and ignores features that perform unfavorably.

The motivation for building this framework is to provide a tool which assists in motion segmentation, optical flow, stereo techniques. As discussed in the opening paragraphs of this chapter, occlusions play a critical role in these techniques, yet there is no single standardized technique to classify such pixels. Our thesis aims to fill this void.

Finally we would compare our results to the current state-of-the-art techniques for detecting occlusions.

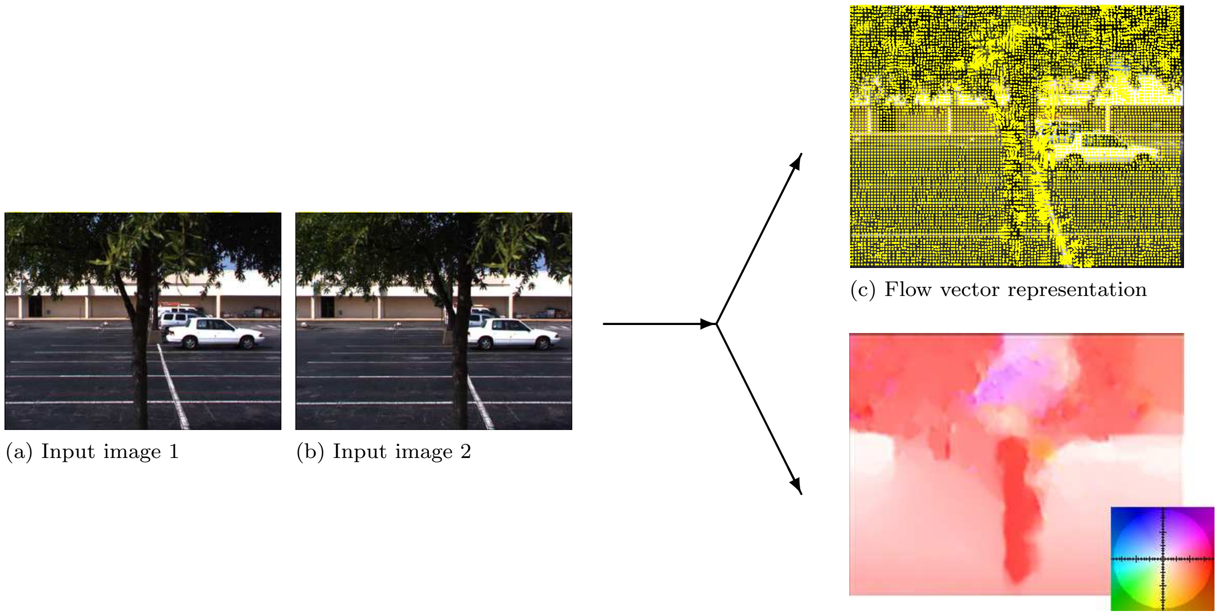
1.2 What is Optical Flow?

The problem of optical flow helps establish the direction of motion in two or more images. Most of the earlier techniques computed these “motion vectors” for each individual pixel. Algorithms proposed later exploited the constancy of motion across planar segments of images. Optical flow can be thought of a specialized segmentation problem itself (see p. 12, Ross [43]) - segmenting a set of images into regions of motion. Current advancements in optic flow help improve segmentation of motion by concentrating efforts on discriminating regions across motion boundaries.

Figure 1.2 shows two representations used to display optical flow results. Figure 1.1d shows the representation in which the different flow vectors are painted using a colour wheel.

Formally, optical flow is defined as follows: given two images I_1, I_2 and regions in them r_i^1, r_j^2, optical flow defines two things: correspondence in regions i.e. $r_x^1 \equiv r_y^2$, and the motion parameters Θ_{xy}^{12} that transform r_x^1 to r_y^2. As explained above, earlier optical flow schemes operated on individual pixels, in which case r_i^1 would refer to individual pixels in image 1, and the motion model Θ could only be translational. In frameworks which deal with regions/segments, r_i^1 would refer to a collection of contiguous pixels and the motion model could be anything from pure translational to projective.

Most segmentation and optical flow schemes are riddled with the same set of problems. The most discussed/researched problem out of them is the problem of finding accurate segmentation or optical flow for regions close to motion boundaries. Interestingly, occlusion regions tend to lie next to these motion boundaries. Most techniques, in both domains, model this problem explicitly in fear of failing on these contentious boundaries. This possibility of failure is largely due to priors in respective techniques in order to make them tractable: many optical flow techniques demand that pixels/regions don’t change



(a) Input image 1 (b) Input image 2 (c) Flow vector representation (d) Color encoded flow representation

Figure 1.1: Shows the two common optical flow representations

some image property (like brightness or texture) as they move; similarly algorithms for segmentation on monocular images suppose that object discontinuities usually coincide with colour/texture discontinuities.

Shi and Malik [46] point out that early approaches for optical flow took clues from discontinuities only in a post-processing step. This proved disadvantageous for mainly the following reasons:

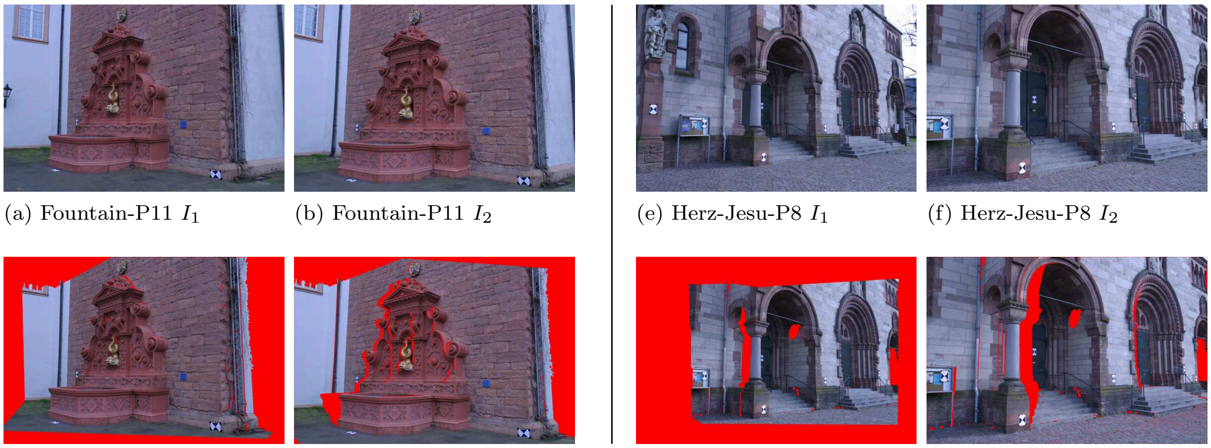
1. Large planar regions with one-dimensional or no texture are notorious for simple optical flow techniques. Discontinuities carry key to solving these regions correctly.

2. To solve (1), smoothing constraints were proposed to interpolate flow fields. But to incorporate some smoothness constraint, the image needs to be segmented before hand to avoid smoothing over discontinuities!

This provokes the thought that the optical flow and segmentation are intimate problems. Moreover segmentation without the knowledge of occlusion will fail temporally, as segment assignment over time would be inconsistent with the actual surfaces in the scene. This necessitates for a framework that computes motion to also look at temporally consistent segmentation, which in turn requires the knowledge of occlusion regions.

1.3 What is an Occlusion?

Now that we have formulated the problem of flow, we are equipped to discuss occlusion. Establishing point correspondence between a pair of images involves pairing a point in one image to a *unique* point in the second image. Due to changes in *visibility*, there are some points in both images which cannot be mapped



(a) Fountain-P11 I_1 (b) Fountain-P11 I_2 (e) Herz-Jesu-P8 I_1 (f) Herz-Jesu-P8 I_2

(c) Occlusions in I_1 from I_2 (d) Occlusions in I_2 from I_1 (g) Occlusions in I_1 from I_2 (h) Occlusions in I_2 from I_1

Figure 1.2: Ground-Truth occlusion/visibility maps from Strecha et al. [54]. Each panel shows visibility in both dirextion of the sequences. The **first row** in both panels show the input images I_1 and I_2. The **second row** shows occlusions overlayed in red. These are identified by geometric visibility reasoning on dense 3D point clouds obtained using LIDAR

to any corresponding point; these points are said to be occluded. Thus, the problem of establishing image correspondence is intimately connected to the problem of finding the occlusions or points without correspondence [36]. Notice that pixels might go through changes in perspective, lighting, or even changes in the camera parameters which makes correspondence hard. Optic flow algorithms are left to tackle with most of these situations. Even if a flow algorithm indicates that it cannot find a reasonable correspondence for a certain pixel, it will be hard to tell if this was either due to large change in the visible characteristics of a pixel or because the pixel was actually occluded.

One of the key contributions of this thesis is finding outputs of flow algorithms which are reminiscent of occlusions. Even if we can identify the right set of features to compute over flow, the interaction between these features might be non-linear, making it hard to build a heuristic manually. Hence, to find the techniques that work and those that do not, we will use a classifier which intelligently picks the right combination of features for optimal classification.

1.4 Supervised Learning

Supervised learning, is the name given to machine learning methods where the output is computed by using the correlation learned between the input features and output labels from training examples. Such methods are well discussed in the machine learning domain. A supervised learning method aims to provide, a single, accurate solution given a data-point which might comprise of multiple *features/variables*. The classifier's performance increases with the quality (correlation with the output) of these input features. In this thesis we are more interested in the problem of achieving optimal performance on a classifier rather than reducing our feature set to an ideal size. The reason for this decision will become apparent when we discuss the classifier we use.

Mathematically, a classifier can be considered as follows. Given a feature set $\mathbf{x} \in \mathbb{R}^M$, we need to produce a label Y, such that $g : \mathbb{R}^M \to Y$. Of course, this will be only possible with the perfect set of features - but in usual cases the classifier is obtained in a way to minimize the misclassification rate. Note, the aim of the training phase for a supervised classifier is to define the function g. In our *Occlusion Classification Algorithm*, the label Y is a binary label indicating occlusion or not and $\mathbf{x}$ is the feature set used to find it.

In the cases we will discuss, $\mathbf{x}$ would belong to a single pixel containing a set of spatio-temporal features. We find that the *random forest* classifier is ideal for our purposes (see Chapter 3.1.2), because of its flexibility and performance.

1.5 Algorithm

The components of our *Occlusion Classification Algorithm* are discussed in depth in Chapter 3. Our approach takes inspiration from two methods in particular. It uses a learning framework quite similar to Mac Aodha et al. [29]. Using both temporal and spatial image properties, they suggest training a random forest to predict which of the k flow algorithms would give the most accurate flow on a pixel. A more closely related method is given by Stein and Hebert [50], which attempts to detect occlusion boundaries using motion and appearance cues trained with Adaboost [16].

Like all other supervised learning methods, we first need to train before we test any data with our classifier. The following are the steps used to train our classifier:

1. Given the input sequence, I_1 and I_2, compute flow using the set of k flow algorithms (Section 3.2.2).
2. Compute the set of M features $\mathbf{x} = \{f_1, f_2, \ldots, f_M\}$. Some of these features will use output of the k flow algorithms.
3. Train a random forest classifier sampling pixels for features across multiple sequences. Formally, this step finds the trained classifier function g (see Section 1.4)

Once our classifier is trained, we use the following method to test data:

1. Perform the first two steps done in training for all pixels of the sequence we want to test.
2. Use all the computed features to test all pixels in the classifier. This returns a classification for each pixel i.e. $g : \mathbf{x} \to Y$.

Given these step, we will discuss the construction of *random forests* and the set of our features in Chapter 3 in detail.

1.6 Organization

The thesis is organised into 4 chapters (excluding this one). First we survey the related work in the field of occlusion detection in the next chapter (2). Here, we will also briefly survey techniques in optical flow, motion segmentation, and some pertinent learning frameworks. These are relevant since all of them are related to the problem of finding occlusions. We next discuss our *Occlusion Classification Algorithm* in detail in Chapter 3. Once the framework has been defined, we put it to test in the evaluation chapter (4). We also test it on unseen sequences beyond the initial training and testing data in Chapter 5. Finally we conclude in Chapter 6, with a motivational segmentation example as an application of our framework. The code is listed in the glossary.

Chapter 2

Related Work

This chapter gives an overview of the literature in the area of optical flow (Section 2.1), motion segmentation (Section 2.2), occlusion detection and resolution (Section 2.2.1) and interesting applications of learning (Section 2.3. Since each of these areas have huge corpus of literature, it would be impossible to survey them completely. Apart from discussing seminal works, this chapter will review publications that have found use in *both* optical flow and occlusion resolution. It will also devote a thorough section on motion segmentation techniques and, lastly, algorithms that apply (supervised) learning schemes.

2.1 Optical Flow

As discussed in Section 1.2, computing motion flow has a lot of uses in many computer vision algorithms. Yet for them to be used as effective tools, they need to resolve motion accurately even at motion boundaries. This section will explore a very small section of literature related to optical flow which either concentrates on motion / intensity discontinuities, or smoothly / linearly varying regions using some special scheme.

One of the earliest approaches for dense motion estimation, Heitz and Bouthemy [23] uses multiple complementary constraints in an attempt to preserve motion at boundaries. The first constraint it uses is on Gradient-based motion. Initially proposed by Horn and Schunck [24], it suggests:

$$\vec{\nabla} f(s).\vec{w}_s + f_t(s) = 0$$

where $f(s)$ is the velocity vector at a particular point, $\vec{\nabla}$ is the spatial image gradient, and f_t is the temporal intensity gradient. This equation shows that only the flow parallel to the spatial image gradient can be recovered if we only rely on local computation. This is the well known *aperture* problem. This was tackled earlier by either supposing smooth velocity variations across the image or invariant velocity in a small neighbourhood. The second edge-based motion constraint is used by thresholding the log-likelihood ratio test based on parameters defined on a surface, where each parameter is tuned to detect edge-locations, orientation and displacement. This paper introduces validation of these constraints using hypothesis testing. This is followed by using image features as observations in a Bayesian estimation process, used to extract motion labels for each portion in the image. The relationship between the observation fields and the labels in this process is specified using an MRF.

It has been argued that the constraints used in the Bayesian method significantly increases the computational complexity of the method. To tackle this, Chang et al. [15] introduces interdependence between the optical flow field and the segmentation map directly through the Bayesian framework. It uses a motion field as a sum of a parametric field and a non-parametric residual field. This helps it conveniently resolve segmentation and optical flow by simply finding the parameters for the parametric field using a least squares solution. The optical flow field is refined by computing the (non-parametric) minimum-norm residual field given the best estimate of the parametric field, under the constraint that the motion field needs to be smooth within each segment. This gives the best estimates for the motion field and segmentation. The segmentation portion of this method will discussed further in Section 2.2.

Shi and Malik [46] introduces a graph based approach to estimate flow fields and motion segmentation (discussed in Section 2.2). It forms a graph $G = (V, E)$ which are pixels connected in its spatiotemporal neighbourhood with weights ($w(i, j)$) denoting the similarity of motion between two pixel nodes. Although only motion weights are used, it mentions that weights can also be based on measures such as colour, brightness, texture, and disparity. It takes the approach of committing motion vectors later in the pipeline, by initially working with just a *motion profile*: the probability distribution of the image velocity at each pixel in the image, which captures both direction and uncertainty:

$$\frac{1}{Z}\exp(-\alpha \mathrm{SSD}[I^t(x_i), I^{t+1}(x_i + u)])$$

where I^t and I^{t+1} are the two image patches and α is the weighting, and Z is the normalisation constant. The weight on the graph is taken as the cross-correlation of the two motion profiles given by P_i and P_j:

$$w(i, j) = \exp(-[1 - \sum_{dx} P_i(dx)P_j(dx)]/\sigma_m^2)$$

where σ_m^2 is the expected variance in the motion profiles. It should be noted that this measure of motion similarity will, indeed, distinguish between two pixels which have exactly the same true motion, but different brightness profiles, which would result in difference in associated motion uncertainties. This will happen if one of the pixels is in a region of constant brightness and another in a region of rich texture. The method handles this in a post-processing step. Eventually the algorithm summarises and solves all motion profiles between each pair of pixels as an eigenvalue problem.

In a related approach, Galun et al. [19] proposes to iteratively improve motion flow estimates by solving systems with increasingly complex motion models. At each level of complexity it chooses seed pixels for computing optical flow using Shi and Malik [46]. It uses the same cross-correlation method to compare motion profiles, after smoothing each profile with a Gaussian. These seed pixels are later associated to individual clusters signifying common motion. A re-estimation step solves for the selected motion models (according to the iterative level) while estimating a common motion for the cluster. As discussed in Section 2.2, this helps in combining motion by supposing that all pixels are strongly associated with a subset of the selected seed pixels.

Following the same idea of employing increasingly complex motion models, Wong and Spetsakis [61] proposes another motion segmentation method for tracking objects on a static background. The tracker, in this technique, needs to be initialised with a small seed window which falls inside the to-be-tracked object in the first frame (it proposes a 10×10 pixel window). Using Least SSD, it attempts to estimate the initial optical flow vectors:

$$\mathrm{LSSD}(u, v) = \min_{(u,v)} \sum_{x,y \in S} (I_{t-1}(y + u, x + v) - I_t(y, x))^2$$

where S is the seed window. Since the seed window is small, the search window is set to be relatively large, going from $-30 \cdots 30$ pixels. To refine the initial estimate of (u, v), it aligns I_{t-1} to I_t (using the initial (u, v)) and attempts to compute the sub-pixel optical flow. This is done by moving the window successively in the 8 surrounding directions and finding the location which minimises the SSD. The motion segmentation technique that follows is discussed in Section 2.2.

2.2 Motion Segmentation

Motion segmentation techniques combine the *segmentation* and *optical flow* paradigms to solve both problems in a unified way. In literature, there are two main categories of motion segmentation schemes. One common scheme is to create segmentations independently across frames. These segments would show some similarity in motion and image features. The second scheme is more semantically meaningful, where segments in a frame are corresponded with segments in other frames. This technique aims to produce whole temporally consistent segments. In this section we will discuss a small section of the large volume of literature available on this topic.

Black and Jepson [6] proposed one of the earliest methods to constrain motion to regions using the brightness information. The solution given can be divided into two stages: early processing and medium-level processing. In this two-stage process, it attempts to estimate optical flow through motion of planar regions and local deformations. These deformations are allowed in the model since the assumption of planarity is likely to be violated in any natural scene. The segmentation, coupled with this method, is done on brightness values to constrain motion to planar regions, as initially proposed in Black [4]. It uses an analog spatial outlier process to define discontinuity between pixels - also defined is a penalty term which needs to be paid with increasing discontinuity. All these steps are performed to eventually minimise an objective function with a data term and spatial coherence term. The first stage of the method estimates a coarse fit to the parametric model and evaluates a set of parameters for each region. It consists of two low-level processes: a process that smooths image brightness while checking for discontinuities; and one that estimates motion. The second medium-level processing stage refines the initial fit of the parameters with “standard area-based regression approaches”. The main aim of this stage is to collate the low-level information from the first stage by connecting piecewise smooth brightness regions and estimating their motion. This is done in three steps: (1) fit a translational/affine/planar model which best captures motion in a region, (2) using this model, warp the regions into alignment, so a Gradient-based optical flow method can help in refining the initial parametric model, (3) allow deformation of low-level patches to improve the motion estimate of each planar patch.

As introduced in Section 2.1, Chang et al. [15] takes a slightly different approach to motion segmentation - by finding a parametric field, that optimises the motion field and segmentation. This is done using a least squares solution. After refining the estimates for the flow field, the segmentation field is also improved to give the minimum-norm residual field using Gibbsian priors. The least squares estimates of the mapping parameters Φ for each segment is computed in closed-form given the MAP estimate:

$$(\hat{u}, \hat{v}, \hat{S}) = \max_{u,v,S} P(I_t|u, v, s, I_{t-1})P(u, v|s, I_{t-1})P(s|I_{t-1})$$

where (u, v) are motion field vectors, I_t (current frame), I_{t-1} (search frame) are the two frames, and s is the segmentation field. The conditional PDF $P(I_t|u, v, s, I_{t-1})$ quantifies how well the motion and segmentation estimates fit the given frames. This PDF is modelled by a Gibbs distribution. $P(u, v|s, I_{t-1})$ is also modelled by a Gibbs distribution with a potential function which aims to minimise the deviation of the motion field (u, v) from the parametric motion (u_p, v_p). The third term $P(s|I_{t-1})$, the priori probability of the segmentation, also follows a Gibbs distribution to discourage formation of small, isolated regions. The method also proposes dense representation of the residual field for improved motion segmentation.

The graph based approach due to Shi and Malik [46] aims at finding motion segments which minimise the normalised cut Shi and Malik [47]. Once the graph is constructed (as discussed in Section 2.1), normalised cuts across the graph will give spatiotemporal volumes corresponding to different moving objects. This technique segments the scene since normalised cuts not only reflect similarity within a partition but also dissimilarity across partitions. Combining the weights $w(i,j)$ in a matrix $\mathbf{W}$, and a diagonal matrix $\mathbf{W}$ where $\mathbf{D}(i,i) = \sum_j w(i,j)$, the method solves the generalised eigen-system $(\mathbf{D} - \mathbf{W})\mathbf{y} = \lambda\mathbf{D}\mathbf{y}$ for the smallest eigenvalues. The graph is then partitioned by Ncuts with the eigenvector belonging to the second smallest eigenvalue. The segment is only used if it is stable (by checking the cut cost). As a result, time slices of the output segments indicate corresponding groups across time.

Since [46] groups pixels based on the affinity of the motion profile, a local measurement, it ignores global constraints and appears unstable in noisy sequences. An approach based on *Geodesic Active Region* due to Paragios and Deriche [39] tackles this problem by incorporating a *Visual Consistency Module* which tries to optimise the segmentation map globally (while keeping track of motion). The main theme of the paper is simultaneous tracking of multiple non-rigid objects. A more generalised formulation of this approach can be found in Paragios and Deriche [40]. The method, at its heart, has a curve-based objective function formed with boundary and region-based terms. This final objective function is minimised using gradient descent methods. The boundary terms aim to find a minimal length contour attracted to region boundaries. On the other hand, region-based terms aim to maximise the quality of the segmentation map. Intuitively, the initially proposed curves are propagated toward the best partition under the influence of boundary, intensity and motion-based forces. The method assumes object motion can be described using global affine model $A(x,y)$.

This technique also incorporates a motion detection module which uses the difference frame to create parametric (Gaussian or Laplacian) distributions which can be used to model static and mobile pixels. The parameter in these distributions are estimated using the Maximum likelihood principle. Also particular to this method is the intensity segmentation module - which moves the curve in the direction which creates interior regions with desirable intensity properties.

Another method which considers an affine motion model was proposed in Wong and Spetsakis [61]. Its initial estimation of the flow field was discussed in Section 2.1, which resulted in sub-pixel estimates of (u,v). This helps align the image segments I_{t-1} and I_t. Since the computation of the affine flow can be costly, the technique uses the initial translational estimates to identify areas where to compute an affine model. This model is then built using a differential approach which computes the standard least squares for the following equation:

$$\mathrm{SSD}_{\mathrm{affine}} = \sum_{\text{all } x,y} {}^{(t-1)}I_{\mathrm{track}} \cdot \left[I_{t-1}(x,y) - I_t(x,y) + I_{t-1,x}(x,y) \begin{bmatrix} u_0 \\ u_x \\ u_y \end{bmatrix} + I_{t-1,y}(x,y) \begin{bmatrix} v_0 \\ v_x \\ v_y \end{bmatrix} \right]^2$$

where $\begin{bmatrix} u_x & u_y & u_0 \\ v_x & v_y & v_0 \end{bmatrix}$ are the affine parameters, $I_{t-1,x}$ and $I_{t-1,y}$ are the image derivatives in the x and y direction, and ${}^{(t-1)}I_{\mathrm{track}}$ represents the region on which the affine optical flow is being computed. The affine parameters are computed by minimising $\mathrm{SSD}_{\mathrm{affine}}$ for all pixels in the region. By aligning all the previous images to the last image using these affine parameters, it segments out the object to-be-tracked by thresholding the pixel-wise SSD. This is a reasonable measure since the SSD of the aligned tracked object would be small. This process is made computationally tractable by aligning just the first and second moments of images (see [61] for details). To segment out the moving object, the threshold is set keeping in mind the camera and motion noise.

As discussed in 2.1, Galun et al. [19] also iteratively improves its motion models in an effort to segment motion. The approach involves iterating over two steps: *clustering* and *re-estimation*; where each iteration is stated as a *level*. Apart from computing the optical flow for each seed pixels in the *clustering* step, it also computes how strongly each pixel is associated to a particular seed. The aim of the *re-estimation* step is to estimate the common motion of each cluster. This coarsening step begins by selecting a subset of the elements from the previous level as seeds, with the constraint that all other elements are strongly associated with (subsets of) these seeds. To aggregate pixels, the motion profile is computed by multiplying all the *child* motion profiles (see [19] for details) - this technique gives sharply peaked motion profiles in textured regions in just 1-2 coarsening steps (note, this scheme doesn't work in uniform regions). These *peaked motion profiles* help accumulate moments of respective seeds. With increasing levels, the aggregates/segments become large and translational motion stops reflecting the true motion. If there are enough constraints available in the *peaked motion profiles*, an affine transformation for the segment can be computed. The paper also describes computing a projective transformation with the fundamental matrix at higher levels. Finally, these motion parameters are combined with intensity cues by using Segmentation by Weighted Aggregation (SWA) (Sharon et al. [45]) to give the output segments.

Stauffer and Grimson [49] also aims to learn patterns of activity in a scene. Their approach is unique amongst the works mentioned here, as it proposes a system which fuses motion information from multiple-cameras (each sensor has some location awareness). Motion segmentation is done using an adaptive background subtraction method where each pixel (process) is modelled as an adaptive mixture of Gaussians, using online approximations to update this model. Each time the parameters of this model are updated, it uses a heuristic to find whether the pixel belongs to a background process or not. Every new pixel X_t is checked against K Gaussian distributions until a match is found (it considers a value within $\mu \pm 2.5\sigma$ of a distribution as a match). If none of the distributions match the current pixel, the least probable distribution is replaced by a new distribution, with a mean of X_t and a high variance and low prior weight $w_{k,t}$. Finally, the Gaussian distributions, having more evidence (prior weight) and less variance (sort distributions by w/σ) are taken as part of the background. Pixel values that do not match any of the background Gaussians, are grouped together using connected components. These connected components are tracked across time using a multiple hypothesis tracker, hence resulting in segments with their motion.

To discuss a subspace method for motion segmentation, we will turn our attention to Zelnik-Manor et al. [65]. This paper tracks using subspace methods applied directly to features pixel intensities. It organises the problem as a multi-body segmentation using a flow field matrix $[U|V]$ (where both U and V are matrices of directional motion vectors of Frames $\times$ Pixels dimensions) giving rise to multi-body factorisation with directional uncertainty. While segmenting $[U|V]$, the decision of grouping together two objects is based on the ranks of the matrix - which exploits the linear dependency of flow fields of a single object. Since the flow-field matrix $[U|V]$ is of rank 1, there exists a set of basis trajectory vectors and a set of basis flow-fields such that they can be factored into two matrices:

$$[U|V] = \underbrace{C}_{\text{flow coefficients}} \quad \underbrace{B}_{\text{flow basis}}$$

In addition, since tracking might not be reliable for all feature points, it introduces directional uncertainty by a weight matrix $[U|V]Q$. Irani and Anandan [25] proves that this results in the covariance-weighted measurement matrix ($[G|H] = [U|V]Q$). Finally, segmenting the entire video into a set of moving objects is now a question of sorting the columns of the covariance-weighted measurement matrix. This is easily done by finding its RREF. Since the rank of $[G|H]$ is considered to be small, RREF is computed on the SVD of $[G|H]$.

The reader is referred to Megret and DeMenthon [34] for a more detailed survey (and taxonomy) of motion segmentation techniques.

2.2.1 Occlusion resolution

Occlusion is a critical concept algorithms have to deal with in motion segmentation. The literature discussed belongs to either occlusion resolution in a *layered* representation or boundary occlusion resolution.

Wang and Adelson [57] uses the term *layers* for moving objects. The method basically gives a method for motion segmentation where each segment becomes a *layer*. Each *layer* is defined using an intensity map and an alpha map (to denote each pixel's transparency). Velocity maps are used to show how *layers* can move in time. Each *layer* is also assigned a depth ordering which follows the rules of compositing. Since optical flow *models* moving objects like "rubber sheets", it argues that this motion assumption breaks down in case of occlusion. It supposes that regions undergoing similar affine motion result from the same plane in the world - and to deal with boundary motion discontinuities, it allows sharp breaks in the flow field using regularisation. These discontinuities are explained by the framework as instances of occlusion. For an initial estimate of motion, optical flow is computed in a local neighbourhood region, as suggested by Bergen et al. [3]. Using these initial estimates of optical flow, it determines a set of affine parameters which are likely to be observed. The scene is then segmented, by an iterative process, which classifies regions of the motion model that provides the best description of the motion withing each region. This technique relies on k-means clustering in affine parameter space. This method was arguably the first to develop the idea of an affine model for flow.

Following a layered based approach using colour segmentation as an input to its stereo algorithm, Zitnick et al. [66] employs an interesting matting technique in order to interpolate views from multiple images. Since boundary pixels might receive contributions from foreground and background objects, it argues that the same colour distribution for boundaries in new views might look unnatural. It approaches this problem as an overlap of layers. In locations of depth discontinuities, matting information is computed in the 4 neighbourhood pixels. Within this neighbourhood, foreground and background pixel colour with alpha values are computed using Bayesian image matting. The information recovered for the foreground is used to compute a new boundary layer. This is useful in generating novel views since the boundary layer can be rendered with different levels of opacity.

Zitnick et al. [68, 67] also computes motion segmentation while accounting for matting in overlapping regions (modelled with α). It proposes a matting model where each pixel can belong to two segments (out of the K segments) with a corresponding association of α and $1-\alpha$:

$$c_i \approx \alpha_i c_{i,s_i^1} + (1-\alpha_i) c_{i,s_i^2}$$

where s_i^1, s_i^2 denotes the two segments, i is the pixel index, c_i is the colour contribution of pixel i, and c_{i,s_i^x} is the colour contribution of pixel i from segment s_i^x.

Another approach using layers to explicitly solve occlusions is given in Kumar et al. [26]. Overall, the technique is an unsupervised approach to generative layered representation for motion segmentation. It learns each rigidly moving object in a sequence and uses it in a layer. Once the parameters of the model have been estimated, any one of the frames in the sequence can be generated by selecting the right *latent variables*: represented by transformation, appearance, layer ordering and lighting parameters. An initial estimate of the model parameters is made using patches in an MRF framework, which segments motion using a loopy belief propagation technique. Given these initial model estimates, one of the methods it uses to improve the shape of the segments is α expansion. It expands each segment and checks for overlaps to see if either layer A occludes layer B or vice versa - by checking which configuration minimises the energy of the layered representation.

Ogale et al. [38] introduces an interesting formulation where it uses occlusions themselves for motion segmentation. It highlights 3 categories of object motion to differentiate motion due to camera and scene elements. The ordinal depth is computed according to the categorisation of the motion. If an optical flow estimate is provided, the method suggests a simple scheme to assign occluded regions to the right layer. 3 frames F_1, F_2, F_3 are given, with their optical flows u_{12} and u_{23}, and their reverse optical flow u_{21} and u_{32}. From previous computations, the method knows regions of occlusion O_{12} (regions present in F_1 but not in F_2), and O_{23}. We first consider finding the region labels of occluded regions in F_3, O_{23}. Given that the regions occluded in F_3 would have been visible in F_1, F_2; the method can segment the flow of u_{21} to find the labels for regions that subsequently got hidden in F_3 i.e. O_{23}. This helps in finding ordinal depth of the occluded regions.

Xiao and Shah [62] assumes planar regions and extracts a set of affine or projective transformations that these regions undergo. In order to do this, it detects the occlusion pixels and segments the scene into motion layers. Using a level set formulation and *graph cut*s it creates an initial set of segments. In a two step process, it merges similar segments into layers on the basis of motion similarity. In the first step, two regions R_1, R_2 from the initial layers are merged if the SSD of the two regions, after applying the transformation H_2 on R_1, results in a majority of the pixels in R_1 supporting R_2. The motion parameters are recomputed after the merger. In the second step, the bi partitioning *graph cuts* algorithm is used to prune the un-supporting pixels from this new merged $R_1|R_2$ region. This results in layers, said to have coherent motion. Finally, it finds occlusion between overlapping layers using the graph cuts algorithm. This is done while ensuring consistency of layer segmentation using occlusion order constraints.

Stein and Hebert [50] also models motion discontinuities as occlusion events. After segmentation (using watershed on non-local maxima suppressed P_b [32] edge map) and motion estimation, it attempts to identify occlusion boundaries (which would belong to multiple surfaces). To detect such a moving motion edge, it considers the movement of the edges in a spatio-temporal volume (see Figure 2.1a). Since the tangent of the angle of this path (temporally) would correspond to the orientation and speed of edge's motion, it applies an oriented edge detector to the temporal slice of this volume. Here the choice of the filter is important. It uses a cylindrical detector capable of dividing data into two halves aimed at detecting significantly different distributions of motion (see Martin et al. [32]). Using this filter, it combines local normal speed estimates along the edge to get full 2D (u, v) motion estimate of the whole edge fragment (see Figure 2.1b). It uses a linear system of equations based on fragment's overall motion vector projected onto the local normal vectors:

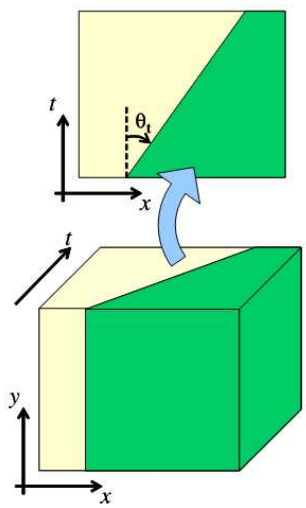
$$\begin{bmatrix} u \\ v \end{bmatrix} = \arg\min \sum_{i \in F} w(i)(n_{x,i}u + n_{y,i}v - s_i)^2$$

where $n_{x,i}$ and $n_{y,i}$ are the components of the normal at point i on the edge fragment F and s_i is the corresponding speed from the spatio-temporal detector. w_i denotes the contribution according to the local edge strength. These detected edge segments are critical to this technique, as they are associated with a set of motion-based and appearance-based cues. These cues are critical in computing the likelihood that an edge fragment is an occluding contour (see Figures 2.1c and 2.1d).

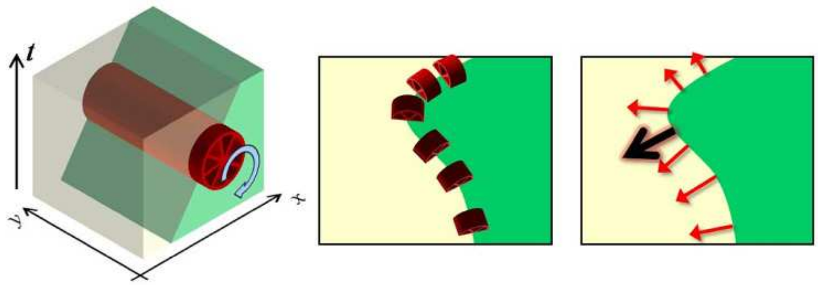
2.3 Learning (Feature selection)

This section discusses a few works in feature selection as a classification task. The first two papers are generic machine learning papers, while the papers in the later half are related to feature selection in the domain of image segmentation, tracking and optical flow.

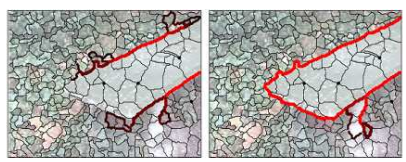
Muja and Lowe [35] concentrates on a common problem in many computer vision algorithms: nearest neighbour matching in a high-dimensional space. More formally, given a set of points $P = \{p_1, \cdots, p_n\}$



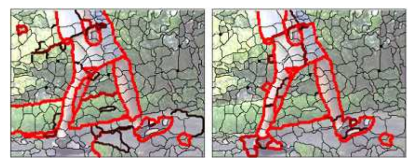
(a) A moving edge sweeps out an oriented path θ_t in a temporal slice



(b) A representation of the spatio-temporal edge detector used (left), shown at each point along a boundary fragment (mid), each detector will yield a combined normal (black) vector (right)



(c) Detection of occlusion boundary shown in red. Output generated with appearance (left) and motion+appearance (right) cues. The gray value of the edges denote the strenght of the edge



(d) Same as 2.1c shown on a different input

Figure 2.1: Shows the steps involved in resolved occlusion boundaries in Stein and Hebert [50]. Taken from Stein and Hebert [50]

in a vector space X, and a query point $q \in X$, it tries to find the approximate nearest neighbour for q in P. To tackle this, the paper introduces a new method of approximate search which is a modification of the *hierarchical k-means tree*. But, in this section we are more interested with the algorithm selection technique it proposes. Given a dataset, this technique uses a cross-validation scheme, to select the ideal algorithm and its parameters. The paper demonstrates using this technique to select either *randomized kd-tree* or *hierarchical k-means tree* and suggests a set of parameters e.g. the number of iterations to use in case of the latter. The choice of algorithm is made by evaluating a cost of each algorithm, computed on a small subset of the input data:

$$\text{cost} = \frac{s + w_b b}{(s + w_b b)_{\text{opt}}} + w_m m_t / m_d$$

where s represents the search time, b represents the tree build-time, and m_t is the memory used for the tree, and m_d is the memory to store data. w_b is the weightage provided by the user for build-time i.e. when $w_b = 0$, the user isn't concerned with the build time. Similarly, w_m gives the importance of memory overhead against time-overhead. The time overhead is computed relative to $(s + w_b b)_{\text{opt}}$, which is the optimal search and build time if memory usage is not a consideration. Since the cost is built only to deal with nearest neighbour searches, adapting it to another method would require changing the cost altogether.

As a more general scheme, Raykar et al. [41] describes the problem of selecting the right label for a classification task given a number of algorithms (where the labels from algorithms might vary by a substantial amount) and no gold standard is given (for the classification problem). In order to find the true label, it presents a MAP estimator, which also learns the classifier, and the algorithm accuracies in parallel. The performance of each algorithm is judged on sensitivity and specificity with respect to the gold standard. Using these performance rating, each algorithm is ranked and assigned weightage that will be useful in all subsequent classification tasks. The EM algorithm is used to iteratively establish the final gold standard, using the performance measures from all the algorithms. The paper also mentions the possibility of using a beta prior (on sensitivity and specificity), in-case some prior knowledge about a particular algorithm needs to be asserted. This formulation is an improvement over a majority vote technique, which fails when some algorithms are more trustworthy than others.

An approach closer to our application is given in Yong et al. [64], which proposes to find the ideal segmentation algorithm for a particular image. The system can be categorised as a "performance prediction based algorithm selection model". This performance prediction is based on both image features, and prior knowledge and experience. Two assumptions are made to make this technique work: (1) the algorithm ideal for segmentation depends on image characteristics such as contrast, noise, and illumination; (2) the performance of an algorithm on images with similar characteristics is approximately the same. The method constitutes of three modules: (1) *performance predictor* which defines a prediction function based on segmentation quality of different algorithms, at the training stage; (2) *performance evaluator* is used in the training stage to rank the algorithms - which is a supervised offline scheme; (3) *feature extractor* which uses a simple intensity histogram. The ranks and the features extracted are eventually used to train the *performance predictor*. At runtime, the *performance predictor* ranks the segmentation algorithms according to the input image's (extracted) features. The paper demonstrates selecting the best algorithm in 85% of the test cases.

Moving from domain of segmentation, Stenger et al. [51] defines a similarly structured technique for tracking. Since tracking is an iterative process, it needs to learn the model of the to-be-tracked object, in hope of detecting it in subsequent frames. Here, an issue that can plague a learning technique in tracking, and any other iterative classification approach (unlike segmentation), is that of adaptability against drift. The tracker can learn (adapt) the object model quickly but it will be more susceptible to drifting off (by erroneously learning the background model). To avoid this scenario, [51] suggests using a trained offline detection component. Each tracker is also evaluated during this training stage. Every tracker O^k outputs the estimate of the tracked position x_t^k on a training set, its error e_t^k from the ground-truth, and a confidence value c_t^k. Selecting a particular threshold on error, the loss of track can be identified automatically. Using these measures, *precision* ($1-$ expected error) and *robustness* (the probability of keeping track) for each tracker is noted. The method proposes two schemes for online tracking. The first one is a *parallel* scheme, where multiple trackers are used simultaneously. The tracker with the lowest expected error given the confidence value ($\min_k E[e^k|c^k]$) is used for the output. The second is a *cascaded* scheme, where trackers are used sequentially until a tracker is found whose expected error is below a threshold ($E[e^k|c^k] > \tau \rightarrow$ evaluate next tracker). The advantage of using a sequential approach is the reduced computational time.

Mac Aodha et al. [29] suggests a novel approach to evaluating optical flow by choosing ideal algorithms per-pixel. It supposes that some form of gold standard is available during the training phase but not during runtime. Like Yong et al. [64], it also tries to find a relation between good performance of an algorithm against specific spatial and temporal features. Experimental results given, use Random forests (see Breiman [9]) to classify which flow algorithm (4 algorithms considered - $k = 4$) to use at a particular pixel. A total of 44 features were input to the classifier which included measure of textured-ness, distance from edges, derivative of proposed flow field (to be wary of motion discontinuities), all at different image pyramid levels. The best features were selected by the classifier and used in the algorithm selection process.

As an added note, if we consider interest point detectors, like Shi and Tomasi [48], as binary classifiers on image features, they already provide a rudimentary base for algorithm selection. Although an advantage of these schemes is that no energy functional needs to be employed over the complete data to get optimal results.

Chapter 3

The Occlusion Classification Algorithm

This chapter describes a method for finding regions of occlusion in a two frame sequence. We take the machine learning approach to solve this problem, where each pixel is considered as an entity and is trained and tested with a classifier using features associated to it. More formally we have features $\mathbf{x}$ for each pixel, and we seek the output label $Y \in \mathcal{Y}$, where $\mathcal{Y}$ is *Categorical* indicating if the data-point is occluded or not (binary/*Bernoulli* to be precise). Since we use *random forests* for classification, the output is a posterior probability for $\mathbf{x}$ taking the positive label (occlusion). In short, our method takes an input sequence; computes features on it; classifies each pixel; and outputs an occlusion probability map. We can later threshold this probability map to obtain a binary labeling.

As explained in the rest of this chapter, the reason for adopting a learning approach is because the relationship between our features $\mathbf{x}$ is non-linear, and developing a heuristic would be hard if not impossible. Although one drawback of using a learning technique like random forests is the lack of any spatial regularization during training or testing. This is rectified by using features which establish relationships over pixel neighborhoods.

The chapter is divided into two main sections. Section 3.1 describes the *learning* method we use. After discussing the building blocks of *forests*, *Classification Trees* (Section 3.1.1), we describe *Random Forests* in detail (Section 3.1.2). Here, we also include a discussion on alternate learning approaches (Section 3.1.4). After deciding on a learning method, Section 3.2 describes the features/variables that will be input to the *forest* for training and testing occlusion regions. This section describes features based on image properties (Section 3.2.1) and features based on flow (Section 3.2.2). All algorithms used herein are implementations by the author of this thesis, unless stated otherwise.

3.1 Learning

There are two broad approaches to any classification task. The first is to manually develop an algorithm which works on heuristics. The second approach is based on *learning* - which, in short, employs some technique to automatically find the relationship between your features $\mathbf{x} = \{f_1, f_2, \ldots, f_M\}$ and the output label. Given that the interaction between the features and the output can be complex, the latter approach outshines many heuristic solutions. In this section we will discuss the *learning* side of our algorithm, and later we will explore our feature vector $\mathbf{x}$ in Section 3.2.

Our chosen *learning* approach is the randomized decision forests, or more simply, *random forests*. We have selected it since it is a fast classifier promising low error rates comparable to *boosting*, and has found wide use in both data mining and machine learning [9]. Before we discuss random forests (Section 3.1.2), we will first explain its building blocks - "classification trees".

3.1.1 Classification Trees

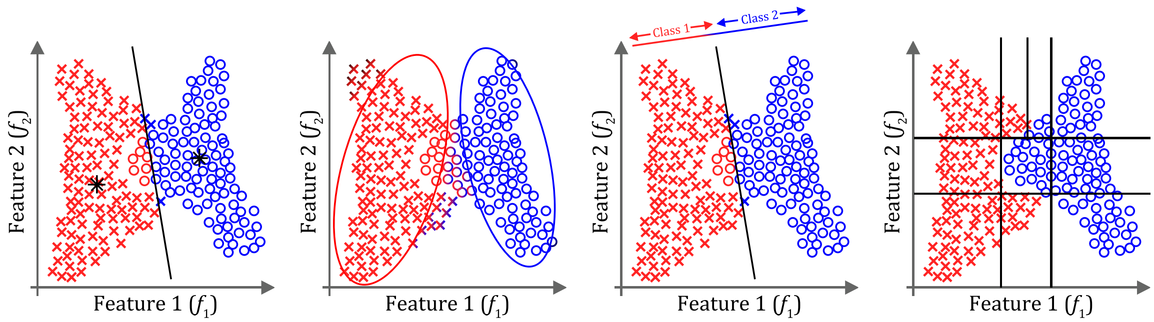
Like any other classification or regression task, the goal of a *Decision Tree* is to predict an output variable Y, given a set of input variables $\mathbf{x}$. When $\mathcal{Y}$, the domain of Y, is a continuous or discrete variable taking real values, decision trees take the form of *Regression Trees*; on the other hand, when $\mathcal{Y}$ is a finite set of unordered values, we need to use *Classification Trees*. The terms *decision trees* and *classification trees* are sometimes used interchangeably. In this section we will be more concerned with classification trees as our $\mathcal{Y}$ contains only 2 unordered values: 1 for *occlusion*, 0 otherwise. In mathematical terms, a classification tree is concerned with finding a function $g(\mathbf{x}, \mathcal{T})$ which maps each value in $\mathcal{X}$, the domain of $\mathbf{x}$, to a value in $\mathcal{Y}$. The construction of $g(\mathbf{x}, \mathcal{T})$ (building the tree) requires a training set $\mathcal{T} = \{(\mathbf{x}_1, Y_1), \ldots, (\mathbf{x}_N, Y_N)\}$. In the domain of machine learning, this technique is categorized as supervised learning or "learning by example". There are many criteria for choosing an $g(\mathbf{x}, \mathcal{T})$ - expected misclassification cost $E\{g(\mathbf{x}, \mathcal{T}) \notin E(Y|\mathbf{x})\}$ being a popular choice, where $E(Y|\mathbf{x})$ is the expected value of Y at $\mathbf{x}$ [28].

In their essence, classification trees are quite simple. Starting with the complete input data D from the root node, each *internal/decision node* in the tree has a question based on one of the variables x_i. This heuristic could be of the form $x_i \geq T_i$ in case x_i is real valued, or $x_i \in V_i$ if x_i is categorical. The data is split according to the answer of each data point D_k to this question. Once the data is split, each of the respective splits is subjected to the questioning on the next internal node. Following this "recursive partitioning", all the data is split down the tree until each D_k point can be definitively classified. The nodes where the method decides to stop splitting, are the *leaf node*. Since the framework revolves around splitting over chosen thresholds, it has no need for normalizing variables.

Instead of a tree, there is another way to view this form of classification. Given the input space $\mathcal{X}$, this *recursive partitioning* can be viewed as splitting of the input space into smaller sets. Mathematically, if $\mathcal{Y}$ contains J distinct values Y_j, the classification output can be viewed as a partitioning of $\mathcal{X}$ into J disjoint pieces $A_j = \{\mathbf{x} : g(\mathbf{x}, \mathcal{T}) = Y_j\}$ such that $\mathcal{X} = \cup_{j=1}^{J} A_j$. Figure 3.1 shows the comparison of clustering from classification trees against some popular unsupervised methods. Figure 3.1d clearly shows that each partition A_j itself is constructed from a union of smaller disjoint sets. This is equivalent to combining data on leaf nodes which have the same classification Y_j.

Given the recursive splitting of the classification tree, three questions come to mind: (1) how to choose the heuristic for partitioning (how to build the tree), (2) when to stop the recursive partitioning, (3) how to predict the value of Y for each $\mathbf{x}$ in a partition. For the first task, many methods employ univariate binary splits, e.g. $x_i \geq T_i$ and $x_i < T_i$. We can also have n-ary splits, but we are not concerned about them since such splits can be represented by multiple binary splits. In building a tree, the interesting question is the choice of the variable x_i and the splitting threshold T_i to use at a *decision node*. In short, methods try to make a choice which leads to information gain. This usually involves an exhaustive search making it one of the expensive operations while constructing a tree. We will discuss three methods below for this tree building operation.

There are many methods for performing the second task; one being to recursively partition until partitioning is not possible, i.e. until each node only has data belonging to one class Y_j. After the popular CART method [11] generates this "maximal tree", it examines smaller trees, obtained by pruning away



(a) k-means (b) MoG (c) LDA (d) Classification Trees

Figure 3.1: Comparison of 4 clustering methods. Although the first 3 methods are unsupervised and can only be used for classification (not regression), it is still interesting to view their clustering techniques in comparison to *Classification Trees*. k-means (shown in 3.1a) clusters data according to the nearest mean, while Mixture of Gaussians (shown in 3.1b) fits multi-variate gaussians to encompass the data. In contrast, Linear Discriminant Analysis finds a linear transformation which best separates the data (the axis is shown in 3.1c). Classification Trees (3.1d), with the help of training samples, partitions the input space, concentrating more on areas which are harder discriminate.

branches of the maximal tree (see Bradski and Kaehler [7] for an example on pruning). Figure 3.2 shows why pruning is important for tree's generalization capacity, since without it, the tree learns the training data perfectly (completely over-fits). Other methods for finding the stopping condition include recursive partitioning until the number of data points in a node reaches a set minimum.

The third task of predicting the output variable Y at a leaf node is quite simple. In classification the output variable Y is the class that minimizes the estimated misclassification cost.

To discuss any decision node selection techniques, we first need to define some terms: let $N_j(t)$ be the number of data points which belong to class Y_j at node t; and let $N(t)$ be the total number of data points at node t. Now we can define the estimated probability that a data observation at node t belongs to class Y_j as $p(j|t) = N_j(t)/N(t)$. Moreover, let $D(t)$ be all the data observations those end up at node t. The complexity of all the techniques below depends on the total possible splitting points T_i. If x_i has v distinct values, the possibilities for T_i are $v - 1$. In-case, x_i is a categorical variable, the number of possible splits is $2^{v-1} - 1$.

There are many heuristics to find the best split: C4.5, C5.0, gain ratio, Gini, to name a few. We will discuss 3 below:

Entropy

Since we are aiming for maximum information gain, entropy is a natural criterion for selecting the variable x_i and the splitting point T_i in order to partition the data. The measure of entropy "impurity" is:

$$i(t) = -\sum_{j=1}^{J} p(j|t) \log_2 p(j|t)$$

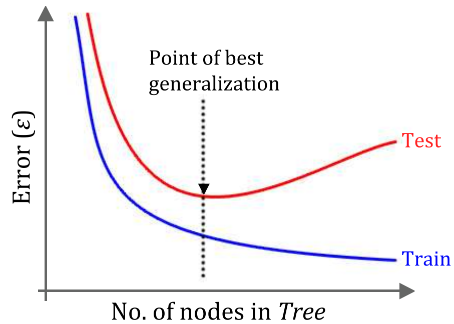


Figure 3.2: The graph shows a typical case of training and testing errors with the increase in the number of training cycles (number of nodes in case of a *Classification Tree*). The practitioner of any supervised learning method aims to find the point of best generalization. Beyond this point, more training data results in *overfitting*.

To use this impurity factor for splitting the data, we will need to examine each variable x_i in turn. If T_i partitions the data to $D(r)$ and $D(l)$, we can compute the decrease in entropy as follows:

$$I(t) = i(t) - \frac{N(r)}{N(t)} i(r) - \frac{N(l)}{N(t)} i(l)$$

where $N(r)$ and $N(l)$ is the number of observations distributed to each of the respective child nodes after the split $(N(r) + N(l) = N(t))$. We can find the ideal variable and split value by finding the maximum $I(t)$ across all variables and all splits.

Gini index

Gini "impurity" work similarly as Entropy [1]. The only difference being the "impurity formula:

$$i(t) = 1 - \sum_{j=1}^{J} p^2(j|t)$$

Now $I(t)$ is computed as before. This impurity function is specifically important for our work since the CART method [11] by default employs Gini impurity, and Random Forests builds trees using CART. To illustrate the Gini impurity, Figure 3.3 builds a classification tree using this technique. In all the data tables, the sub-scripted dark-blue values are $I(t)$ computed using the Gini index. Data is partitioned using the variable and the split value which gives the maximum $I(t)$ (given in white).

One important aspect to note about both the Gini and Entropy measures is that they are biased towards variables with more missing data. During split selecting, if a variable x_i has missing values, only the observations non-missing in both x_i and Y are used in computing the decrease in impurity. This makes it easier to "purify" a node by splitting using a variable with more missing values [28].

[1] A simple `MATLAB` code for a *Classification Tree* using Gini "impurity" is given in Appendix .1

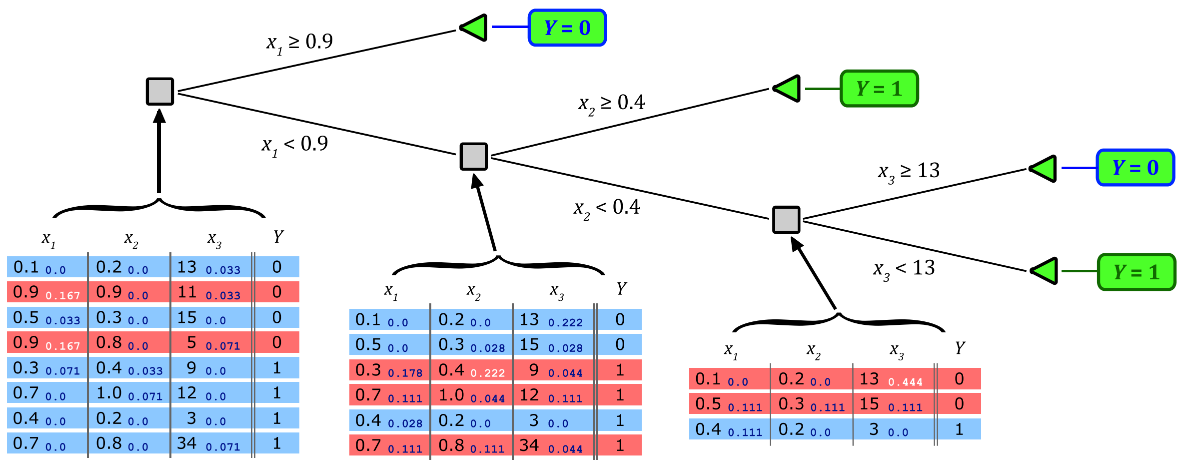


Figure 3.3: Shows the process of building a *Classification Tree* using the Gini criterion. The decision nodes in the tree are given as gray boxes, and the leaf nodes as green triangles. The rounded rectangles show the outcome label Y at each leaf node. Each table pointing to a decision node gives the data in question at that node. The left-most table, pointing to the root node, shows the complete set of 8 training observations. Apart from the values x_i, the information gain $I(t)$ using Gini "impurity" is also given as sub-scripted dark blue values. These values are computed by splitting data $x_i < T_i$ and $x_i \geq T_i$. The $I(t)$ selected for splitting the data is given in white (blue rows for $< T_i$ and red rows for $\geq T_i$).

C4.5

C4.5 is one of the techniques which fits well for classification tasks. Suppose the node t is split into sub-nodes $t_1, t_2, \ldots, t_k$. Given the entropy "impurity" $i(t)$, we define the *gain* of a split as before: $I(t) = i(t) - \sum_q \frac{N(t_q)}{N(t)} i(t_q)$. C4.5 selects the split which yields the highest *gain ratio*:

$$\frac{N(t)I(t)}{\sum_i N(t_i)(\log N(t_i) - \log N(t))}$$

The C4.5 method grows a large tree, and then prunes it back using a conservative estimate of the error at each node.

Classification trees have found wide-spread use for inferring complex `AND`/`OR` relationships amongst features, and their ability to work with different data-types (categorical, real-valued) in a unified framework. Their power to handle missing data through *surrogate splits*, and finding variable importance of the data features by order of splitting have made classification trees a popular choice in both machine learning and data mining community.

3.1.2 Random Forests

Random Forest [9] performs classification (or regression) by growing many random decision trees. These random trees are grown using the CART method [11] with the Gini impurity (explained in Section 3.1.1).

Once the forest is built, classification is done by trickling down data from each random tree. Eventually each tree votes for the class it thinks is best fit for the data provided, and the votes from all the trees are combined to give the final classification. The method ensures that each classification tree in the forest has a different structure and split tests, since correlation between any two trees would increase error rate. The collective consensus of these randomly perturbed trees is what makes Random Forests a powerful classification technique.

The term "randomized" refers to the training algorithm of random forests in two ways. Firstly, each tree is trained on a random subset of the data. Secondly, when building the tree, several candidate split tests are chosen at random from the pool of potential features/variables, and the test that optimally splits the data (under an optimization criterion, as discussed in Section 3.1.1) is chosen. These two forms of randomization help generalization by ensuring that no two trees in the forest can overfit the whole training set.

As we discussed in Section 3.1.1, *pruning* plays an important role in a classification tree's generalization ability. The complexity of the tree model has to be just right (see Figure 3.2): too little or too much training results in poor generalization. Interestingly, Random Forests have no need for pruning. All trees are built exhaustively, while dealing with its detrimental effects through *random features* and *bagging*.

Random Features

Unlike classification trees, each node in the tree makes its decision based on a random subset of the M features. On each node $m \ll M$ random features are offered to compute the node impurity (see Figure 3.4. Breiman [9] mentions that using random features not only reduces computation time, but also minimizes the correlation between trees while maintaining their strength. Choosing the right m is critical to the performance of the forest. Reducing m increases both the strength and correlation of trees. Finding the optimum m is question of analyzing the OOB (out-of-bag) error rate.

Bagging

Another difference in building random forests from typical classification trees is the selection of training data. Given the training data $\mathcal{T}$ of N training points, each tree samples $N_i \leq N$ data points with replacement (*bagging*), which we will denote as $\mathcal{T}_k$. When $N_i = N$, $\mathcal{T}_k$ is known as a *bootstrap* sample. Unlike classification trees which are concerned with the construction of $g(\mathbf{x}, \mathcal{T})$, trees in a random forest use *bootstrap* samples to create a sequence of predictors $\{g(\mathbf{x}, \mathcal{T}_k), k = 1, \ldots, L\}$, where L is the number of trees in the forest. If each tree $g(\mathbf{x}, \mathcal{T}_k)$ predicts a class $Y_j \in \{1, \ldots, J\}$, C_j will be the number of trees that vote for class Y_j. Now the final prediction of the random forest would be the class with the maximum votes $\arg\max_j C_j$. This makes random forest a *bagging predictor* [8]. Once we have the final vote, we can also derive the posterior probability of the winning class as the ratio of votes C_j/L. Since we are dealing with a binary predictor ($J = 2$), the posterior probability for each pixel would indicate how likely it is an occluded pixel.

There are several reasons for working with a *bagging predictor* rather than using the training data directly. This technique of selecting input data helps in increasing both the stability of trees and classification accuracy while avoiding overfitting. The other advantage of *bagging* is that it allows to compute a running estimate of the generalization error, as well as estimates for the strength and correlation. These estimates are done OOB (out-of-bag) which is explained next.

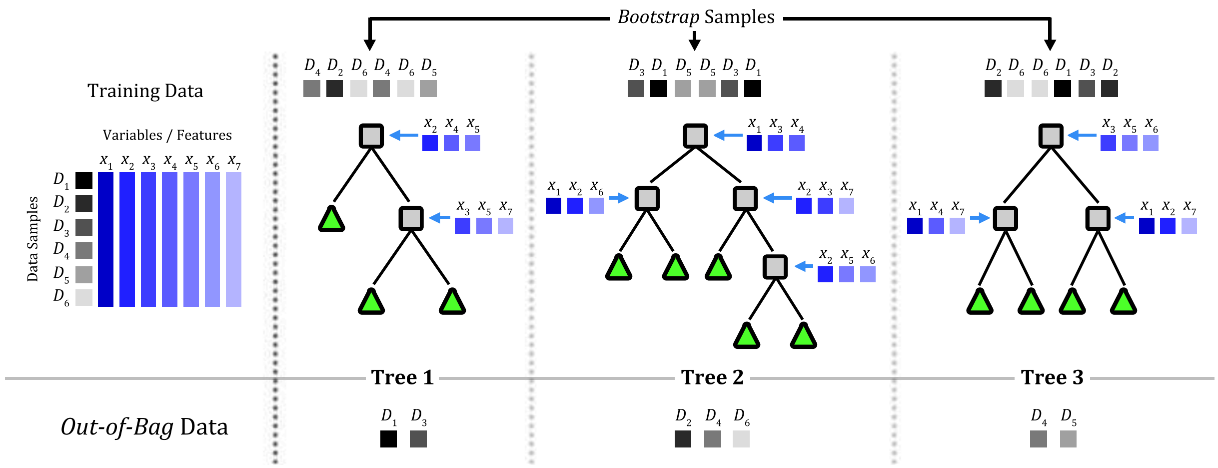


Figure 3.4: Example of a Random Forest training phase. The table on the left shows 6 training data points each with 7 variables/features. 3 trees from the forest are shown. Each tree has its own bootstrap sample $\mathcal{T}_k$ which is used for building the tree. The remaining training samples $\hat{\mathcal{T}}_k$, given in the bottom row, are used as the *out-of-bag* data. Also, each node computes node impurity on only $m = 3$ random features.

Out-of-Bag Error Rate

In a random forest there is no need for separate test data to get an unbiased estimate of the test error. When each tree takes a *bootstrap* sample, (for large datasets) around 1/3 of the data is not sampled by chance. This remaining 1/3 population is the *Out-of-Bag* data. Mathematically, since we created L classifiers of the form $g(\mathbf{x}, \mathcal{T}_k)$, each classifier would have some training set $\hat{\mathcal{T}}_k$ which it has never seen:

$$\hat{\mathcal{T}}_k = \{(\mathbf{x}_i, Y_i) : \forall(\mathbf{x}_i, Y_i) \in \mathcal{T}, (\mathbf{x}_i, Y_i) \notin \mathcal{T}_k\}$$

Hence, for any $(\mathbf{x}_i, Y_i)$ there will be around $L/3$ trees which have not trained on this sample. The set of $L/3$ trees for each $(\mathbf{x}_i, Y_i)$ is known as the *out-of-bag* classifier. We can use this *out-of-bag* classifier to get an estimate for the generalization error on the training set i.e. each $g(\mathbf{x}, \mathcal{T}_k)$ can be tested using the training set $\hat{\mathcal{T}}_k$ to get an error estimate. This estimate is known as the *Out-of-Bag Error Rate*

Strength and correlation can also be estimated using the *out-of-bag* classifier. These measures help in finding the classification accuracy and if improvements can be made. See [9] for details. The *out-of-bag* estimate is also used to get the relative importance of variables.

Variable Importance

The idea behind finding variable importance is to randomly perturb the variables in the *out-of-bag* data and find the change in classification accuracy. Given there are M input variables, their relative importance needs to be computed in turn. In each iteration we take $\hat{\mathcal{T}}_k$; randomly permute the locations of only one variable m; and test this new perturbed $\ddot{\mathcal{T}}_{k,m}$ down the respective tree. The prediction given by the random forest with $\ddot{\mathcal{T}}_{k,m}$ can be compared with the true class label to give the misclassification rate. Given the increase in the misclassification rate over all the trees, the relative importance of each variable m can be computed.

Proximities

Proximity is a measure of how similar two data points are. In decision trees, seeing if two samples end up at the same leaf node is a convenient way of judging *proximity*. To get a value for *proximity* in random forests, two data points are tested on all trees while noting how often they end up at the same leaf node. Normalizing this by L, the number of trees in the forest, gives a measure of similarity of the two samples. This measure is used for finding outliers (data point not similar to any other) and groups of points that can be clustered.

Replacing Missing Data

In training sometimes some values can be missing in the data. For instance variable m of a data point belonging to class Y_j is amiss, it can be replaced by the median value of variable m from all data records labeled as class Y_j. In case the variable is categorical, this replacement is done by selecting the mode value. In random forests, this method can be enhanced by using *proximity*. After having filled missing data with initial estimates, an iteration of random forests is run. The missing value is updated using values of variable m, having the same class label, weighted by proximity. Good estimates can be derived for the missing values with a few iterations of random forests [10].

3.1.3 Implementation

We have used the `C/C++` implementation of random forests packaged with `OpenCV 2.1`. The middle-ware layer to interact with random forests API was written in `C++`. This code has been adapted from Mac Aodha et al. [29] [2]. The binary executable of the middle-ware layer is directly called from `MATLAB` using the system() command.

3.1.4 Learning Framework Alternatives

When the data has lots of features which interact in complicated, nonlinear ways, assembling a single global model can be very difficult, and hopelessly confusing when you do succeed. Random forests aside, there are many approaches to perform such nonlinear classification. Most of them partition the space into smaller regions, where the interactions are more manageable. Below we will discuss two alternatives to random forests for our problem.

Support Vector Machines

Support Vector Machines (SVM) is a supervised learning method for binary classification [17]. Since we are dealing with a two-way classification problem, this technique is one of the popular alternatives (SVMs have also been extended to n-ary classification). SVMs are known to be better performers than random forests when training samples are limited [7]. To deal with the non-linear relationship between the input variables, SVMs map the training/testing data to a high dimensional feature space using some non-linear mapping chosen a priori. The aim is to find an optimal *hyperplane* which not only separates the classes in training, but also generalizes well. To achieve this, the optimal *hyperplane* is defined as the linear decision function with the maximal margin between the data samples of the two classes; which conveniently only requires a small amount of training to find.

[2] The `C++` code for interacting with `OpenCV` random forests is given in Appendix .1

AdaBoost

AdaBoost, in essence, is a binary predictor belonging to the family of *boosting* algorithms those try to build a strong classifier out of many weak ones [18]. Quite often the weak classifiers are built using decision trees - usually curtailed to a few levels. AdaBoost is an iterative approach, where the weight of each classifier and the distribution over the input data is iteratively adjusted. The distribution over the data-set indicates which samples need more attention from the classifiers - hence, whenever a sample is incorrectly classified, its cost is increased. This distribution is initialized by the cost of misclassifying each individual data point. On the other hand, the weight of each classifier decides what would be its contribution to the final decision. This weight typically depends on how well the classifier does on high ranking data samples. The key motivation behind AdaBoost is that it evolves the concentration of multiple weak classifiers in an attempt to improve results on hard cases instead of cases which are easily classified.

3.2 Feature Set

Given a set of features $\mathbf{x} = \{f_1, f_2, \ldots, f_M\}$, accompanied with a training (labeled) set, we now know how to train and test a random forest classifier. The aim of this section is to construct a set of features $\{f_1, f_2, \ldots, f_M\}$ which are correlated with occlusion regions. Once we have our set of features, in Chapter 4 we evaluate using them the discussed classification framework.

In this section, we describe two types of features: (1) features based on image properties (Section 3.2.1); (2) features using flow computed between the images of the sequence (Section 3.2.2). In both sections we explain the merits and drawbacks of using the respective feature to identify occlusions. To view accuracy of each feature see Chapter 4.

Throughout the thesis we will be concerned with two frame sequences - we refer to them as I_1 and I_2. We also compute some features using an image pyramid. Here, we use the notation $I_{1,z}$ to denote an image at pyramid level z.

3.2.1 Features on Image Properties

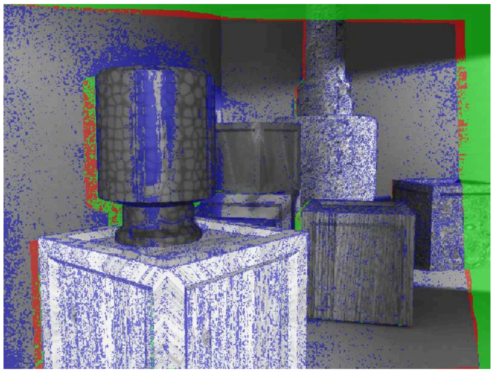
Edge Distance

All occlusion regions have to lie adjacent to surface boundaries/edges. This motivates us to include a feature based on edges, where higher distances from a true edge makes occlusion of pixels less likely. It is worth noting that using edges directly as a feature will not work since all edge detectors only mark pixels *on* a boundary - ideally all other pixels, even the ones close-by, are marked as 0. Mac Aodha et al. [29] suggests using edge distance on I_1 - the distance transform of an edge-detector's output:

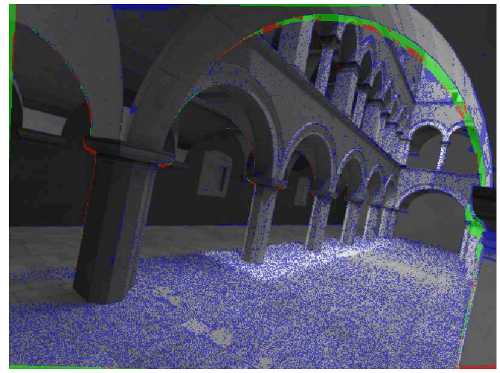
$$f_{\mathrm{ED}}(x, y, z) = \mathrm{distTrans}(\|\nabla I_{1,z}\| > T)$$

where T relates to the thresholding method the edge-detector uses, and the z indicates the level in the image pyramid ($I_{1,z}$ is I_1 at pyramid level z). We use canny-edge detector throughout this thesis (see Figure 3.8c), where hysteresis thresholds are chosen automatically. In comparison, we also experimented with *Pb. edge classifier*, which we discuss in Section 3.2.3.

The `MATLAB` class `EdgeDistFeature`, used to compute this feature, is given in Appendix .1.



(a) Photo Constancy on 4.1c



(b) Photo Constancy on 4.1i

Figure 3.5: The two images show Photo Constancy output overlayed on the respective images. Regions marked in green are true positives; regions in red are false negatives; and regions in blue are false positives. Notice the false positives both in regions of significant texture (crate and pillar in 3.5a; brick ground in 3.5b) and no texture (wall in 3.5a).

Photo Constancy

Considering that flow at an occluded pixel should be invalid, a good indicator for detecting occlusion regions could be the photoconsistency residual - the absolute difference in pixel intensities of $I_{1,z}$ against the advected location in $I_{2,z}$:

$$f_{\mathrm{PC}}(x, y, z, A) = |I_{1,z}(x, y) - \text{bicubic}\left(I_{2,z}(x + u(x, y, A), y + v(x, y, A))\right)|$$

where $u(x, y, A)$ and $v(x, y, A)$ are flow vectors at a given pixel from the candidate flow algorithm A. z is the pyramid level. Photo Constancy is one of the high importance indicators (as given by *random forest*) in our feature set. Even though it is quite powerful in distinguishing occlusion regions, it has some drawbacks. It is prone to false positives in regions of significant texture because even small errors in flow can incur large changes in pixel intensities. Photo Constancy is also inclined to make errors wherever flow breaks down even in low texture areas. This is true when regions are under an illumination gradient, and the sequence undergoes a drastic change in the FOV (see Figure 3.5 for examples). Nonetheless, like all features based on flow, Photo Constancy is ideal for finding occlusions due to change of FOV (regions that go out of frame).

The `MATLAB` class `PhotoConstancyFeature`, used to compute this feature, is given in Appendix .1.

Sparse set of Texture Features

To compute image portion similarities, it is important to have a feature which takes texture into account. Texture, here, can be defined as the repetition of basic image elements, the so-called *Textons*. Note that we are seeking a texture feature which might not be suited for texture regeneration, but performs well in texture discrimination.

Assuming that texture can be represented as a linear combination of some basis functions, one can measure how much each basis function contributes to the image. Gabor filter bank is one such form of basis functions, and Gabor energy derived on-top of it is widely used for texture analysis [22]. Although this

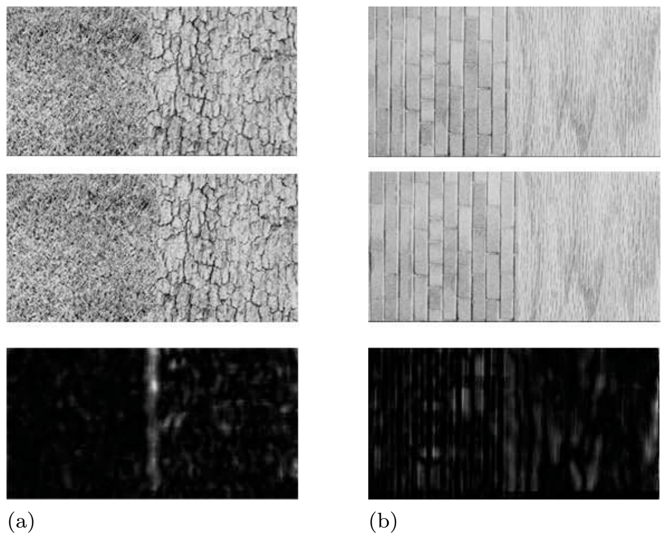


Figure 3.6: Each panel shows results of texture dissimilarity on two adjacent textures shifted by 30 pixels, using f^n_{ST} based on *A Sparse Set of Texture Features* [13]. The textures have been taken from Brodatz [12]. A window size of $n = 41$ pixels is used for computing a texture patch. Both 3.6a and 3.6b show a texture image with its shifted version. The bottom row image shows the texture comparison result as explained in the text. Note the 30 pixels wide high texture high dissimilarity in both cases - but a higher amount of noise in 3.6b due to mismatches from long vertical texture streaks.

technique for texture analysis is quite popular, Brox [13] argues that Gabor energy extracts the magnitude, orientation and scale of local texture - texture properties which are hidden in highly redundant texture. Gabor energy might hold the right amount of information for texture regeneration, but can be condensed for texture discrimination. In short, the advantage of discriminative texture models is that they lead to a low-dimensional feature space.

In this thesis we use a discriminative texture model, *A Sparse Set of Texture Features* [13] for finding how similar two regions of texture are. They are built using *structure tensor* (second moment matrix) which integrates information from the neighborhood using “nonlinear diffusion, in particular by TV flow”. This construction is more effective in preserving discontinuities than structure tensors which use Gaussian convolution. The feature vector output by this method is:

$$\mathbf{T} := \left(J_{11}, J_{22}, J_{12}, \frac{1}{\bar{m}}, P_I\right)$$

where (J_{11}, J_{22}, J_{12}) are structure tensor components which have undergone a nonlinear coupled isotropic matrix valued diffusion to give texture strength and orientation. $\frac{1}{\bar{m}}$ is the texture scale measure giving the average speed of change of the pixel intensity in a *Total Variational* framework. P_I is the pixel intensity.

To deal with texture comparisons, Brox [13] proposes a method to compare texture patches:

$$\Delta\mathbf{T} = \frac{1}{M}\sum_{k=1}^{M}\left(\frac{\mu(\mathbf{T}_{1,k}) - \mu(\mathbf{T}_{2,k})}{\sigma(\mathbf{T}_{1,k}) + \sigma(\mathbf{T}_{2,k})}\right)^2$$

where $\mathbf{T}_{1,k}$ and $\mathbf{T}_{2,k}$ denote feature k of the two texture patches which we need to compare; $M = 5$ is the number of texture features; μ and σ are the mean and standard-deviation of the given k feature.

Since, we would like to use the same technique for image comparison, we need to develop a sense of texture patch for each pixel. For this purpose we use a window-based approach, where each pixel's texture is the $n \times n$ neighborhood surrounding it. Using this neighborhood, both μ and σ are computed. After computing both these statistics for each pixel, we will advect them for I_1 by a candidate flow algorithm and compare these with statistics of I_2 to get a texture dissimilarity measure:

$$f^n_{\mathrm{ST}}(x,y,z,A) = \frac{1}{M}\sum_{k=1}^{M}\left(\frac{\mu_n(\mathbf{T}_{1,z,k}(x,y)) - \mathrm{bicubic}(\mu_n(\mathbf{T}_{2,z,k}(x+u(x,y,A), y+v(x,y,A))))}{\sigma_n(\mathbf{T}_{1,z,k}(x,y)) + \mathrm{bicubic}(\sigma_n(\mathbf{T}_{2,z,k}(x+u(x,y,A), y+v(x,y,A))))}\right)^2$$

where $\mu_n(\mathbf{T}_{1,z,k}(x,y))$ denotes the mean of texture feature k at pixel (x,y) of $I_{1,z}$ computed on a $n \times n$ window, and so on. $u(x,y,A)$ and $v(x,y,A)$ are flow vectors at the given pixel by the flow algorithm A, and z is the pyramid level. See Figure 3.6 for example results. The results of our experiments of changing the window size n of f_{ST} are given in Section 4.4.

We also experimented with a different statistic based on *A Sparse Set of Texture Features*. Unlike f_{ST}, here we compute a pixel-wise statistic. This method not only makes it quicker than f_{ST} but also performs better than f_{ST} in regions with little texture (since here the window is 1×1 pixel). Like before we advect $\mathbf{T}_{2,k}$ using a candidate flow algorithm. Using the advected texture we compute the Mahalanobis distance per pixel between the two texture features:

$$f_{\mathrm{ST}m}(x,y,z,A) = \sqrt{\sum_{k=1}^{M}\frac{\left(\mathbf{T}_{1,z,k}(x,y) - \mathrm{bicubic}\left(\mathbf{T}_{2,z,k}(x+u(x,y,A), y+v(x,y,A))\right)\right)^2}{\sigma^2_{z,k}}}$$

here $\sigma^2_{z,k}$ is the variance (over both $\mathbf{T}_{1,z,k}$ and $\mathbf{T}_{2,z,k}$) of feature k at pyramid level z. Although texture is not a local property, this is a valid statistic for comparing texture since the feature for each pixel is influenced by its neighbors due to TV flow. The comparative results of f_{ST} and $f_{\mathrm{ST}m}$ are discussed in Section 4.4.

A `MATLAB` implementation of *A Sparse Set of Texture Features* from `http://www.csc.kth.se/~omida/` has been used. With it, we use a `MEX C++` implementation of *Nonlinear Coupled Diffusion* borrowed from the same source. Our `MATLAB` class `SparseSetTextureFeature2` computes f^n_{ST} whereas `SparseSetTextureFeature` computes $f_{\mathrm{ST}m}$. These implementations are given in Appendix .1. Both classes interact with *A Sparse Set of Texture Features* to compute texture.

3.2.2 Features based on Optical Flow

The motivation for basing some features on dense flow algorithms is to take advantage of the fact that these methods tend to break down around regions of occlusion. Most of the features we discuss rely on detecting these inconsistencies in flow, both spatially and temporally. In this section we discuss these features based on optical flow. Before we do so, we describe briefly the flow algorithms we employ.

Note that both features based on *Photo Constancy* (Section 3.2.1) and *Sparse set of Texture Features* (Section 3.2.1) could have been described as flow based approaches - but they are categorized to Section 3.2.1 since they explicitly use some image properties.

Optical Flow methods used

Our first candidate flow algorithm was proposed by Horn and Schunck [24] which is a differential technique to compute flow. By taylor expansion of the brightness constancy assumption, they suggest computing the speed and direction of the pixel using partial derivatives of the image brightness with respect to x, y and t. Moreover, they argue that the computation of flow locally on a pixel is an under-constrained problem (the *aperture* problem). Since flow should change smoothly in most regions, additional constraints are imposed by having smoothness of flow in a neighborhood.

The assumption of having similar flow in a neighborhood breaks down at depth or motion discontinuities and at transparencies. Building on top of [24], our second algorithm, suggested by Black and Anandan [5] relaxes the requirement of constancy of flow in a neighborhood. They suggest estimation of flow which aims to compute sharp motion discontinuities. Using *robust statistics* which treats motion discontinuities as "outliers" in a statistical framework. The method focuses on the recovery of multiple parametric motion models within a region, as well as the recovery of piecewise smooth flow fields.

In our set of flow algorithms, we use two methods based on total variation. The first one, proposed by Wedel et al. [58], gives improvements on top of the original TV$-L^1$ optical flow algorithm. They decompose images into structure and texture to reduce the effects due to illumination changes. They also suggest using a median filter to flow fields to increase the robustness of the method. Our second total variation method, Huber$-L^1$ formulated by [59], uses anisotropic regularization in order to conform well with the underlying image structure. In an effort aimed toward video restoration, they drop the assumption of gradual flow changes over time, in lieu of a symmetry constraint with respect to the central frame in the sequence.

In order to deal with large motion, we add the method suggested by Brox and Malik [14] to our arsenal of flow algorithms. Rather than employing a coarse-to-fine technique to recover motion, they use local descriptor matching to correspond regions with large flow. This is done in conjunction with a variational framework to avoid the problem of outliers.

Sun et al. [55], Baker et al. [1] argue that the basic formulation of flow has not changed much since Horn and Schunck [24]. They suggest that all flow algorithms develop an objective function which combines a data constancy term, to maintain constancy of some image property; with a spatial term, to model how the flow should change spatially. This objective function is then optimized in a computationally tractable way. The "Classic+NL" flow algorithm, proposed by Sun et al. [55], suggests applying median filter to intermediate flow values during incremental estimation of flow to improve the objective function. They also suggest incorporating image structure using a spatially weighted term to avoid smoothing over image boundaries.

The implementations used for all these algorithms have been provided by their respective authors. The `MATLAB` classes those compute these features are `HornSchunckOF`, `BlackAnandanOF`, `TVL1OF`, `HuberL1OF`, `LargeDisplacementOF`, and `ClassicNLOF` are given in Appendix .1.

In our experiments we observed Classic+NL and Huber$-L^1$ to be the best performing flow algorithms for detecting occlusion using our features.

Temporal Gradient

One method of finding where flow algorithms perform badly, as suggested by Mac Aodha et al. [29], is to take derivatives of flow fields. Such temporal gradient is a good indicator for motion boundaries, which should make it a reasonable feature to classify occluded pixels. It is computed as:

$$f_{\mathrm{TG},x}(x, y, z) = \|\nabla \bar{u}\|$$

$$f_{\mathrm{TG},y}(x, y, z) = \|\nabla \bar{v}\|$$

Here $\bar{u}$ and $\bar{v}$ is the median flow for our candidate flow algorithms, and z is the pyramid level. This feature is not amongst the best performers in our set, partly because motion boundaries do not always co-occur with occlusions in a scene.

The `MATLAB` implementation of this feature is present in the class `TemporalGradFeature`. This is given in Appendix .1.

Flow vector features

The following set of features hypothesize that flow in occlusion regions should be noisy. This is a reasonable assumption because: (1) occlusions lie close to motion boundaries where flow looks like a random perturbation of the actual flow; (2) flow assigned to regions of occlusion is usually invalid, and is incorrectly regularized.

To check this noise in flow, we analyze the variance in the direction of flow vectors. This **Angle Variance** is computed in a small $n \times n$ square window surrounding each pixel (see Section 4.4 for a discussion on the window size). To make the notation simpler we denote the half window size as $\mathrm{w} = (n-1)/2$:

$$\theta(x, y, z, A) = \arctan\left[\, v(x, y, z, A) \;/\; u(x, y, z, A) \,\right]$$

$$\theta^n_\mu(x, y, z, A) = \frac{\sum_{c=-\mathrm{w}}^{\mathrm{w}} \sum_{r=-\mathrm{w}}^{\mathrm{w}} \theta(x+c,\, y+r,\, z,\, A)}{n^2}$$

$$f^n_{\mathrm{AV}}(x, y, z, A) = \frac{\sum_{c=-\mathrm{w}}^{\mathrm{w}} \sum_{r=-\mathrm{w}}^{\mathrm{w}} \left(\theta(x+c,\, y+r,\, z,\, A) - \theta^n_\mu(x, y, z, A)\right)^2}{n^2}$$

as before, A is the candidate flow algorithm, and z is the pyramid level of the flow on which the feature is computed. Similarly, we compute the variance of the length of optical flow vectors in an $n \times n$ window. This **Length Variance** is computed as:

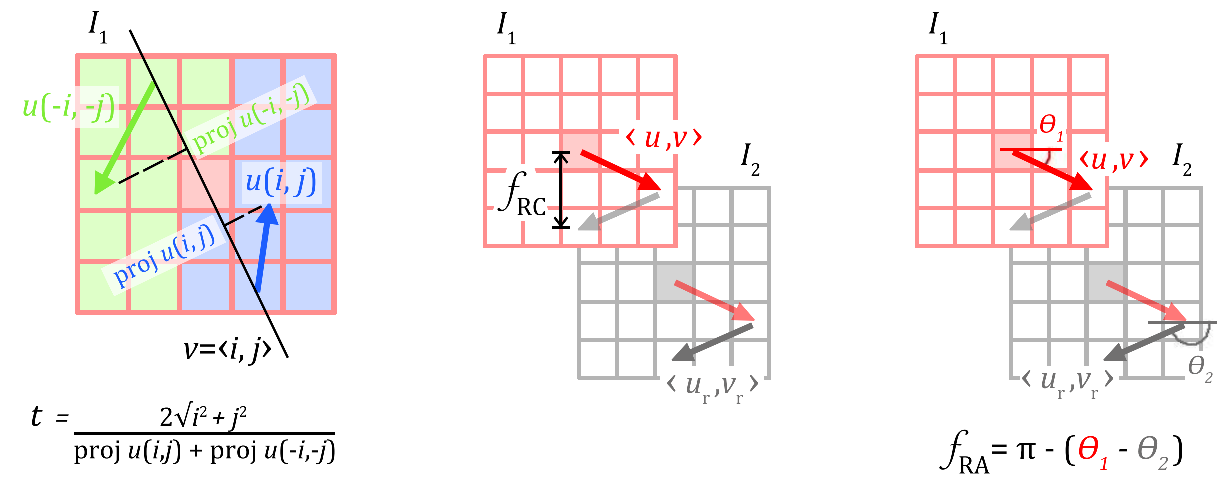
$$L(x, y, z, A) = \sqrt{u(x, y, z, A)^2 + v(x, y, z, A)^2}$$

$$L^n_\mu(x, y, z, A) = \frac{\sum_{c=-\mathrm{w}}^{\mathrm{w}} \sum_{r=-\mathrm{w}}^{\mathrm{w}} L(x+c,\, y+r,\, z,\, A)}{n^2}$$

$$f^n_{\mathrm{LV}}(x, y, z, A) = \frac{\sum_{c=-\mathrm{w}}^{\mathrm{w}} \sum_{r=-\mathrm{w}}^{\mathrm{w}} \left(L(x+c,\, y+r,\, z,\, A) - L^n_\mu(x, y, z, A)\right)^2}{n^2}$$

Surprisingly, *Length Variance* performs much better than *Angle Variance* in our tests. This indicates that flow algorithms used in our framework regularize the direction of flow more than the length at occlusion pixels.

Another method of finding occlusion regions is to see if flow in a neighborhood indicates directions which can lead to an overlap of pixels. If we know the width of the occlusion region, we can compute such a feature by looking at flow at pixels beyond that width - which should be headed for collision. Note that



(a) f^n_{CS} (b) f_{RC} (c) f_{RA}

Figure 3.7: Illustrates the idea behind the features f^n_{CS}, f_{RC} and f^n_{RA}. Figure 3.7a shows the computation of *time* t for computing the three features of f^n_{CS}. This is done by selecting a pixel from the green region and the diagonally opposite pixel in the blue region; projecting the flow onto the diagonal; and computing the *time*. Figure 3.7b shows the computation of f_{RC} as the distance after loopback flow. Figure 3.7c illustrates f_{RA} as the angle difference between the two corresponding flow vectors in I_1 and I_2

for this method to work, accuracy of flow is needed on non-occluded pixels and not at occlusion regions themselves. We formulate this **Colliding Speed** feature over an $n \times n$ pixels neighborhood in which we observe the "time" it would take for pairs of diagonally opposite pixels to collide. These pixel pairs are centered around the current pixel under consideration for occlusion. This *time* metric can be computed because we know the distance between the pair of pixels and the speed at which they are approaching each other. We consider the *time* before collision as a metric for measuring chances of occlusion, since it quantifies the urgency for occlusion.

If we consider the pixel (x, y), we can get two diagonally opposite pixels $(x-i,\ y-j)$ and $(x+i,\ y+j)$, where the distance between them is $2\sqrt{i^2+j^2}$. To compute the speed of approach, we first need to project the flow vectors on the line/vector connecting the two pixels. This direction on which the projection needs to happen is $\vec{v} = \langle i, j\rangle$. Now, the *time* before collision is computed by dividing the distance with the *projected* speed:

$$\vec{u}(x,y,i,j,z,A) = \langle u(x+i,\ y+j,\ z,\ A),\ v(x+i,\ y+j,\ z,\ A)\rangle$$
$$t(x,y,i,j,z,A) = \frac{2\sqrt{i^2+j^2}}{\text{proj}_{\vec{v}}(\vec{u}(x,y,i,j,z,A)) + \text{proj}_{\vec{v}}(\vec{u}(x,y,-i,-j,z,A))}$$

For an $n \times n$ neighborhood, this static can be computed for $(n^2-1)/2$ pixel pairs. But, not all of these pairs hold essential information for deciding the occlusion of a pixel. Our tests show that condensing this feature to three statistics is reasonable: the maximum *maximum time*, *minimum time* and *time variance*:

$$N = \{ \{(0,0),(0,1),\dots,(1,-w),\dots,(1,w),\dots,(\mathrm{w},\mathrm{w})\} \setminus (0,T) : T \leq 0 \}$$

$$f^n_{\mathrm{CS,max}}(x,y,z,A) = \inf \{ \, t(x,y,i,j,z,A) : \; (i,j) \; \in \; N \}$$

$$f^n_{\mathrm{CS,min}}(x,y,z,A) = -\inf \{ \, -t(x,y,i,j,z,A) : \; (i,j) \; \in \; N \, \}$$

$$f^n_{\mathrm{CS},\sigma^2}(x,y,z,A) = \mathrm{E}\left[t(x,y,i,j,z,A)^2\right] \quad \text{where} \quad (i,j) \; \in \; N$$

Note that $f^n_{\mathrm{CS,max}}$, at best, would be inversely proportional to the propensity of occlusion at a pixel. As expected, our tests show that $f^n_{\mathrm{CS,min}}$ is the most important among these 3 statistics (see Figure 4.10).

Our `MATLAB` implementations of the features mentioned are given in classes `OFAngleVarianceFeature`, `OFLengthVarianceFeature`, and `OFCollidingSpeedFeature`. These are given in Appendix .1.

Reverse flow features

Since we can compute flow from not only I_1 to I_2, but also in reverse, some features can be naturally developed using these dual flow vectors. In this section we will refer to $u(\cdot)$ and $v(\cdot)$ as flow from I_1 to I_2, and $u_r(\cdot)$ and $v_r(\cdot)$ as the *reverse flow* in the direction I_2 to I_1.

The first obvious feature to use is to find the difference in locations when going forward by the flow $\langle u, v\rangle$ to reach the position (x', y') in I_2; and then backward by flow $\langle u_r, v_r\rangle$ to reach a pixel in I_1. If flow is perfect in both directions, we should arrive back at the pixel we started from in I_1. Since we expect flow on occluded pixels to be invalid, this loop-back path using forward and reverse flow should result in a location which is far from the original. Hence, we compute our **Reverse Constancy** feature as the euclidean distance from the original pixel after this loopback:

$$(x', y')_{z,A} = \mathrm{round}\,(x + u(x,y,z,A),\; y + v(x,y,z,A))$$

$$f_{\mathrm{RC}}(x,y,z,A) = \|x - (x' + u_r(x',y',z,A)),\; y - (y' + v_r(x',y',z,A))\|$$

z here is the pyramid level, and A is the candidate flow algorihtm whose flow is used. Similarly, if both set of flow is perfect, the angle difference between the two should be πrad.. Our **Reverse Flow Angle Difference** feature is computed as:

$$f_{\mathrm{RA}}(x,y,z,A) = |\,\pi \; - \; \arccos\,[\; u(x,y,z,A) \; \cdot \; u_r(x',y',z,A) \;]\,|$$

The `MATLAB` classes `ReverseFlowConstancyFeature` and `ReverseFlowAngleDiffFeature` used for computing these features are given in Appendix .1.

3.2.3 Other Features experimented

The following are features we tested, but dropped from the final set of features due to the reasons stated.

Gradient Magnitude

Mac Aodha et al. [29] uses gradient magnitude of I_1 to measure textured-ness in a scene:

$$f_{\mathrm{GM}}(x,y,z) = \|\nabla I_{1,z}(x,y)\|$$

(a) Pb. posterior probability (b) Pb. *Edge Distance* (c) Canny *Edge Distance*

Figure 3.8: 3.8a shows *Probability of Boundary* (Pb.) edge classification result on image given in Figure 4.1c. Notice that each edge is given an edge-strength according to its posterior probability of being a surface boundary. Brighter edges shown here are of higher edge-strength. 3.8b shows the *Edge Distance* feature after thresholding 3.8a at 0.2. In comparison, 3.8c shows *Edge Distance* using *canny* edge detector. Notice the added noise in 3.8c, but the disappearance of edges of pillar and the right-most crate in 3.8b

where z is the pyramid level. We tried this feature to see if the correlation of gradient magnitude with edge boundaries would help us find pixels close to regions of occlusion. Although this feature does well in assigning high values to surface boundaries, the *random forest* classifier does not assign much importance to it because: (1) gradient magnitude is not only high at edges but also large on areas of significant texture; (2) the *Edge Distance* feature (see Section 3.2.1) makes the former slightly redundant.

The `MATLAB` class `GradientMagFeature`, used to compute this feature, is given in Appendix .1.

Pb. Edge Classifier

Apart from using a *canny* edge detector for *Edge Distance* feature (see Section 3.2.1), we also evaluated the *Probability of Boundary* (Pb.) edge classifier proposed by Martin et al. [33]. As compared to *canny*, which attempts to find any abrupt changes in pixels, Pb. tries to mark edges wherever pixels move from surface of one object to another. Like our occlusion detection method, Pb. edges are also computed by combining features and using them in a supervised classification framework (their ground-truth comes from a database of human-marked boundaries [31]). Martin et al. [33] propose four features to classify a pixel as a boundary. To detect brightness edges, *Oriented Energy* is computed from even and odd-symmetric filter responses at a certain orientations. The remaining three gradient features compare statistics over opposing halves of a circle along a given angle. To compute *Brightness* and *Color Gradient* features, kernel density estimates of the distributions of pixel luminance and chrominance are binned in each disc half. For *Texture Gradient*, histograms of vector quantized filter outputs are computed for each half. These four features are evolved to make features for classification using first-order approximation of the distance to the nearest maximum. These features are then combined using *logistic regression.* We denote this feature at a pixel as:

$$f^{T}_{\mathrm{PB}}(x, y, z) = \mathrm{distTrans}(Pb. > T)$$

where z is the pyramid level, and T is the threshold used on the posterior probability output by the Pb. edge classifier.

The reason for not adopting Pb. over *canny* edge classifier for *Edge Distance* feature is partially due to the speed of computing it on an image pyramid. It can take up to 2 hours to compute the pyramid on any one of our sequences (`Intel Core 2 Duo 2.5GHz`). Using Pb. without any pyramid levels was tested against 10 pyramid levels of *canny*. The results show that computing on scale is necessary for an *edge distance* feature (see Section 4.4). The second reason for not using Pb. is the misclassification of some edges in synthetic scenes. We think that this is largely due to training on only natural scenes. Although the Pb. edge classifier has found wide use, due to the lack of occlusion-marked training sets on natural scenes, we don't achieve the performance one would expect using a feature based on Pb. edge classification. Nevertheless, if a feature is successfully developed using Pb., the strength of boundaries could play some role in discriminating which are surface boundaries as opposed to texture edges - an important cue for occlusion classification.

We use a `MATLAB` implementation of Pb. edge classifier given at `http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/`. The `MATLAB` class `PbEdgeStrengthFeature`, which interacts with these implementations, is given in Appendix .1.

Chapter 4

Evaluation / Experiments

The training phase in any learning technique plays a critical role. Selecting the set of sequences providing ideal characteristics to "learn" on, while avoiding any overfit over training is the theme of this chapter. After giving the methodology behind our tests, the second Section (4.2) describes the dataset we have chosen. In Section 4.3 we look at tweaking the parameters of the Random Forests (see Section 3.1.2) to achieve optimal performance on our tests. In Section 4.4 we evaluate individual and collective features to discover their strengths and weaknesses. The remaining sections look at particular problems in our classification and proposes solutions. Section 4.5 discusses the classification on *within-FOV* pixels, and steps taken to quantitatively analyze it. In Section 4.6 we explore the role texture plays in the performance of our technique. Since the majority of our features are dependent on optical flow, we will look at training and testing our method with *Ground-Truth* flow in Section 4.7. This will illustrate the power of our technique if the input flow was close to ideal. In Chapter 5 we show results on unseen sequences.

4.1 Methodology of Evaluation

To evaluate any classification framework, we need to carefully choose our tests to identify key problems with the system. Since we use random forests our concern would be more toward what feature parameters are right to achieve optimal performance rather than what features are redundant and need to be discarded. This is because forests are trained to use features more often which closely correlate with the output label (see Section 3.1.2)

As discussed in Section 4.2, it is important to have a dataset which provides examples that would occur in the wild. The performance of the classifier on unseen data would depend on this choice. Since we have a limited sized dataset it is also important to choose a reasonable training and testing technique. To remove the need to have two separate training and testing datasets, we use K-fold cross-validation [21]. Using cross-validation, the dataset is divided into k partitions (k sequences in our case), and on each round the classifier is trained on $k-1$ partitions, and tested on the remaining partition. "K-fold" refrers to this process being repeated k times to average results. If the dataset is representative of the unseen data the classifier will be tested on, these average results are good estimates of the classifier's accuracy. Throughout this chapter we will show results on partitions (sequence) using cross-validation.

To analyze the results of one round of cross-validation, we use the receiver operating characteristic (ROC). ROC is a graph of true positives (TP) against false positives (FP), as we adjust some metric. In our case this metric is the threshold on the posterior probability of each data-point output by the random forest (based on votes on its leaves). The aim of a classifier should be to get as many true positives without incurring false positives i.e. reaching the top left point $(0, 1)$ of the ROC graph. The area under the ROC curve $A \in [0, 1]$ (AUC) is a single number which used throughout this thesis for judging the classifier's performance.

Once we have an ROC curve, we might want to choose an appropriate threshold to get a binary output. To get this "best" threshold, we select the point where the gradient of the ROC curve is:

$$\beta = \frac{N\ C_{\mathrm{FP}}}{P\ C_{\mathrm{FN}}} \tag{4.1}$$

where N is the total number of negatives, and P is the total number of positives. C_{FP} and C_{FN} are the costs of false positive and false negative respectively.

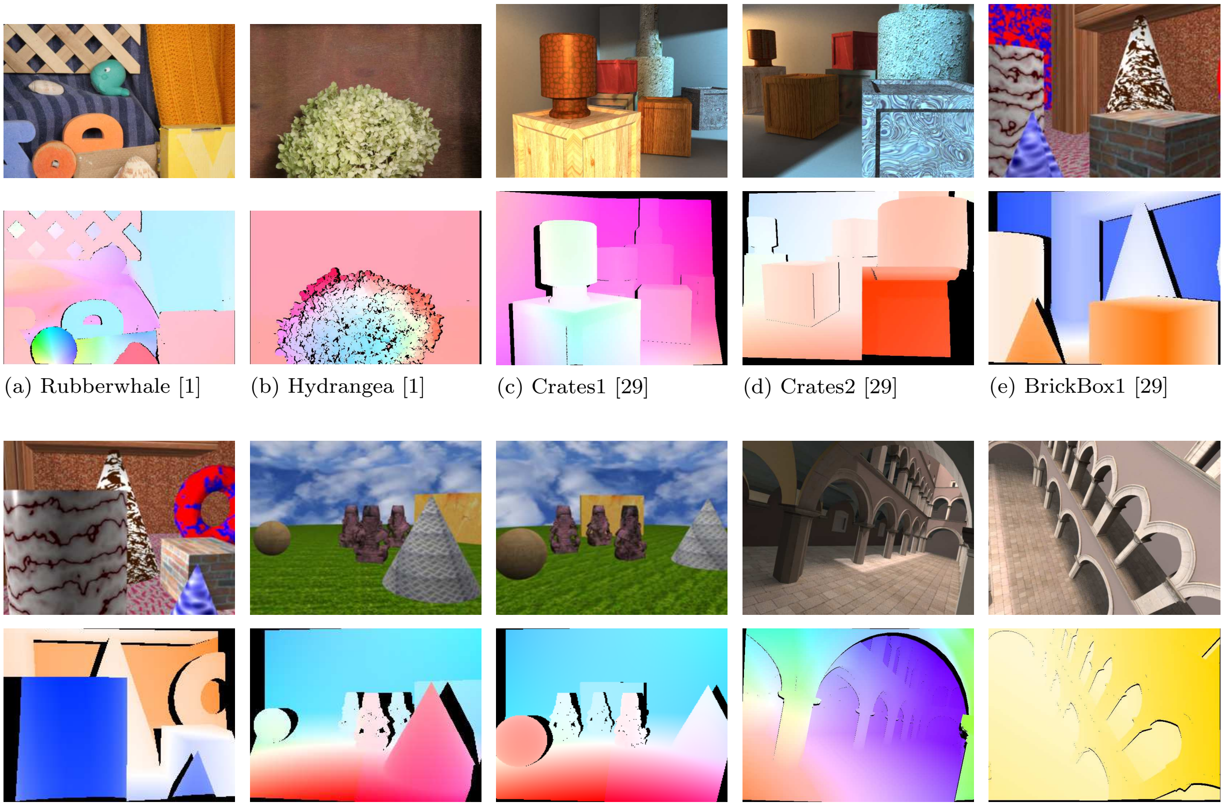
4.2 Training Dataset

Our training dataset has 10 sequences shown in Figure 4.1: 2 natural sequences from Baker et al. [1]; and 8 synthetic sequences from Mac Aodha et al. [29]. All 8 synthetic sequences are static scenes where only the camera moves. The movement is significant but cannot be categorized as wide-baseline. They were modeled and lighted using `Maya`, and the ground-truth flow (including occlusion markings) was computed using a `Maya Mel` scripts. This involves projecting the 3D motion of the scene corresponding to I_1 onto the 2D image plane. Furthermore we use a few variations of sequences given in Figure 4.1c and 4.1d to test the effect of texture on our classification. The results are given in Section 4.6.

The complete middlebury dataset [1] provides three types of sequences with GT: real imagery of nonrigidly moving scenes; synthetic imagery; and real stereo imagery adjusted for optical flow. The GT for synthetic and real stereo imagery does not have the occlusion regions marked. Even though these sequences cannot be used for training, they will be used in Section 4.7 for training and testing using exact flow.

The paucity of natural ground-truth flow sequences is understandable as collecting them is non-trivial. For years its need was felt amongst researchers working on motion, who have largely coped with rudimentary synthetic sequences. In an effort to fill this void, Baker et al. [1] provide real imagery of nonrigidly moving scenes with GT which has regions marked wherever flow is inapplicable. These scenes are built using a real computer-controlled motion stage, where the camera is stationary but the objects in the scene move in a non-rigid fashion. To compute the GT, each object in the scene is splattered with fluorescent paints closely matching the color of the surfaces. The scene is then captured under both ambient and UV lighting. Note that fluorescent paint absorbs UV light but reflects light in the visible spectrum - this allows scene capture and GT capture from the same camera. To maintain accuracy, flow is computed on high-resolution images while their low-resolution versions are distributed.

The GT flow is computed using a brute-force SSD tracker searching in a small window. The results of all flow vectors are cross-checked by tracking both forward and backward while requiring perfect correspondence. Pixels that fail cross-checking are marked as occluded. Baker et al. [1] mentions that although the chance of failure is low with cross-checking, the method is not fool-proof. Even using this robust technique, some valid visible regions get misclassified as occlusions. Moreover, since the aim of the dataset is to provide accurate GT flow, and not reliable markings of occlusions, they can argue for making flow unavailable at regions where there is even slight uncertainty. Contrary to their aims, we are concerned with the accuracy of occlusion markings, and not with the accuracy of flow. To deal with the most apparent mis-classifications in the 2 sequences used, we attempt to manually mark pixels wherever we believe that an "occlusion" was marked in GT not because of an actual occlusion, but due to the



(a) Rubberwhale [1] (b) Hydrangea [1] (c) Crates1 [29] (d) Crates2 [29] (e) BrickBox1 [29]

(f) BrickBox2 [29] (g) Mayan1 [29] (h) Mayan2 [29] (i) Sponza1 [29] (j) Sponza2 [29]

Figure 4.1: Dataset used for training the *Random Forest* classifier. Each of these is a 2 frame sequence. The first image for each sequence shows the first frame I_1 of the sequence, and the second image shows the Ground-Truth (GT) flow. 4.1a and 4.1b are the only natural sequences, taken from the middlebury flow dataset [1]. The areas marked black in the GT are regions of occlusion. The middlebury dataset marks black regions based on the reliability of tracks of those pixels - hence, some pixels amongst them are not occluded. These mistakes have been partially corrected (see Figure 4.2).

failure of the SSD tracker to cross-check flow accurately. An example of these markings is given in Figure 4.2. These pixels of uncertain occlusions are ignored while training our *random forest* classifier.

The 10 combined sequences are sufficient for training and testing since they contain a good mix of textured / non-textured; camera motion / non-rigid subject motion; and small area occlusions / wide occlusion sequences. This training set contains nearly 136k occluded pixels, and 1851k odd non-occluded pixels.

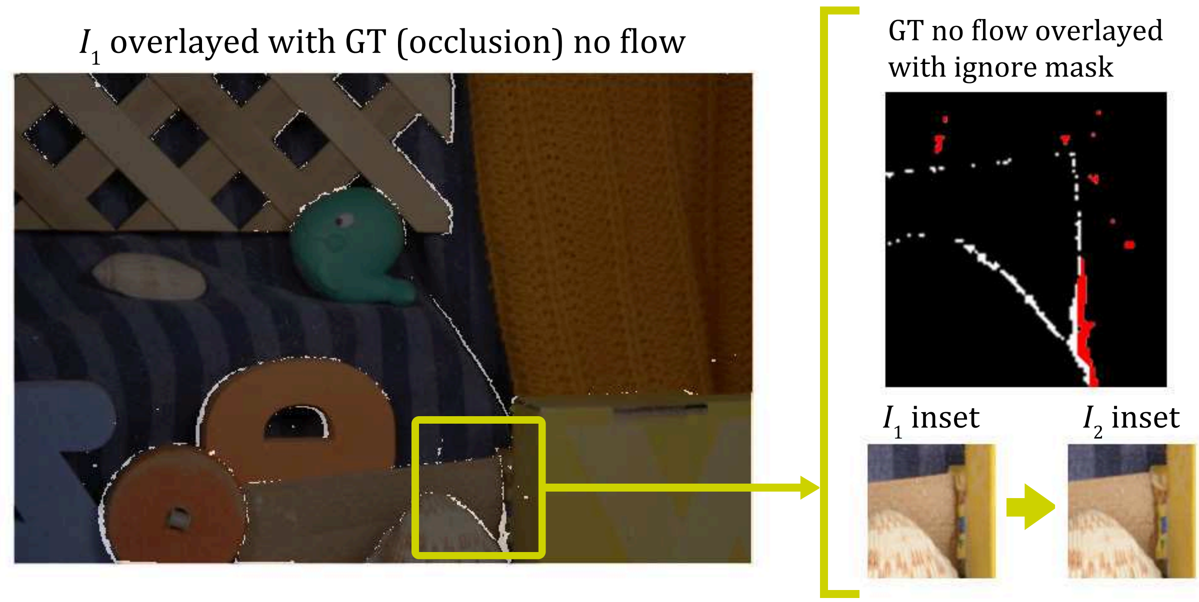


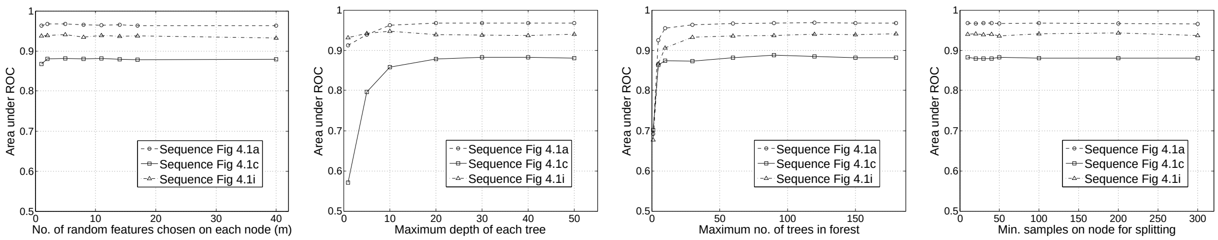
Figure 4.2: Shows some pixels from Figure 4.1a which are marked as unreliable for training the forest. The left-side shows I_1 from the sequence overlayed with the occlusion regions from the Ground-Truth (GT). The right-side shows an inset of the GT with some errors overlayed in red. The respective inset from the two image sequence is shown at the bottom. Notice, how the ceramic behind the shell has moved towards the left - making its right boundary "visible" rather than "occluded". Also note the invalid (not-occluded) spots around the ceramic boundary. Some of these errors have been manually identified and marked in red. These red regions will not be used for training the classifier.

4.3 Random Forest Evaluation

4.3.1 Random Forest Parameters

In one of our experiments we tried tweaking the parameters of random forests to evaluate the performance of the forest itself and find the ideal parameters. We considered 4 parameters which might have an effect on our results: m is the number of random features offered to each node for making node impurity decisions; d is the maximum depth allowed for each classification tree in the forest; t is the maximum number of trees the forest is allowed to grow; and c is the minimum number of samples needed on a node to allow a split. For greater detail on these parameters, refer to Section 3.1.2.

Figure 4.3 shows the area under the receiver operating characteristic (ROC) produced by individually testing the four parameters shown above. The ROC curves are generated by thresholding the posterior probability output by the random forest. If not stated otherwise in these tests, $m = 4$, $d = 30$, $t = 100$,



(a) (b) (c) (d)

Figure 4.3: Shows four cases of adjusting parameters on the *random forest*. All graphs display the area under the receiver operating characteristic (built by thresholding the output posterior) of a *random forest* by cross-validation - training on all sequences except the one it is being tested on. Note y-axis in all plots are in the range $[0.5 - 1.0]$. Figure 4.3a shows the effect of changing the number of random features (m) used for computing node impurity. Figure 4.3b shows results of changing the maximum depth a classification tree is allowed to reach. Figure 4.3c shows the maximum number of trees in a forest is allowed to grow. The last plot, Figure 4.3d, shows the effect of increasing the minimum number of samples on a node to allow a split.

and $c = 25$. The features used are:

$$\begin{aligned}\mathbf{x}_i = \{\ & f_{\mathrm{GM}}(x, y, [1-10]),\ \ f_{\mathrm{ED}}(x, y, [1-10]),\ \ f_{\mathrm{TG},x}(x, y, [1-10]),\ \ f_{\mathrm{TG},y}(x, y, [1-10]), \\ & f^3_{\mathrm{AV}}(x, y, [1-4], [1-k]),\ \ f^3_{\mathrm{LV}}(x, y, [1-4], [1-k]),\ \ f^3_{\mathrm{CS,max}}(x, y, [1-4], [1-k]), \\ & f^3_{\mathrm{CS,min}}(x, y, [1-4], [1-k]),\ \ f^3_{\mathrm{CS},\sigma^2}(x, y, [1-4], [1-k]),\ \ f_{\mathrm{PC}}(x, y, [1], [1-k])\ \}\end{aligned}$$

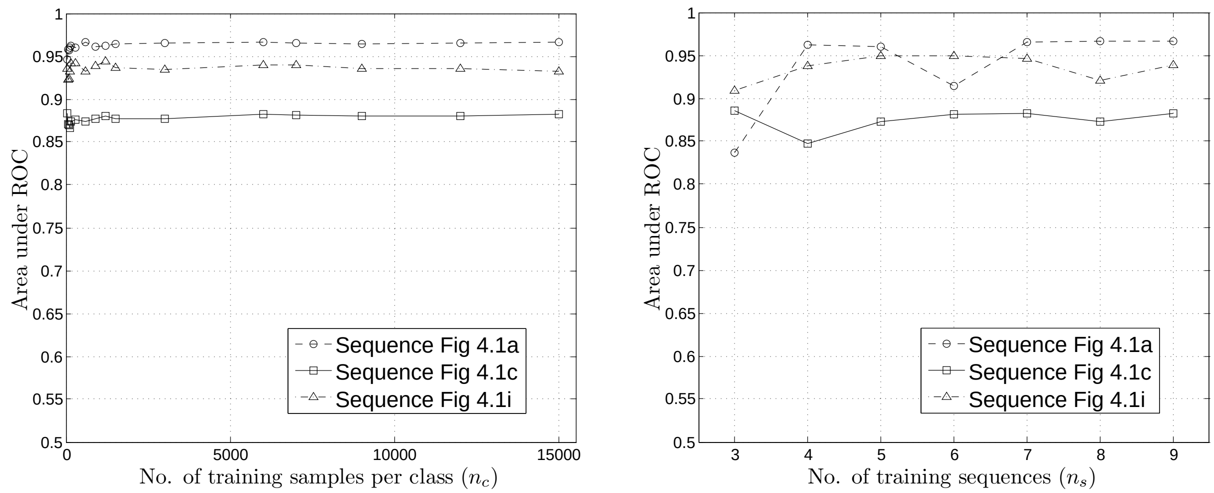
where $k = 6$ is the number of flow algorithms as explained in Section 3.2.2. For more explanation on these features, refer to Section 3.2.

We experimented with m in the range of 1 to 40. One expects that decreasing the number of random features m chosen for computing node impurity at node should also increase the misclassification rate of the forest. Observe in Figure 4.3a that decreasing m does not drastically reduce the classification accuracy. We can explain this by noting that by reducing m, the number of nodes in the forest might even increase depending on the stopping criterion for each tree. If this is so, the classification accuracy will only be slightly effected since the trees in the forest keeps splitting data until the results are within a certain accuracy. Hence for this feature length, $m \geq 5$ works well.

We also experimented with changing the maximum allowed depth d for the trees in the forest from 1 to 50. As expected, the classification accuracy drops when the trees are not allowed to extend the depth below which they were still adding nodes. Figure 4.3b indicates that at-least some trees were reaching a depth of 20. Hence for our tests we have maintained $d \geq 20$.

Another critical parameter is the maximum number of trees t allowed in the forest. We experimented with values between 1 and 180, and we observed that 25 or less trees can have severe detrimental effects on the accuracy of the forest. This effect can be attributed to the fact that forests produce a posterior probability by averaging votes on all trees - decreasing the number of trees increases the chances of collecting votes from a badly trained tree. Setting maximum trees $t \geq 50$ gives reasonable results.

The last parameter we experimented with is c, the minimum number of samples required on a node to a allow the tree to split the node. We tested a wide range of values for c between 10 and 300. As can



(a) (b)

Figure 4.4: Shows two experiments where the samples given to the *random forest* are varied (see Figure 4.3 for an explanation on ROC plots). Figure 4.4a shows the effect of changing the number of data points (n_c) sampled randomly from each sequence for each class. Figure 4.4b shows choosing n_s number of sequences randomly for training.

be seen in Figure 4.3d, changing c has little effect on the forest's performance, even though we expect it to decrease with increasing c. We can reason from the results that although the classification accuracy should decrease when large amount of samples are not allowed to split on a node, it also helps avoid situations where we over-fit on the training set. Hence using a $c \leq 150$ works well for our framework.

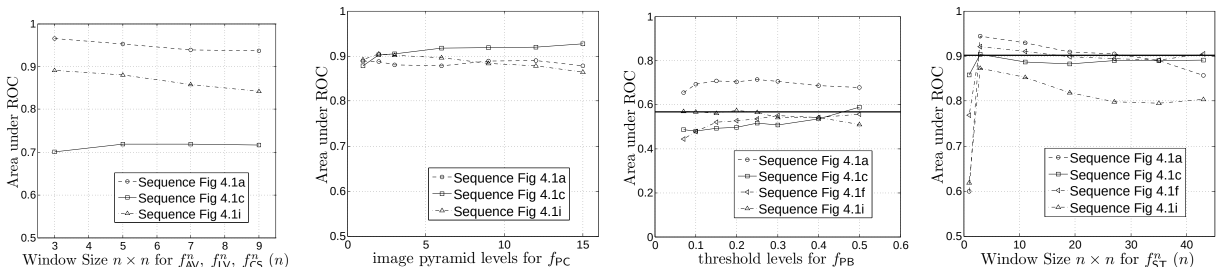
Concluding from these tests, we use $m = 11$, $d = 35$, $t = 105$, and $c = 20$ throughout this thesis for training random forests.

4.3.2 Random Forest Training Set

We also experimented with changing the number of samples used for training. Having a low amount of training samples can severely effect the performance of a supervised classification method. Nevertheless, if sampling randomly from the training set, using large amounts of data not only makes *learning* slow, but it is also redundant. Moreover, with some supervised methods, using large training sets can also lead to overfitting problems. See Section 3.1.2 to see how random forests avoids this problem while training.

We did two experiments where the number and quality of samples given for training was varied. Given that for each test we have 9 sequences available for training (after removing the sequence we are going to test on), we randomly sample n_c pixels for each class (occluded and not-occluded) from each sequence i.e. we have $9n_c$ samples for each class to train on. Our results of varying n_c from 30 to 15000 are given in Figure 4.4a. Surprisingly the forest accuracy is reasonable even when $n_c = 600$.

The more interesting case is when we keep the number of samples for each class from a sequence constant, $n_c = 7000$, and vary the number of training sequences n_s itself i.e. we train with $7000n_s$ samples for each class. Figure 4.4b shows results when we use 3 to 9 sequences for training. Since the



(a) (b) (c) (d)

Figure 4.5: Shows four different experiments for finding the right parameters for different features. All figures show plots of area under the ROC curve (see Figure 4.3 for an explanation on ROC plots). The first Figure 4.5a shows the effect of increasing the window size for f^n_{AV}, f^n_{LV}, f^n_{CS} trained in a single forest. Figure 4.5b plots the effect of varying the number of image pyramid levels for f_{PC}. The third plot, Figure 4.5c shows comparisons between f_{ED} (computed on a pyramid) against f_{PB} (not computed on a pyramid). Results for 4 sequences are shown for f_{PB}, whereas the horizontal line gives the average area under the ROC using f_{ED} on the same sequences. Note that y-axis in this plot is in the range $[0.0 - 1.0]$. Figure 4.5d plots the result of changing the window size used in f_{ST}. Results for 4 sequences are shown, and the horizontal line gives the average area under the ROC using $f_{\mathrm{ST}m}$ for the same sequences.

dataset is a mixture of synthetic and natural sequences, the classifier accuracy greatly depends on what actual sequences it got to train on. Notice how the classifier does better on sequence 4.1c than on sequence 4.1a when $n_s = 3$, but the opposite is true when $n_s = 4$. Note on each n_s a new random set of sequences is chosen.

For these tests we used the same feature set $\mathbf{x}$ used in Section 4.3.1. We trained on a random forest with the parameters which were decided in the same section.

We can conclude from these tests that although the number of samples taken from each sequence for each class n_c can be low, but it does not hurt the classifier performance when it is high (no overfitting). Hence, throughout this thesis we use $n_c = 7000$. Relating to the second test, it is best to use samples from all sequences ($n_s = 9$) for better generalization of the classifier.

4.4 Features

As discussed in Section 3.2, some features have parameters which need testing. We test these parameters by training the random forest classifier using only the feature in question. The forest is trained each time by a new variation on the parameter while everything else is kept constant.

In the first experiment we test the effect of changing the window size $n \times n$ using the feature set:

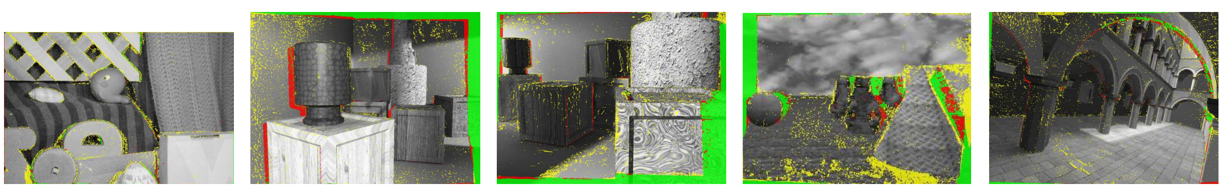
$$\begin{aligned}\mathbf{x}_i = \{\ & f^n_{\mathrm{AV}}(x, y, [1-4], [1-k]),\ \ f^n_{\mathrm{LV}}(x, y, [1-4], [1-k]),\ \ f^n_{\mathrm{CS,max}}(x, y, [1-4], [1-k]),\\ & f^n_{\mathrm{CS,min}}(x, y, [1-4], [1-k]),\ \ f^n_{\mathrm{CS},\sigma^2}(x, y, [1-4], [1-k])\ \}\end{aligned}$$

where $k = 6$ is the number of flow algorithms. Note that all these features use a neighborhood window to compute statistics (see Section 3.2.2). Our results of testing with a window size from 3×3 to 9×9 are given in Figure 4.5a. Interestingly, we get better results when the window size is small. This could be because bigger windows are prone to smoothing statistics across object boundaries. Although a significant window size is required for collecting reliable statistics, bigger windows are also prone to returning values close to motion boundaries which are reminiscent of occlusion. Deriving from these results, we conclude that a 3×3 pixels window works best for these features. We use this window size for these features throughout this thesis, unless stated otherwise.

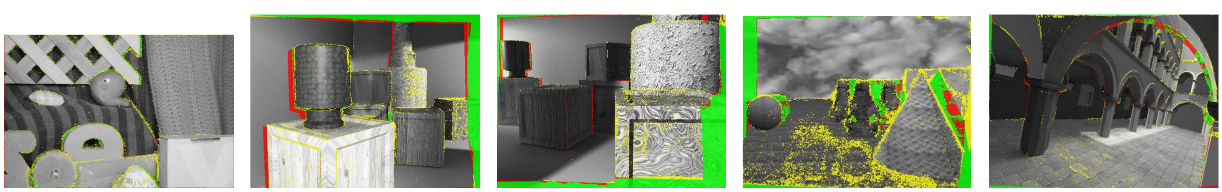
The second set of experiments is about finding the number of pyramid scales up-to which f_{PC} is effective. Photo-constancy is one of the best performing features in our set - and it was initially observed that computing it on an increasing depth of the pyramid decreased the misclassification rate. This test shows that after 4 levels, computing f_{PC} on pyramid has diminishing returns (see Figure 4.5b. This is also apparent in the variable ranking returned by the random forest. Note that the rescaling factor for the pyramid is set to 0.8.

As described in Section 3.2 we experiment with computing edge distance from the outputs given by a Pb. edge classifier and the canny edge detector. These features attempt to find occlusions occurring close to object boundaries. The Pb. edge classifier is built for the purpose of distinguishing true surface edges from strong pixel contrasts within a surface - an ideal property to have in our feature vector. Figure 4.5c shows the effect of varying the threshold T (see Section 3.2.3) applied on the posterior probability returned by Pb.. The classifier's performance f_{ED} is also given as a horizontal line. f_{ED} shows comparable performance to f_{PC}^{T} since its computed on a 10 level image pyramid. These test show that scaling is important for an edge-distance feature. We use f_{ED} in our final set features instead of f_{PB}^{T} because of the reasonable performance of *canny* in these results and other reasons discussed in Section 3.2.3.

As discussed in Section 3.2.1, we experiment with two texture statistics based on the same underlying texture feature. Since we need to compare a pixel's texture to another pixel we have the choice of using a local window based approach (f_{ST}^{n}) or a pixel-wise statistic ($f_{\text{ST}m}$). By changing the window size n of



(a) f_{ST}^{3} on seq. 4.1a (b) f_{ST}^{3} on seq. 4.1c (c) f_{ST}^{3} on seq. 4.1d (d) f_{ST}^{3} on seq. 4.1g (e) f_{ST}^{3} on seq. 4.1i



(f) $f_{\text{ST}m}$ on seq. 4.1a (g) $f_{\text{ST}m}$ on seq. 4.1c (h) $f_{\text{ST}m}$ on seq. 4.1d (i) $f_{\text{ST}m}$ on seq. 4.1g (j) $f_{\text{ST}m}$ on seq. 4.1i

Figure 4.6: Comparison of f_{ST}^{3} (**first row**) against $f_{\text{ST}m}$ (**second row**). The green overlay over the image shows true positives; the red overlay shows false negatives; and the yellow overlay shows false positives. The images were produced by thresholding the posterior output by the forest with a threshold using Equation 4.1 where $C_{\text{FP}} = 1$ and $C_{\text{FN}} = 10$. Notice how $f_{\text{ST}m}$ does better on regions with little texture as compared to f_{ST}^{3}. On the other hand, f_{ST^3} does better than $f_{\text{ST}m}$ on textured regions.

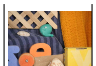
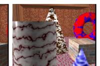
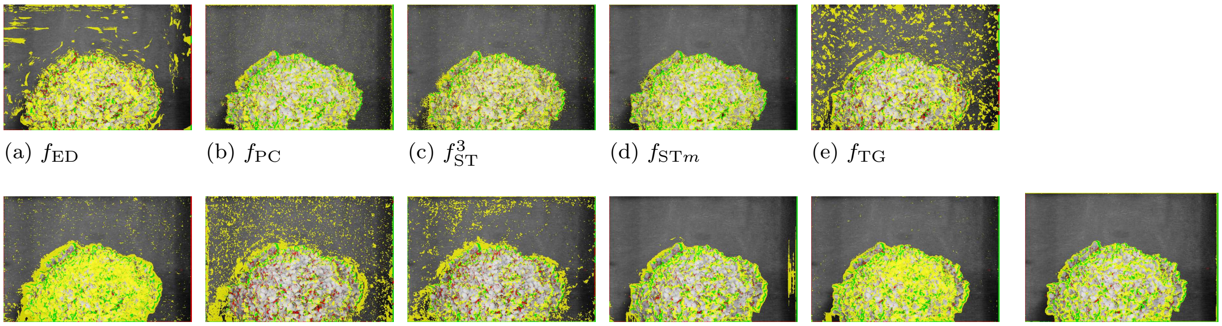
	 seq. 4.1a	 seq. 4.1b	 seq. 4.1c	 seq. 4.1d	 seq. 4.1e	 seq. 4.1f	 seq. 4.1g	 seq. 4.1h	 seq. 4.1i	 seq. 4.1j
$f_{\mathrm{ED}}(x, y, [1-10])$	0.724	0.702	0.412	0.569	0.598	0.554	0.508	0.612	0.577	0.568
$f_{\mathrm{PC}}(x, y, [1-4], [1-k])$	0.948	0.900	0.913	0.970	0.963	0.934	0.945	0.979	0.903	0.893
$f^3_{\mathrm{ST}}(x, y, [1], [1-k])$	0.945	0.845	0.904	0.958	0.927	0.906	0.921	0.956	0.873	0.868
$f_{\mathrm{ST}m}(x, y, [1], [1-k])$	0.965	0.904	0.855	0.952	0.926	0.901	0.935	0.972	0.885	0.914
$f_{\mathrm{TG}}(x, y, [1-10])$	0.744	0.737	0.540	0.625	0.747	0.612	0.637	0.627	0.520	0.498
$f^3_{\mathrm{AV}}(x, y, [1-4], [1-k])$	0.848	0.767	0.389	0.610	0.747	0.571	0.620	0.687	0.757	0.516
$f^3_{\mathrm{LV}}(x, y, [1-4], [1-k])$	0.831	0.716	0.685	0.748	0.863	0.747	0.733	0.869	0.800	0.706
$f^3_{\mathrm{CS}}(x, y, [1-4], [1-k])$	0.922	0.773	0.672	0.791	0.911	0.746	0.798	0.895	0.881	0.781
$f_{\mathrm{RC}}(x, y, [1-10], [1-k])$	0.965	0.896	0.920	0.958	0.964	0.919	0.932	0.973	0.836	0.872
$f_{\mathrm{RA}}(x, y, [1-10], [1-k])$	0.963	0.899	0.929	0.976	0.928	0.932	0.942	0.968	0.925	0.854
$f^3_{\mathrm{AV}},\ f^3_{\mathrm{LV}},\ f^3_{\mathrm{CS}}$	0.961	0.597	0.714	0.810	0.919	0.751	0.819	0.915	0.892	0.788
$f^3_{\mathrm{AV}},\ f^3_{\mathrm{LV}},\ f^3_{\mathrm{CS}},\ f_{\mathrm{PC}}$	0.979	0.908	0.924	0.976	0.964	0.956	0.936	0.984	0.943	0.942
All	0.983	0.925	0.913	0.984	0.970	0.958	0.959	0.990	0.942	0.943
$f_{\mathrm{GM}}(x, y, [1-10])$	0.706	0.696	0.491	0.583	0.584	0.567	0.551	0.627	0.567	0.522
$f^{.25}_{\mathrm{PB}}(x, y, [1])$	0.714	0.551	0.517	0.534	0.474	0.366	0.534	0.555	0.565	0.574

Table 4.1: Shows the results of training a random forest with the features in the left column. Each column shows the result on a sequence using k-fold cross-validation. Each cell gives area under the ROC curve (see Figure 4.3 for an explanation on ROC plots). Features below the thick line are not included in the final set of features. The dark gray row shows results using all the features finalized in a single random forest. The two lighter gray rows give results of training a forest using a small subset of the features.

f^n_{ST} we see how it compares to the pixel-wise feature ($f_{\mathrm{ST}m}$). Figure 4.5d shows the effect of using a very local (1×1) to a wide-area window (43×43). Figure 4.5a shows increasing the window size does not bring much benefits beyond a 3×3 window. The same reasons why big windows do not improve classification accuracy of the experiment given in the Figure 4.5a also apply here. Increasing n has the undesired effect of smoothing statistics across object boundaries. Concluding from this experiment, we always use $n = 3$ for f^n_{ST}.

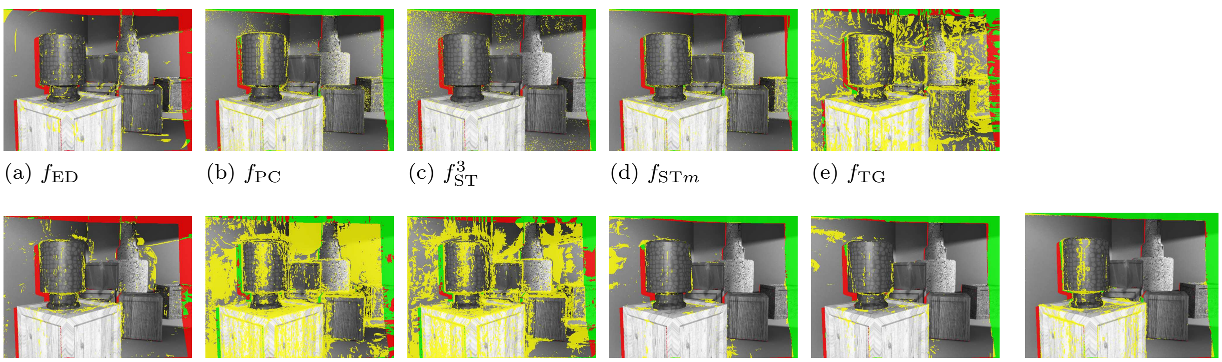
Looking at the results of a classifier trained on f^n_{ST} and $f_{\mathrm{ST}m}$ individually, it seems that both have benefits in different regions of the sequence. Understandably, $f_{\mathrm{ST}m}$ tends to do better on regions of less texture whereas f_{ST^3} performs better on regions of significant texture. Figure 4.6 shows the random forest output on 5 test sequences. Notice the performance of the classifiers on the floor in Figure 4.6e and 4.6j and the wall in Figure 4.6b and 4.6g. Due to this complementary performance, both features f^3_{ST} and $f_{\mathrm{ST}m}$ make it to the final set of features.



(a) f_{ED} (b) f_{PC} (c) f^3_{ST} (d) $f_{\mathrm{ST}m}$ (e) f_{TG}

(f) f^3_{AV} (g) f^3_{LV} (h) f^3_{CS} (i) f_{RC} (j) f_{RA} (k) All

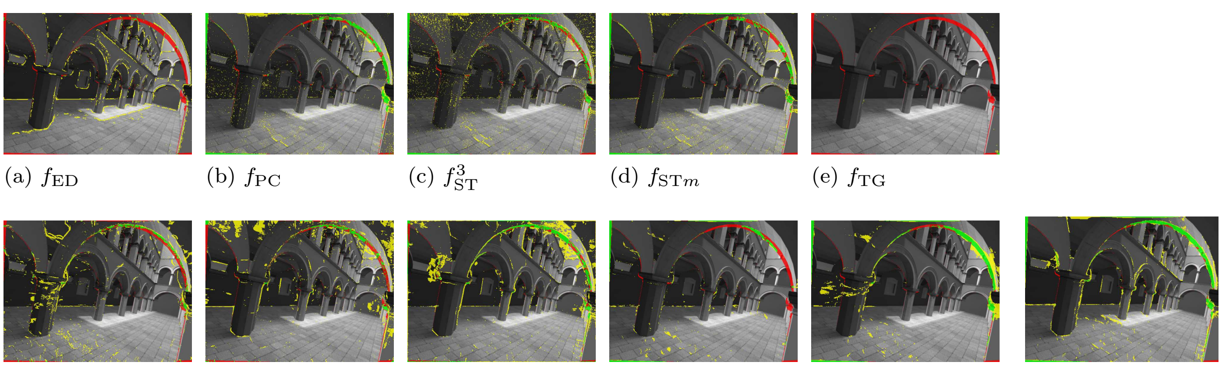
Figure 4.7: Performance of random forest on sequence 4.1b using the different set of features. Images on this page are overlayed with the posterior output using the method explained in Figure 4.6.



(a) f_{ED} (b) f_{PC} (c) f^3_{ST} (d) $f_{\mathrm{ST}m}$ (e) f_{TG}

(f) f^3_{AV} (g) f^3_{LV} (h) f^3_{CS} (i) f_{RC} (j) f_{RA} 4.1c (k) All 4.1c

Figure 4.8: Shows the performance of random forest on sequence 4.1c using the different set of features.



(a) f_{ED} (b) f_{PC} (c) f^3_{ST} (d) $f_{\mathrm{ST}m}$ (e) f_{TG}

(f) f^3_{AV} (g) f^3_{LV} (h) f^3_{CS} (i) f_{RC} (j) f_{RA} (k) All

Figure 4.9: Shows the performance of random forest on sequence 4.1i using the different set of features.

4.4.1 Final Feature Set

To evaluate the performance of individual features, we train random forests using only single features. The same cross-validation technique is used for obtaining comparative results across all sequences. The results can be seen in Table 4.1.

Observing the results for all features, we finalize our feature set as follows:

$$\begin{aligned}\mathbf{x}_i = \{\, & f_{\mathrm{ED}}(x,y,[1-10]),\ f_{\mathrm{PC}}(x,y,[1-4],[1-k]),\ f^3_{\mathrm{ST}}(x,y,[1],[1-k]),\ f_{\mathrm{ST}m}(x,y,[1],[1-k]),\\ & f_{\mathrm{TG},x}(x,y,[1-10]),\ f_{\mathrm{TG},y}(x,y,[1-10]),\ f^3_{\mathrm{AV}}(x,y,[1-4],[1-k]),\ f^3_{\mathrm{LV}}(x,y,[1-4],[1-k]),\\ & f^3_{\mathrm{CS,max}}(x,y,[1-4],[1-k]),\ f^3_{\mathrm{CS,min}}(x,y,[1-4],[1-k]),\ f^3_{\mathrm{CS},\sigma^2}(x,y,[1-4],[1-k]),\\ & f_{\mathrm{RC}}(x,y,[1-10],[1-k]),\ f_{\mathrm{RA}}(x,y,[1-10],[1-k])\,\}\end{aligned}$$

As discussed in Section 4.4 we observe that f_{ED} performs comparatively better than f_{PB}. Also notice the reasonable performance of features f_{PC}, f^3_{ST}, $f_{\mathrm{ST}m}$, f_{RC} and f_{RA} throughout all the sequences. In our initial tests we also verified the performance of a classifier trained only using features f^3_{AV}, f^3_{LV}, f^3_{CS} and f^3_{PC}. The results using these features are quite comparable to a random forest trained using all our features. At this stage we do not want to discount a large number of features because random forests are good at ranking and discarding features as need be. Also notice some sequences have visibly higher misclassification rate. The problems in these sequences will be discussed in the following sections.

Figure 4.7 to 4.9 shows the output posterior overlayed onto the test sequences 4.1b, 4.1c and 4.1i. Notice how features based on flow do well on out-of-FOV pixels (see Section 4.5). Also note the performance of features on untextured regions.

For the final set of features, Figure 4.10 shows the importance assigned to each feature by the random

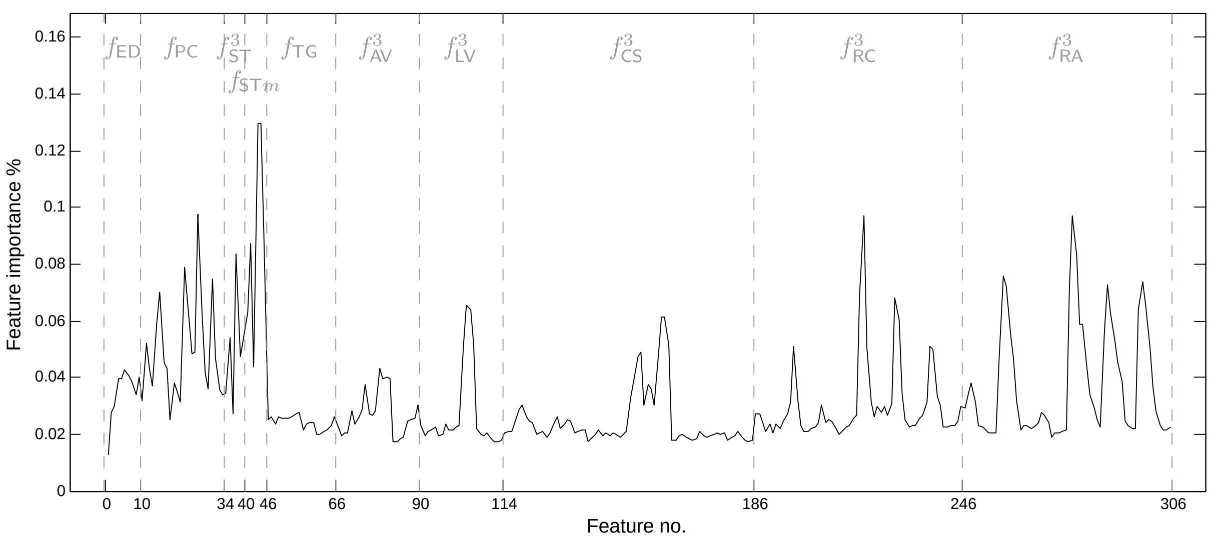


Figure 4.10: The graph shows the relative importance of variables assigned by the random forest as a result of training. The values are obtained by averaging all the variable importances *output* when cross-validating the 10 sequences given in Section 4.2. The forest was trained with the features finalized for our classifier. Each feature type is printed on top of the graph.

forest. This is the average for cross-validation of the 10 sequences. For f_{ED} increasing feature numbers gives variables when we proceed up in the image pyramid (as Gaussian becomes wider). For features f_{PC}, f^3_{AV}, f^3_{LV}, f_{RC} and f_{RA} variables are first ordered by the flow algorithms and then with the increasing Gaussian pyramid scale. f_{TG} is first ordered by pyramid scale and then by its x and y components. The first 24 features for f^3_{CS} belong to $f^3_{\mathrm{CS,max}}$ - which are ordered first by flow algorithms and then by the pyramid scale. Similarly, the next 24 features in it belong to $f^3_{\mathrm{CS,min}}$, and the next 24 to $f^3_{\mathrm{CS},\sigma^2}$. f^3_{ST} and $f_{\mathrm{ST}m}$ are ordered by flow algorithms.

4.5 Cropping out-of-FOV regions

Some pixels move out of the frame when going from image I_1 to I_2. In our training dataset, this is due to camera motion in the synthetic training data ([29]) and due to non-rigid object movement in the natural sequences ([1]). We will call all such pixels as *out-of-FOV* pixels.

During our experiments, it was apparent that features based on flow performed well on such pixels. The reason for this good performance could be because there is little need for accuracy from flow at such regions - as long as a flow algorithm directs a pixel beyond image boundaries, these pixels can be classified correctly. Needless to say that the dependency of our features on flow make these flow algorithms crucial in classification of any region. Lowering expectations of accuracy of flow for any pixel will lower the misclassification rate.

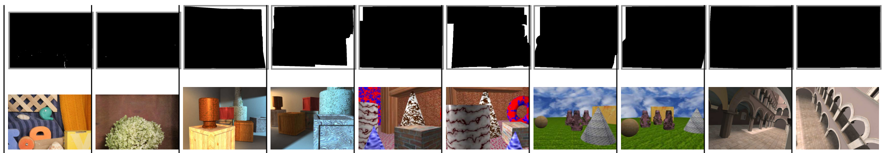
Since we are certain that some features perform well on out-of-FOV regions, it is important to check the performance sans these pixels. In this section we develop an experiment to analyze the classifier's performance only on pixels which stay inside the FOV. To remove influence of these out-of-FOV regions, we use a mask which blocks features from getting sampled from these regions. These masks are given in Table 4.2. We use the same set of features finalized in Section 4.4.1 throughout this experiment. Like before, we use k-fold cross-validation to test the complete training set. Apart from testing with forests trained on the complete set of features, we also experimented forests trained on single features. This is done to see which features perform best on within-FOV pixels. As explained in Section 4.4.1, this could also be done by analyzing the ranking of features given by the random forest - but this method has the drawback of not giving representative scores when there is strong correlation between features.

To remain fair, all statistics drawn from the output of these forests use only pixels which are not in the out-of-FOV mask i.e. the ROC produced does not take into account out-of-FOV regions. Table 4.2 shows results from testing using single features and the complete set of final features.

As expected, performance remains unchanges on sequences which have a miniscule number of out-of-FOV pixels. On other sequences like 4.1c, the performance on within-FOV pixels is below average as compared to other sequences. This can be attributed to the large amount of camera shift. When this happens in conjunction with low texture pixels getting occluded, the performance hit can be significant (see Section 4.6). A similar case occurs for features in

Surprisingly, f^3_{AV}, f^3_{LV}, and f^3_{CS} perform much better when out-of-FOV pixels are not present. This could be because these flow based features overfit on characteristics of out-of-FOV regions. This is a likely possibility, since, as stated before, it is easy for features based on flow to learn such regions. Apart from these three features, f_{ED} and f_{TG} also perform better in this experiment. Since the latter is based on flow, similar reasons can be put forward for its performance improvement. The features based on reverse flow and texture perform worse in this scenario.

If we can separate features performing well on out-of-FOV pixels from those performing better on within-FOV regions, we can formulate our problem into a ternary classification task. It is worth a



	seq. 4.1a	seq. 4.1b	seq. 4.1c	seq. 4.1d	seq. 4.1e	seq. 4.1f	seq. 4.1g	seq. 4.1h	seq. 4.1i	seq. 4.1j
$f_{\mathrm{ED}}(x, y, [1-10])$	0.829 115%	0.815 116%	0.627 152%	0.658 116%	0.695 116%	0.513 93%	0.544 107%	0.736 120%	0.704 122%	0.759 134%
$f_{\mathrm{PC}}(x, y, [1-4], [1-k])$	0.948 100%	0.893 99%	0.723 79%	0.613 63%	0.954 99%	0.850 91%	0.849 90%	0.968 99%	0.871 96%	0.702 79%
$f_{\mathrm{ST}m}(x, y, [1], [1-k])$	0.953 99%	0.882 98%	0.576 67%	0.645 68%	0.883 95%	0.757 84%	0.871 93%	0.952 98%	0.853 96%	0.797 87%
$f_{\mathrm{TG}}(x, y, [1-10])$	0.920 124%	0.886 120%	0.536 99%	0.680 109%	0.865 116%	0.768 125%	0.728 114%	0.799 127%	0.779 150%	0.723 145%
$f^3_{\mathrm{AV}}(x, y, [1-4], [1-k])$	0.938 111%	0.869 113%	0.648 167%	0.763 125%	0.853 114%	0.857 150%	0.821 132%	0.887 129%	0.852 113%	0.773 150%
$f^3_{\mathrm{LV}}(x, y, [1-4], [1-k])$	0.951 114%	0.895 125%	0.630 92%	0.792 106%	0.933 108%	0.857 115%	0.793 108%	0.922 106%	0.912 114%	0.867 123%
$f^3_{\mathrm{CS}}(x, y, [1-4], [1-k])$	0.964 105%	0.900 116%	0.687 102%	0.796 101%	0.936 103%	0.823 110%	0.831 104%	0.947 106%	0.937 106%	0.905 116%
$f_{\mathrm{RC}}(x, y, [1-10], [1-k])$	0.960 99%	0.888 99%	0.715 78%	0.692 72%	0.957 99%	0.838 91%	0.831 89%	0.954 98%	0.771 92%	0.684 78%
$f_{\mathrm{RA}}(x, y, [1-10], [1-k])$	0.938 97%	0.887 99%	0.723 78%	0.726 74%	0.904 97%	0.826 89%	0.838 89%	0.940 97%	0.892 96%	0.643 75%
All	0.977 99%	0.920 99%	0.684 75%	0.793 81%	0.963 99%	0.895 93%	0.895 93%	0.981 99%	0.951 101%	0.880 93%

Table 4.2: Shows the comparison of a forest trained and tested over all pixels against a forest trained and tested only over pixels which remain inside the field-of-view (FOV) over the sequence. The header shows the sequences and the mask used to ignore the out-of-FOV pixels. Like Table 4.1, each column shows the result (area under the ROC curve - AUC) on a sequence using k-fold cross-validation. The values in gray show the percentage difference in AUC while training with or without the pixels out-of-FOV (results when including out-of-FOV pixels shown in Table 4.1). The dark gray row shows results using all the features finalized in a single random forest. Note the effect on the classification accuracy when the number of out-of-FOV pixels is significant (sequences 4.1c, 4.1d, 4.1f, 4.1g, and 4.1h).

thought that the our framework might be transformed to a two staged process: in the first we attempt to find out-of-FOV pixels; and in the second, we mask the out-of-FOV regions found and use features that perform better within-FOV to classify the remaining pixels. This will result in an output label Y with values non-occluded, occluded, or out-of-FOV pixels.

4.6 Effect of Texture

Observing results of sequence 4.1c, it is apparent where our classifier learns weakly. Since there is no inference on the scene structure, the forests tend to perform badly whenever there is wide-baseline camera movement causing large parts of the scene to be occluded. When this occurs on regions lacking texture, the optical flow algorithms we use tend to compress pixels to a certain region rather than assigning good flow to some and giving confused flow on occluded pixels. A good example is the left side of the vase in Figure 4.8k. We think this situation can be improved by having features which explicitly infer scene structure. For this purpose, many stereo algorithms compute an occlusion map with depth disparities [53, 56, 37, 63]. Even when computing optical flow, Strecha et al. [52] proposes a solution by considering large displacement occlusion pixels as hidden quantities in an EM framework. Such techniques can surely act as additional features in our framework.

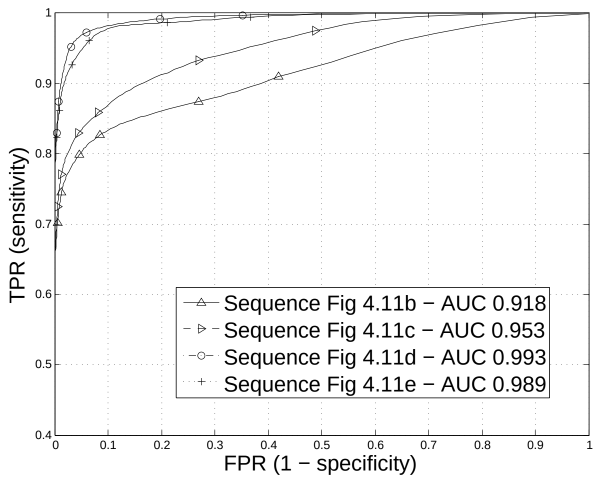
In the experiments in this section, we analyze the role of texture on the classification accuracy. The goal is to vary texture and see the effects on the sequences with wide-baseline camera movement - without making inferences on the scene structure. Using the sequences 4.1c and 4.1d, we adjust texture using `Maya` while keeping lighting, object placement, and camera FOV constant. We conducted two sets of experiments: one where we vary the texture of 4.1c and the other where we vary texture of 4.1d. Each test has four sequences with variations of texture (including the original texture sequence given in Section 4.2). Amongst them, two sequences lack textures, whereas the other two have significant texture. To test, we use forests trained using the following set of features:

$$\begin{aligned}\mathbf{x}_i = \{\ & f_{\mathrm{GM}}(x,y,[1-10]),\ \ f_{\mathrm{ED}}(x,y,[1-10]),\ \ f_{\mathrm{PC}}(x,y,[1-4],[1-k]),\ \ f_{\mathrm{ST}m}(x,y,[1],[1-k]),\\ & f_{\mathrm{TG},x}(x,y,[1-10]),\ \ f_{\mathrm{TG},y}(x,y,[1-10]),\ \ f^3_{\mathrm{AV}}(x,y,[1-4],[1-k]),\ \ f^3_{\mathrm{LV}}(x,y,[1-4],[1-k]),\\ & f^3_{\mathrm{CS,max}}(x,y,[1-4],[1-k]),\ \ f^3_{\mathrm{CS,min}}(x,y,[1-4],[1-k]),\ \ f^3_{\mathrm{CS},\sigma^2}(x,y,[1-4],[1-k]),\\ & f_{\mathrm{RC}}(x,y,[1-10],[1-k]),\ \ f_{\mathrm{RA}}(x,y,[1-10],[1-k])\ \}\end{aligned}$$

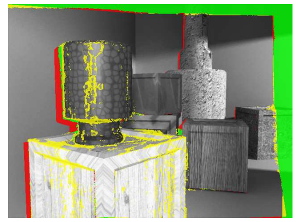
ROC curves computed on the output posteriors for these texture sequences are given in Figure 4.11. Like before k-fold cross-validation was used for testing. Although in these experiments the testing data included the new texture sequences. To avoid using data from the same sequence (with or without texture), we made sure when testing for a sequence, lets say sequence 4.11c, we did not train on its companion textured sequences i.e. sequence 4.11b, 4.11d, and 4.11e. Hence for testing a sequence, we have 12 sequences to train on. All other parameters remain the same.

The experiments clearly show that texture plays a key role in such scenarios. The ROC curves indicate that increasing texture significantly improves performance. This can be largely attributed to the comparatively better outputs of the flow algorithms. Notice the increase in classification accuracy on the left side of the vase in Figures 4.11d and 4.11e. Also notice the change in performance around the vase from Figures 4.11g to 4.11j. Apart from pixels whose texture has changed, there is also decrease in misclassification on regions whose texture has not changed. The crate and the surface of the vase in Figures 4.11b to 4.11e are good examples. As additional experiments, it would be interesting to see the classifier's output on low textured images when training on just high texture sequences.

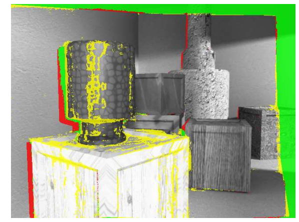
Concluding from these experiments, we expect that occlusion regions with less texture are more likely to be misclassified in our framework as compared to occluded pixels with significant texture.



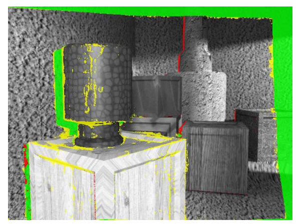
(a)



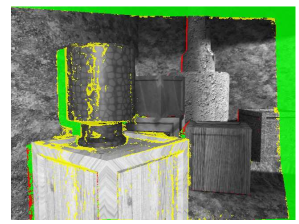
(b) Original - seq. 4.1c



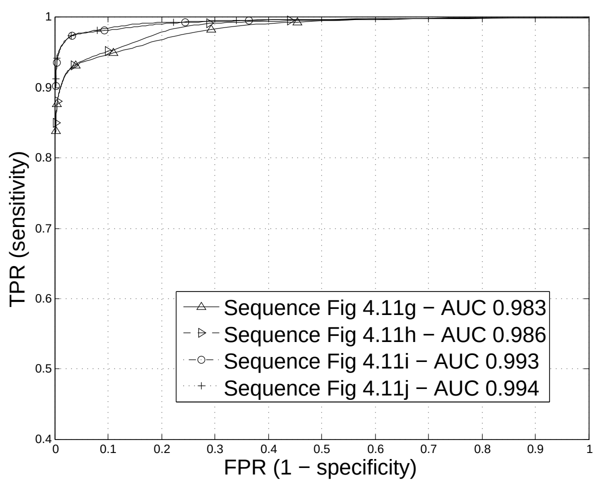
(c) Texture 1 - seq. 4.1c



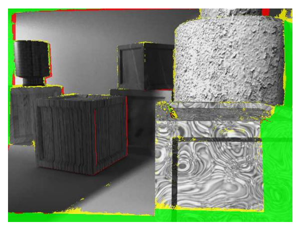
(d) Texture 2 - seq. 4.1c



(e) Texture 3 - seq. 4.1c

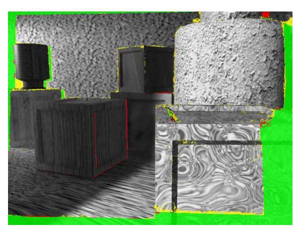


(f)

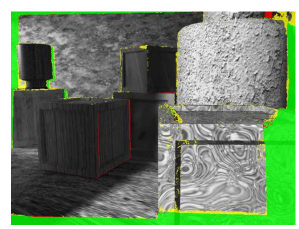


(g) Original - seq. 4.1d

(h) Texture 1 - seq. 4.1d



(i) Texture 2 - seq. 4.1d



(j) Texture 3 - seq. 4.1d

Figure 4.11: Shows the effect of texture on the classification accuracy. Each ROC plot is produced by training and testing a classifier on one of the 4 images on its right. The plot's legend indicates which curve was produced by which texture sequence. The legend also gives the respective area under the curve statistic. Seq. 4.11b and Seq. 4.11g are the original images from the training dataset. Seq. 4.11c and Seq. 4.11h are produced with a slightly enhanced texture. Seq. 4.11d/4.11e and Seq. 4.11i/4.11j are produced by replacing all "no-texture" planes with significant texture. The images on the right are overlayed with the posterior output using the method explained in Figure 4.6. The lighting, object placement, and camera FOV were kept constant in both experiments.

4.7 Using ground-truth flow

As discussed in Section 3.2.2, a large number of our features depend on optical flow methods. The performance of these features can only be as good as the performance of the underlying flow algorithms we use. In this section we experiment with the idea of training and testing with accurate (Ground-Truth) flow. This will allow us to see how accurate our method (features) can be if a candidate flow algorithm was flawless.

The training set we have used uptil now (Section 4.2) comes with GT flow where occlusion regions have been marked. This was an advantage that allowed us to train and test to evaluate our method, but here it is a drawback since we need to have flow on all pixels to compute the complete set of features. In lieu of this dataset, we use a different set of 4 training sequences from Baker et al. [1] - where the GT flow is provided for even occluded pixels. We also add the Yosemite sequence [2] to this new dataset. Since these sequences provide flow for all pixels, making the occlusion GT becomes a challenge. If the flow is not large and there is little change in lighting, we can suppose that the intensity of pixels will not vary much. This is the photo-constancy assumption. Using the flow provided, we can select a threshold for the photo-constancy feature (f_{PC}) which produces visibly accurate occlusion regions. We use this technique on these 5 sequences as a starting point to produce the GT occlusion mask. Although this gives reasonable results it is plagued with the same problems as the photo-constancy feature. To obtain a better results we manually touch-up the GT using `Photoshop`. This gives a reasonable occlusion GT to test our hypothesis. You can see the GT masks produced in Figure 4.12. A better method to create a dataset for this section would be to create new synthetic sequences which has two ground-truths: one for flow and one for occlusion.

Using k-fold cross-validation we train and test our random forest, using the set of features finalized in Section 4.4.1. The results are given in Figure 4.12. For comparison, we also test using the same set of features, but this time using the 6 candidate flow algorithms. The results look slightly better with these sequences as compared to results in other sections, partially due to the use of photo-constancy to create the GT. This is also apparent in the variable importance output by the forest, where f_{PC} is assigned significantly higher importance than usual. Nevertheless it is clear that using GT flow can increase the classifier's accuracy. Notice the errors the classifier makes on *Venus*, *Urban3* and *Grove2* using the flow algorithms. These errors are significantly curtailed when using GT flow.

In conclusion, it would be interesting to see how our framework performs as better flow algorithms are proposed.

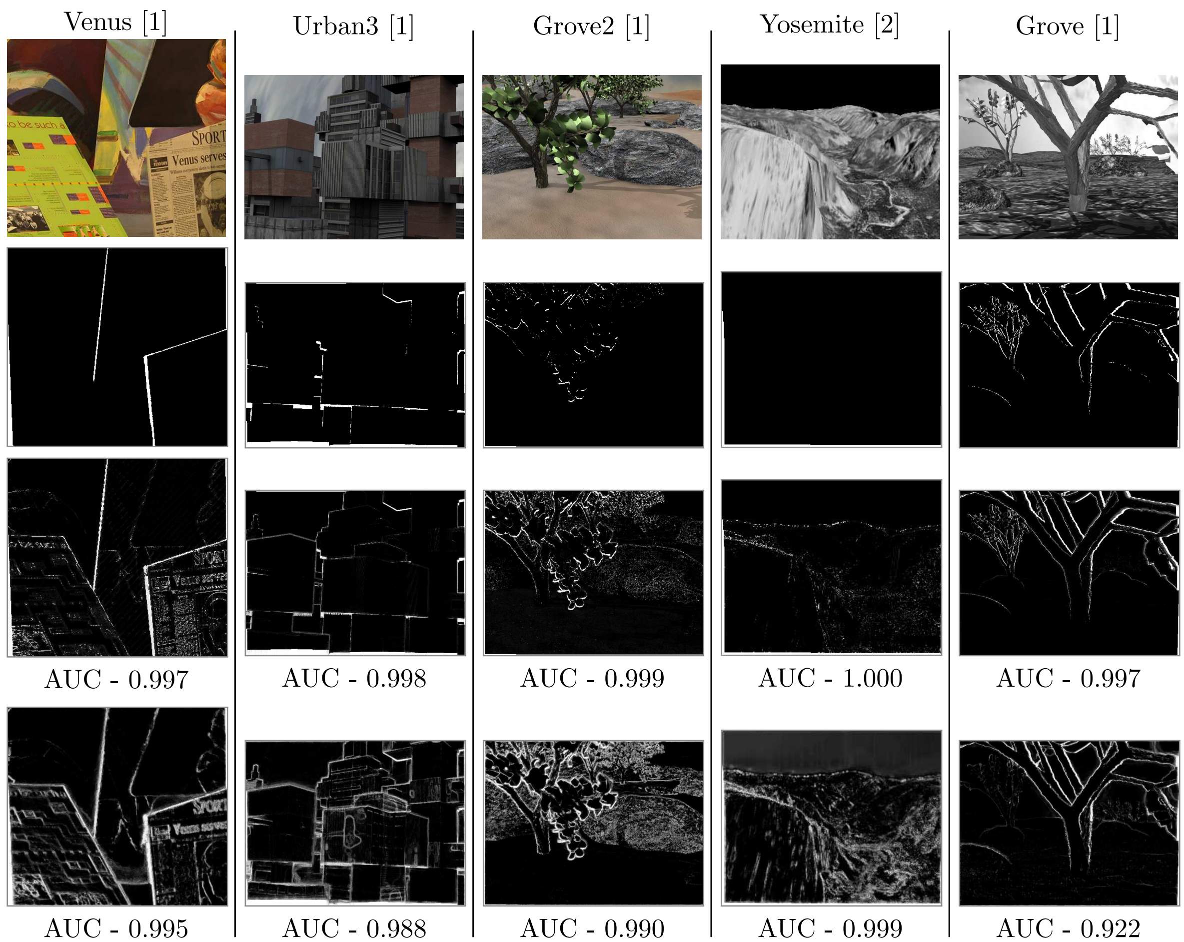


Figure 4.12: Shows comparative results when using GT flow. The **first row** in each column shows I_1 for the sequence; the **second row** shows the GT produced using GT flow; the **third row** shows the output posterior when the random forest is trained with features using the GT flow; and the **fourth row** shows the posterior when the forest is trained as normal (using the 6 flow algorithms). Below each posterior output, the area under the ROC curve is given.

Chapter 5

Results

Having trained and tested our method on the training dataset, we would like to test our method on un-seen sequences. To see the classification accuracy and the generalization capability of our classifier, we use a variety of datasets. In the first Section (5.1) we show quantitative results on sequences where the occlusion region ground-truth is present (similar to our training set). In Section 5.2 we provide qualitative comparisons to Stein and Hebert [50], which we think is the current state-of-art in occlusion *boundary* detection. In the final Section (5.3) we show qualitative results on popular datasets.

5.1 Results on Sequences with GT

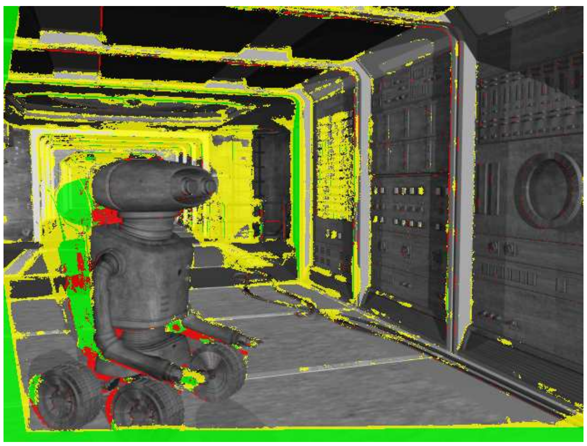
From our original dataset comprising of (see Section 4.2), we removed some sequences for this stage. One of them is the synthetic robot sequence produced by Mac Aodha et al. [29]. The result on it is given in two forms in Figure 5.1. This sequence is unique in some ways: there is significant depth in the scene since it is modeled in a hallway; moreover it has small surfaces which are occluded from one frame to the next (the panels on the walls). Interestingly the classifier does well on objects nearby, but its accuracy increases with depth. This coupled with the fact that the robot acts as an occluder to the hallway, produces relatively weak results around the robot and into the hallway.

The second sequence is a grass-sky also produced by produced by Mac Aodha et al. [29]. Using this sequence of 11 frames, occlusions are classified between each pair of frames. Since our training sequences, Mayan 1 (4.1g) and Mayan 2 (4.1h), uses frames from grass-sky, to test this sequence, a new classifier is created using the remaining 8 sequences as the training set. The classifier performs reasonably well on all regions except the ground-plane and occasionally on the side of the statues. One can also observe an interesing case of texture here. The classifier tends to perform weakly next to the statues when the occluded surface is the background sky - but the same regions adjacent to the statues are classified well when the occluded surface is the wall.

5.2 Comparative results on Stein and Hebert [50] dataset

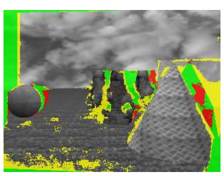
We intdroduced the work of Stein and Hebert [50] on detection occlusion boundaries in Section 2.2.1. As discussed before, they propose a method to classify boundaries in a scene where occlusions occur, using *learned* appearance and motion cues. They use sequences with 6 or more frames to make occlusion boundary predictions on the center pair of images in the sequence. In this section we compare our results to their method on the dataset they provide. Although not an apples to apples comparison, it is worth looking if our occlusion *regions* have any correlation with the occlusion boundary ground-truth provided and their results on detecting occlusion boundary fragments.

The classifier as before was trained using the dataset given in Section 4.2 and the features finalized in Section 4.4.1. Although we tested our classifier on all 30 sequences provided in this dataset, we display

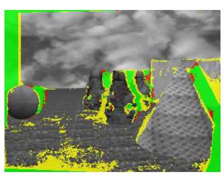


(a) Robot I_1 [29] overlayed with output - AUC 0.917

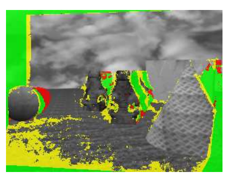
(b) Robot posterior



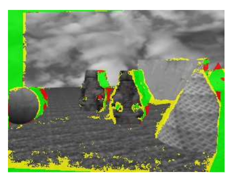
(c) Grass-sky [29] I_1 - AUC 0.950



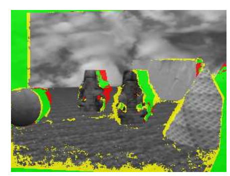
(d) Grass-sky I_2 - AUC 0.977



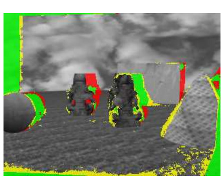
(e) Grass-sky I_3 - AUC 0.965



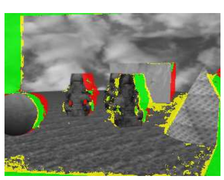
(f) Grass-sky I_4 - AUC 0.972



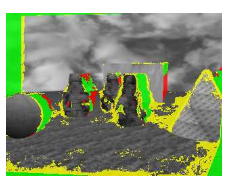
(g) Grass-sky I_5 - AUC 0.965



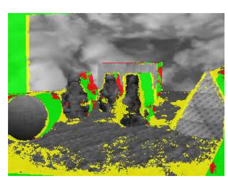
(h) Grass-sky I_6 - AUC 0.959



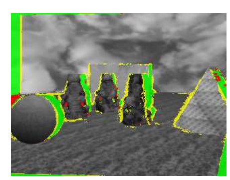
(i) Grass-sky I_7 - AUC 0.955



(j) Grass-sky I_8 - AUC 0.961



(k) Grass-sky I_9 - AUC 0.938



(l) Grass-sky I_{10} - AUC 0.985

Figure 5.1: Shows the GT evaluation using two sets of sequences, both taken from Mac Aodha et al. [29]. The **first row** shows results with the robot sequence. Images in this figure are overlayed with the posterior output using the method explained in Figure 4.6, except Figure 5.1b, which is the direct posterior output of the classifier. The **second row** and **third row** gives results on the 10 frame grass-sky sequence. All area under the curve of ROC are given in captions.

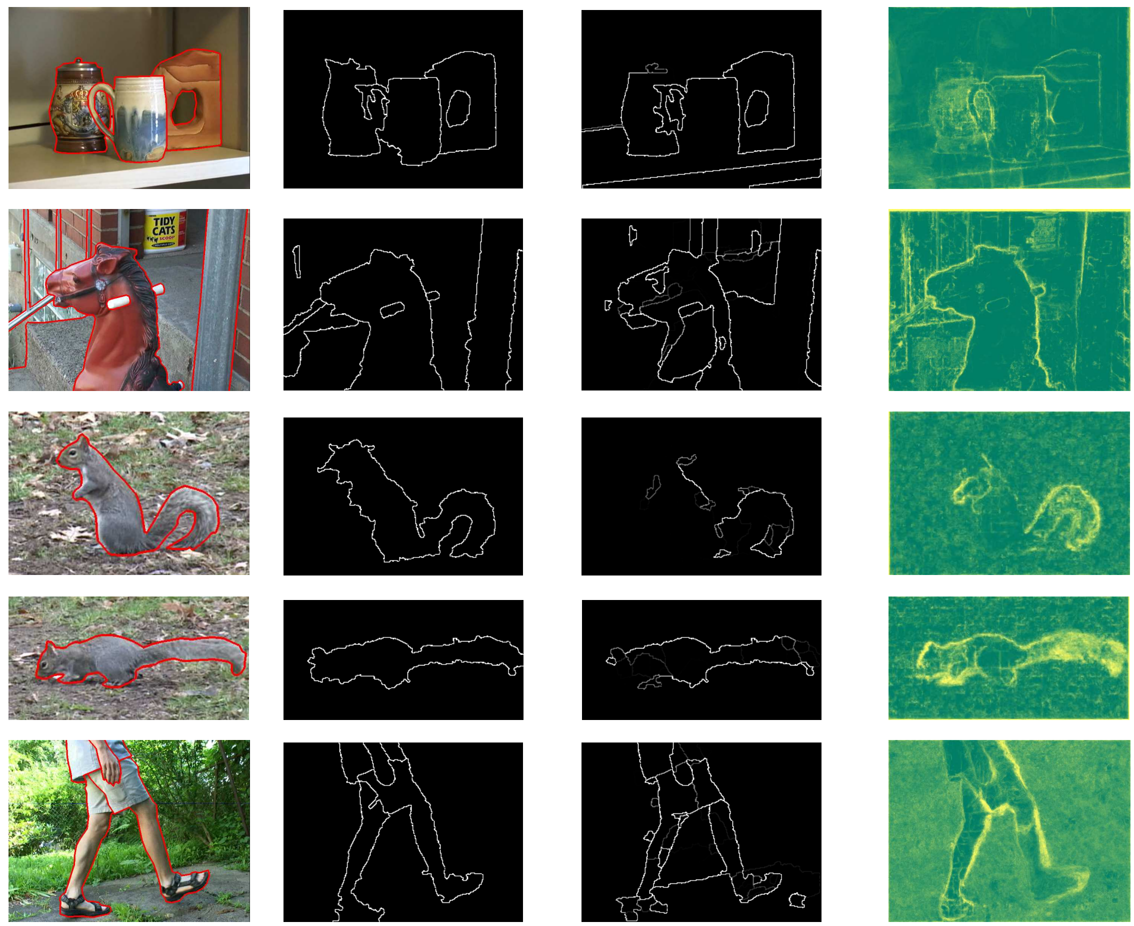


Table 5.1: Comparative results against the occlusion boundary GT and results provided by Stein and Hebert [50]. The **first column** of images shows the middle frame on which [50] computes occlusion boundaries. The frame is overlayed in red with the GT object layer boundaries. The **second column** shows the boundary fragment GT created using the segmentation method discussed in Section 2.2.1 and fragment chaining. Stein and Hebert [50] use this GT for training their classifier. Their results are given in the **third column**. Note all GT was is provided by Stein and Hebert [50]. The **last column** shows our results where color ranges from green (low occlusion region probability) to yellow (high probability).

results here for 12 chosen sequence, where in some the classifier seems to be performing qualitatively well, and in others the posterior output is quite noisy.

Notice the classifier's ability on the Rocking Horse and Walking Legs sequence in Table 5.1. The classifier scores very highly on regions where the rocking horse or the legs are moving over to. Also note the results on the Squirrel 2 sequence where the major movement is on the squirrel's tail and hands.

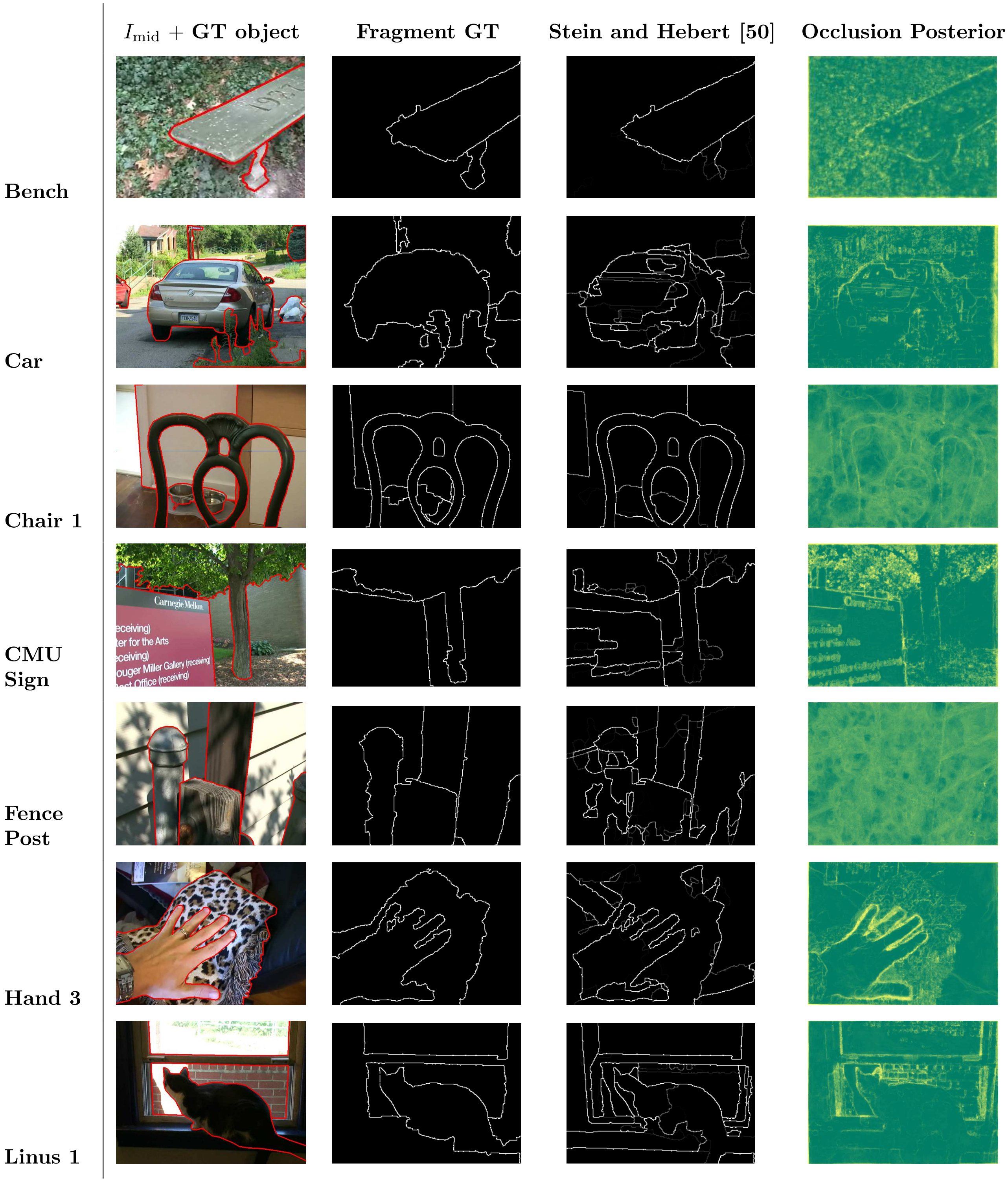
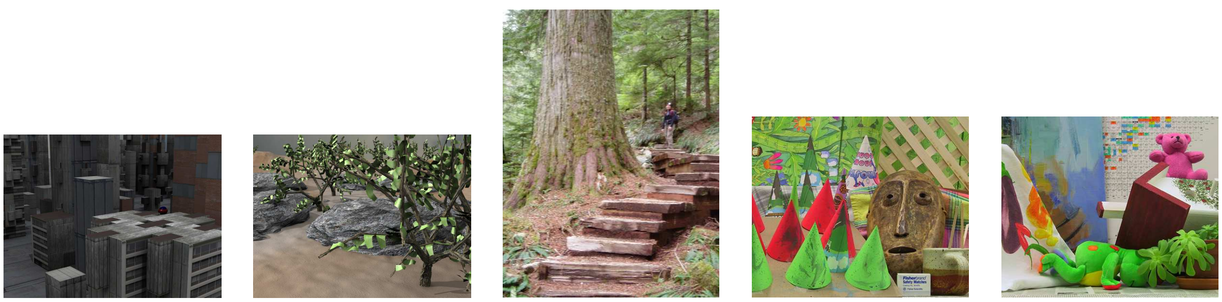


Table 5.2: See Table 5.1 for the description.

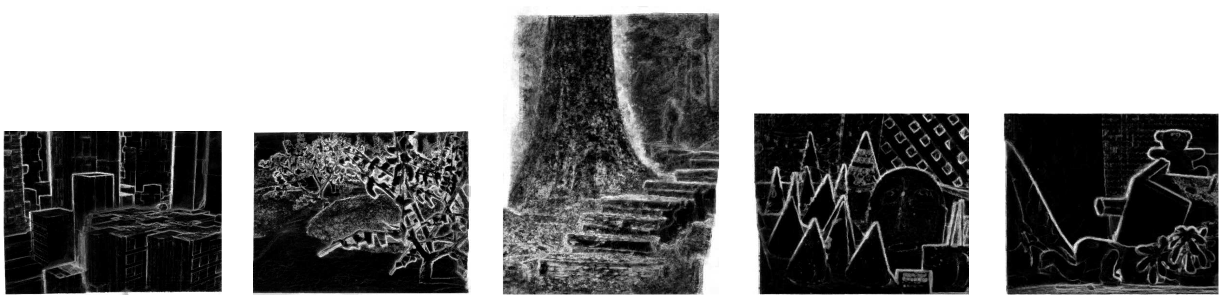
On the other hand, notice the relatively poor performance on the mug sequence. This is due to the significantly difficult texture of the occluded surface. Interestingly, this where Stein and Hebert [50] method also performs badly. Although, the surface of occluding mug handle itself is marked with a low occlusion probability, whereas the former method performs poorly on all areas close to the texture.

For sequences in Table 5.2, the Hand 3 sequence seems to perform the best. This could be due to the strong contrast of the hand against the occluded pillow. As we have seen before, our method tends to perform badly wherever lighting changes occur. The Fence Post sequence is no exception, as the classifier only shows small discriminative ability on the fence post. For the former sequence and Chair 1, it would be interesting to analyze if the optical flow algorithms are breaking down, or the features themselves are to be blamed. Notice also the CMU Sign sequence, where the lettering on the board is also being classified as occlusion regions. This could be due to the specularities on this sign board.

The GT and results of Stein and Hebert [50] can be accessed from `http://www.cs.cmu.edu/~stein/occlusion_data/`.



(a) Urban2 [1] I_1 (b) Grove3 [1] I_1 (c) Forest [68] I_1 (d) Cones [44] I_1 (e) Teddy [44] I_1



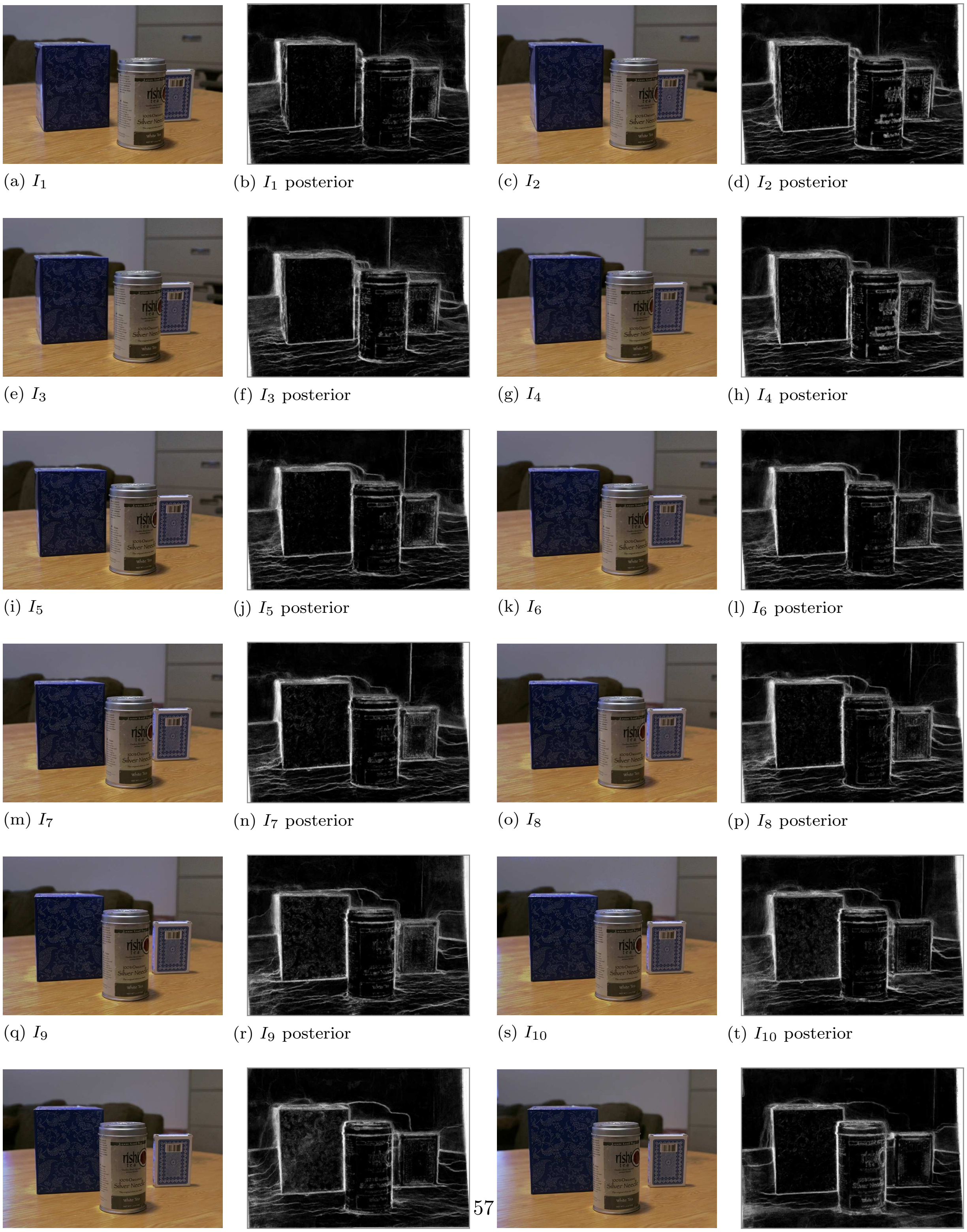
(f) Urban2 posterior (g) Grove3 posterior (h) Forest posterior (i) Cones posterior (j) Teddy posterior

Figure 5.2: Shows results on evaluation sequences which have no GT. Sequences have been taken from Baker et al. [1], Zitnick et al. [68], and Scharstein and Szeliski [44]. The **first row** shows the input image I_1 of the sequence. The **second row** gives results on the respective sequences. Lighter values indicate a higher posterior probability.

5.3 Results on Sequences with no GT

As last set of tests, we use datasets which are popular in the vision community. This includes some untested sequences from the middlebury stereo dataset [44], and the middlebury flow dataset [1]. In these sequences, we also include the forest sequence from [68], which is a considerably hard to classify due to large varying texture and significant camera motion. Nevertheless our method seems to resolve the right side of the trunk in the sequence (see Figure 5.2h).

We also test our method on the rotating table sequence from the MIT human annotated dataset [27]. The results on the 13 frame sequence can be seen in Figure 5.3. Notice the effect of specular reflection in Figures 5.3u, and 5.3w. Also note the classification of the left-side of the jar as we mover over the sequence. Also the disappearing left-face of the blue box makes an interesting case for occlusion classification.



(a) I_1 (b) I_1 posterior (c) I_2 (d) I_2 posterior

(e) I_3 (f) I_3 posterior (g) I_4 (h) I_4 posterior

(i) I_5 (j) I_5 posterior (k) I_6 (l) I_6 posterior

(m) I_7 (n) I_7 posterior (o) I_8 (p) I_8 posterior

(q) I_9 (r) I_9 posterior (s) I_{10} (t) I_{10} posterior

(u) I_{11} (v) I_{11} posterior (w) I_{12} (x) I_{12} posterior

Figure 5.3: Classification results on table sequence from Liu et al. [27].

Chapter 6

Conclusions and Future Work

This thesis proposes combining multiple low level visual features in a framework to classify regions of occlusion. As discussed in the introduction, occlusions pose significant hurdles in the computation of flow, motion segmentation and even stereo. Finding these regions is critical to good performance in all such techniques. Despite the importance of detecting occlusions, there has been no standard procedure to classify such pixels reliably.

This thesis proposed an algorithm which *learns* regions of occlusions. Our major contribution was the set of features correlated with occlusion regions and the accompanying framework to classify pixels. Although the method proposed provided good classification ability using our features, the framework is open-ended to accept new features sets.

We have based a majority of our features on optical flow, since without any temporal reasoning over the image sequence there is no concept of occlusions. Some features in our set work purely on flow whereas others take image properties into consideration. We also compute features using flows from different candidate algorithms which reduces chances of the classifier being biased by the faults of a particular flow algorithm.

The results overall are promising when compared to the current state-of-art algorithms. This demonstrates that combining features in a classification framework can correctly identify occlusions in some cases. However, as discussed in the next section, some problems in the classifier still remain unanswered.

6.1 Future Work

One drawback we observed with our framework during testing was the decrease in performance while working with sequences undergoing large change in field-of-view. We concluded that this was mostly due to the lack of features having any understanding of the scene geometry. It should be possible improve the accuracy of our classifier by incorporating features from stereo algorithms. Since we would not want such features to hurt our classifier's performance during small baseline camera movements, it might be feasible to weight such features by the amount of camera movement (a simple 8-point algorithm might work here).

We also noticed that our classifier's performance was unpredictable on areas of high texture or on texture-less surfaces. It might be of interest to find better perform texture features. To this end, we tried both *Gabor* and *A Sparse Set of Texture Features*. Although they perform reasonably, there is still an opportunity for improvement.

Another concern with our algorithm could be speed. Although the trained classifier just takes around 90 seconds per sequence, another 30 minutes are required to compute the whole feature set. Here decreasing the size of the feature set can have dual advantages. Apart from the decrease in the computation cost for these features, the size of the classifier will also help reduce the 90 seconds taken to test the data (note that a majority of this time is consumed in loading the classifier into memory).

.1 Code Appendix

Listing 1: AbstractFeature class

```
classdef AbstractFeature
    %ABSTRACTFEATURE Abstract class for computing a feature

    properties (Abstract, Constant)
        FEATURE_TYPE;
        FEATURE_SHORT_TYPE;
    end


    methods (Abstract)
        [ grad feature_depth ] = calcFeatures( obj, calc_feature_vec );
    end


    methods
        function feature_no_id = returnNoID(obj)
        % creates unique feature number, good for storing with the file
        % name

            % create unique ID
            nos = uint8(obj.FEATURE_SHORT_TYPE);
            nos = double(nos) .* ([1:length(nos)].^2);
            feature_no_id = sum(nos);
        end


        function return_feature_list = returnFeatureList(obj)
        % creates a cell vector where each item contains a string of the
        % feature type (in the order the will be spit out by calcFeatures)

            return_feature_list{1} = {obj.FEATURE_TYPE, 'no_scaling'};
        end
    end

end
```

Listing 2: EdgeDistFeature class

```
classdef EdgeDistFeature < AbstractFeature
    %EDGEDISTFEATURE the distance transfrom from the edges in the first
```

```
    %   image (using canny edge detector). The constructor either takes
    %   nothing or size 2 vector for computing the feature on scalespace
    %   (first value: number of scales, second value: resizing factor). If
    %   using scalespace, ComputeFeatureVectors object passed to
    %   calcFeatures should have im1_scalespace (the scalespace structure),
    %   apart from image_sz. image_sz and im1_gray are required for
    %   computing this feature without scalespace. . If using the
    %   scalespace, usually, the output features go up in the scalespace
    %   (increasing gaussian std-dev) with increasing depth.


    properties
        no_scales = 1;
        scale = 1;
    end


    properties (Constant)
        FEATURE_TYPE = 'Edge Distance';
        FEATURE_SHORT_TYPE = 'ED';
    end


    methods
        function obj = EdgeDistFeature( varargin )
            if nargin > 0 && isvector(varargin{1}) && length(varargin{1}) == 2
                obj.no_scales = varargin{1}(1);
                obj.scale = varargin{1}(2);
            end
        end


        function [ dist feature_depth ] = calcFeatures( obj, calc_feature_vec )
        % this function outputs the feature for this class, and the depth
        %   of this feature (number of unique features associated with this
        %   class). The size of dist is the same as the input image, with a
        %   depth equivalent to the number of scales

            if obj.no_scales > 1
                assert(~isempty(fields(calc_feature_vec.im1_scalespace)), 'The scale space for im 1 has not been defined in the passed ComputeFeatureVectors');

                assert(calc_feature_vec.im1_scalespace.scale == obj.scale &&
```

```
                ...
                calc_feature_vec.im1_scalespace.no_scales >= obj.
                    no_scales, 'The scale space given for im 1 in 
                    ComputeFeatureVectors is incompatible');

            % initialize the output feature
            dist = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.
                image_sz(2), obj.no_scales);

            % iterate for multiple scales
            for scale_idx = 1:obj.no_scales
                % get the next flow image in the scale space
                im_resized = calc_feature_vec.im1_scalespace.ss{
                    scale_idx};

                % compute the edge image
                edge_im = edge(im_resized, 'canny');

                % compute distance transform and resize it to the
                    original image size
                dist(:,:,scale_idx) = imresize(bwdist(edge_im),
                    calc_feature_vec.image_sz);
            end
        else
            % compute the edge image
            edge_im = edge(calc_feature_vec.im1_gray, 'canny');

            % compute distance transform and resize it to the original
                image size
            dist = imresize(bwdist(edge_im), calc_feature_vec.image_sz);
        end

        feature_depth = size(dist,3);
    end


    function feature_no_id = returnNoID(obj)
    % creates unique feature number, good for storing with the file
    % name

        % create unique ID
        nos = returnNoID@AbstractFeature(obj);

        temp = obj.no_scales^obj.scale;
        % get first 2 decimal digits
        temp = mod(round(temp*100), 100);
```

```
            feature_no_id = (nos*100) + temp;
        end


        function return_feature_list = returnFeatureList(obj)
        % creates a cell vector where each item contains a string of the
        % feature type (in the order the will be spit out by calcFeatures)

            return_feature_list = cell(obj.no_scales,1);

            return_feature_list{1} = {obj.FEATURE_TYPE, 'no scaling'};

            for scale_id = 2:obj.no_scales
                return_feature_list{scale_id} = {obj.FEATURE_TYPE, ['scale '
                    num2str(scale_id)], ['size ' sprintf('%.1f%%', (obj.
                    scale^(scale_id-1))*100)]};
            end
        end
    end

end
```

Listing 3: OFAngleVarianceFeature class

```
classdef OFAngleVarianceFeature < AbstractFeature
    %OFANGLEVARIANCEFEATURE computes the variance of the flow vector angles
    %   in a small window (defined by nhood) around each pixel. The
    %   constructor takes a cell array of Flow objects which will be used
    %   for computing this feature. Second argument is of the nhood (a 5x5
    %   window [c r] = meshgrid(-2:2, -2:2); nhood = cat(3, r(:), c(:));).
    %   The constructor also optionally takes a size 2 vector for computing
    %   the feature on scalespace (first value: number of scales, second
    %   value: resizing factor). If using scalespace, ComputeFeatureVectors
    %   object passed to calcFeatures should have
    %   extra_info.flow_scalespace (the scalespace structure), apart from
    %   image_sz. Note that it is the responsibility of the user to provide
    %   enough number of scales in all scalespace structure. If not
    %   using scalespace, extra_info.calc_flows.uv_flows is required for
    %   computing this feature. If using the scalespace, usually, the
    %   output features go up in the scalespace (increasing gaussian
    %   std-dev) with increasing depth.
    %
    %   The features are first ordered by algorithms and then with their
    %   respective scale

```

```
    properties
        no_scales = 1;
        scale = 1;
        nhood;

        flow_ids = [];
        flow_short_types = {};
    end


    properties (Constant)
        FEATURE_TYPE = 'Angle Variance';
        FEATURE_SHORT_TYPE = 'AV';
    end


    methods
        function obj = OFAngleVarianceFeature( cell_flows, nhood, varargin )
            assert(~isempty(cell_flows), ['There should be atleast 1 flow
                algorithm to compute ' class(obj)]);

            % store the flow algorithms to be used and their ids
            for algo_idx = 1:length(cell_flows)
                obj.flow_short_types{end+1} = cell_flows{algo_idx}.
                    OF_SHORT_TYPE;
                obj.flow_ids(end+1) = cell_flows{algo_idx}.returnNoID();
            end

            % neighborhood window provided by user
            obj.nhood = nhood;

            % store any scalespace info provided by user
            if nargin > 2 && isvector(varargin{1}) && length(varargin{1}) ==
                2
                obj.no_scales = varargin{1}(1);
                obj.scale = varargin{1}(2);
            end
        end


        function [ angvar feature_depth ] = calcFeatures( obj,
            calc_feature_vec )
        % this function outputs the feature for this class, and the depth
        %   of this feature (number of unique features associated with this
        %   class). The size of angvar is the same as the input image,
```

```
    %   with a depth equivalent to the number of flow algos times the
    %   number of scales

        % find which algos to use
        algos_to_use = cellfun(@(x) find(strcmp(x, calc_feature_vec.extra_info.calc_flows.algo_ids)), obj.flow_short_types);

        assert(length(algos_to_use)==length(obj.flow_short_types), ['Can''t find matching flow algorithm(s) used in computation of ' class(obj)]);

        if obj.no_scales > 1
            assert(isfield(calc_feature_vec.extra_info, 'flow_scalespace') && ...
                ~isempty(fields(calc_feature_vec.extra_info.flow_scalespace)), ...
                'The scale space for UV flow has not been defined in the passed ComputeFeatureVectors');

            assert(calc_feature_vec.extra_info.flow_scalespace.scale == obj.scale && ...
                calc_feature_vec.extra_info.flow_scalespace.no_scales >= obj.no_scales, ...
                'The scale space given for UV flow in ComputeFeatureVectors is incompatible');

            % get the number of flow algorithms
            no_flow_algos = length(obj.flow_short_types);

            % initialize the output feature
            angvar = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos*obj.no_scales);

            % iterate for multiple scales
            for scale_idx = 1:obj.no_scales

                image_sz = size(calc_feature_vec.extra_info.flow_scalespace.ss{scale_idx});
                image_sz = image_sz([1 2]);

                % compute angle variance for each optical flow given
                angvar_temp = obj.computeAngVarForEachUV(calc_feature_vec.extra_info.flow_scalespace.ss{scale_idx}(:,:,:,algos_to_use), image_sz );

```

```
                    % iterate over all the candidate flow algorithms
                    for feat_idx = 1:size(angvar_temp,3)
                        % resize and store
                        angvar(:,:,((feat_idx-1)*obj.no_scales)+scale_idx) = imresize(angvar_temp(:,:,feat_idx), calc_feature_vec.image_sz);
                    end
                end
            else
                assert(isfield(calc_feature_vec.extra_info, 'calc_flows'), 'The CalcFlows object has not been defined in the passed ComputeFeatureVectors');

                % compute angle variance for each optical flow given
                angvar = obj.computeAngVarForEachUV( calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,:,algos_to_use), calc_feature_vec.image_sz );
            end

            feature_depth = size(angvar,3);
        end


        function feature_no_id = returnNoID(obj)
        % creates unique feature number, good for storing with the file
        % name

            % create unique ID
            nos = returnNoID@AbstractFeature(obj);

            temp = (obj.no_scales^obj.scale)*numel(obj.nhood);
            % get first 2 decimal digits
            temp = mod(round(temp*100), 100);
            feature_no_id = (nos*100) + temp;

            feature_no_id = feature_no_id + sum(obj.flow_ids);
        end


        function return_feature_list = returnFeatureList(obj)
        % creates a cell vector where each item contains a string of the
        % feature type (in the order the will be spit out by calcFeatures)

            window_size = num2str(max((max(obj.nhood,[],1) - min(obj.nhood,[],1))+1));
            return_feature_list = cell(obj.no_scales * length(obj.
```

```
                    flow_short_types),1);

            for flow_id = 1:length(obj.flow_short_types)
                starting_no = (flow_id-1)*obj.no_scales;

                return_feature_list{starting_no+1} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], ['window_size ' window_size], 'no_scaling'};

                for scale_id = 2:obj.no_scales
                    return_feature_list{starting_no+scale_id} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], ['window_size ' window_size], ['scale ' num2str(scale_id)], ['size ' sprintf('%.1f%%', (obj.scale^(scale_id-1))*100)]};
                end
            end
        end
    end


    methods (Access = private)
        function [ angvar ] = computeAngVarForEachUV( obj, uv_flows, image_sz )

            no_flow_algos = size(uv_flows, 4);

            % initialize the output feature
            angvar = zeros(image_sz(1), image_sz(2), no_flow_algos);

            % get the nhood r and c's (each col given a neighborhood
            %  around a pixel - nhood_r is row ind, nhood_c is col ind)
            [cols rows] = meshgrid(1:image_sz(2), 1:image_sz(1));
            nhood_rep = repmat(obj.nhood, [1 numel(rows) 1]);
            nhood_r = nhood_rep(:,:,1) + repmat(rows(:)', [size(obj.nhood,1) 1]);
            nhood_c = nhood_rep(:,:,2) + repmat(cols(:)', [size(obj.nhood,1) 1]);

            % get the pixel indices which are outside
            idxs_outside = nhood_r <= 0 | nhood_c <= 0 | nhood_r > image_sz(1) | nhood_c > image_sz(2);

            % find how many nhood pixels are outside for each pixel
            sums_outside = sum(idxs_outside, 1);

```

```
        % find the unique no. of nhood pixels outside (will iterate
        % over these no.s)
        unique_sums = unique(sums_outside);


        % iterate over all the candidate flow algorithms
        for algo_idx = 1:no_flow_algos

            % get the flow for this candidate algorithm
            xfl = uv_flows(:,:,1,algo_idx);
            yfl = uv_flows(:,:,2,algo_idx);

            % initialize the feature to return
            features = zeros(numel(xfl),1);

            % iterate over all unique no. of pixels outside
            for s = unique_sums
                % get the pixels which fall in this category
                curr_idxs = sums_outside==s;

                % get rows and cols for for these valid pixels
                temp_r = nhood_r(:,curr_idxs);
                temp_c = nhood_c(:,curr_idxs);

                % throw away indices which fall outside (fix temp_r
                % and temp_c)
                if s ~= 0
                    % select the pixel (nhoods) which have this no. of nhood pixels outside
                    temp_idxs_outside = idxs_outside(:,curr_idxs);

                    % sort and delete the nhood pixels which are outside
                    [temp, remaining_idxs_rs] = sort(temp_idxs_outside, 1);
                    remaining_idxs_rs(end-s+1:end,:) = [];

                    % adjust temp_r and temp_c with the indxs found which are not outside the image
                    remaining_idxs_rs = sub2ind(size(temp_r), remaining_idxs_rs, repmat(1:size(temp_c,2), [size(remaining_idxs_rs,1) 1]));
                    temp_r = temp_r(remaining_idxs_rs);
                    temp_c = temp_c(remaining_idxs_rs);
                end

                % get the indxs for each pixel nhood
```

```
                    temp_indxs = sub2ind(size(xfl), temp_r, temp_c);
                    temp_u = xfl(temp_indxs);
                    temp_v = yfl(temp_indxs);


                    %%% The main feature computation
                    ang = atan(temp_v ./ temp_u);
                    avg_ang = anglesUnwrappedMean( ang, 'rad', 1 );
                    avg_ang = repmat(avg_ang, [size(ang,1) 1]);
                    avg_ang = anglesUnwrappedDiff(ang, avg_ang);

                    % angle variance
                    avg_ang = mean(avg_ang.^2, 1);
                    features(curr_idxs,1) = avg_ang;
                end

                % store
                angvar(:,:,algo_idx) = reshape(features, image_sz);
            end
        end
    end

end
```

Listing 4: OFCollidingSpeedFeature class

```
classdef OFCollidingSpeedFeature < AbstractFeature
    %OFCOLLIDINGSPEEDEFEATURE computes the speed of collision given the
    %   flow vectors of diagonally opposite pixels in a  lengths in a small
    %   window (defined by nhood) around each pixel. This class computes
    %   summary statistics of the different collision speeds in a certain
    %   pixel nhood. The constructor takes a cell array of Flow objects
    %   which will be used for computing this feature. Second argument is
    %   of the nhood (a 5x5 window [c r] = meshgrid(-2:2, -2:2);
    %   nhood = cat(3, r(:), c(:));
    %   nhood_cs(nhood_cs(:,:,1)==0 & nhood_cs(:,:,2)==0,:,:) = [];).
    %   The constructor also optionally takes a size 2 vector for computing
    %   the feature on scalespace (first value: number of scales, second
    %   value: resizing factor). If using scalespace, ComputeFeatureVectors
    %   object passed to calcFeatures should have
    %   extra_info.flow_scalespace (the scalespace structure), apart from
    %   image_sz. Note that it is the responsibility of the user to provide
    %   enough number of scales in all scalespace structure. If not
    %   using scalespace, extra_info.calc_flows.uv_flows is required for
    %   computing this feature. If using the scalespace, usually, the
```

```
    %   output features go up in the scalespace (increasing gaussian
    %   std-dev) with increasing depth.
    %
    %   The features are first ordered by algorithms and then with max /
    %   min / var features and then by their respective scale


    properties
        no_scales = 1;
        scale = 1;
        nhood_1;
        nhood_2;

        flow_ids = [];
        flow_short_types = {};
    end


    properties (Transient)
        pinv_dist_u;
        pinv_dist_v;
        proj_a1;
        proj_a2;
        proj_a3;
        proj_a4;
    end


    properties (Constant)
        FEATURE_TYPE = 'Colliding Speed';
        FEATURE_SHORT_TYPE = 'CS';

        FEATURES_PER_PIXEL = 3;
        FEATURES_PER_PIXEL_TYPES = {'MAX', 'MIN', 'VAR'};
    end


    methods
        function obj = OFCollidingSpeedFeature( cell_flows, nhood, varargin
            )
            assert(~isempty(cell_flows), ['There should be atleast 1 flow
                algorithm to compute ' class(obj)]);

            % store the flow algorithms to be used and their ids
            for algo_idx = 1:length(cell_flows)
                obj.flow_short_types{end+1} = cell_flows{algo_idx}.
```

```
                OF_SHORT_TYPE;
            obj.flow_ids(end+1) = cell_flows{algo_idx}.returnNoID();
        end

        % neighborhood window provided by user
        assert(mod(sqrt(size(nhood,1)+1), 1) == 0, 'The number of nhood pixels can be only (Z^2)-1');
        obj.nhood_1 = nhood(1:size(nhood,1)/2,:,:);
        obj.nhood_2 = nhood(end:-1:(size(nhood,1)/2)+1,:,:);

        % initialize the other transient info required to compute this feature
        obj = obj.extraInfo();

        % store any scalespace info provided by user
        if nargin > 2 && isvector(varargin{1}) && length(varargin{1}) == 2
            obj.no_scales = varargin{1}(1);
            obj.scale = varargin{1}(2);
        end
    end


    function [ colspd feature_depth ] = calcFeatures( obj, calc_feature_vec )
    % this function outputs the feature for this class, and the depth
    %   of this feature (number of unique features associated with this
    %   class). The size of colspd is the same as the input image,
    %   with a depth equivalent to the number of flow algos times the
    %   features per pixel times the number of scales

        % find which algos to use
        algos_to_use = cellfun(@(x) find(strcmp(x, calc_feature_vec.extra_info.calc_flows.algo_ids)), obj.flow_short_types);

        assert(length(algos_to_use)==length(obj.flow_short_types), ['Can''t find matching flow algorithm(s) used in computation of ' class(obj)]);

        if obj.no_scales > 1
            assert(isfield(calc_feature_vec.extra_info, 'flow_scalespace') && ...
                ~isempty(fields(calc_feature_vec.extra_info.flow_scalespace)), ...
                'The scale space for UV flow has not been defined in the passed ComputeFeatureVectors');
```

```
            assert(calc_feature_vec.extra_info.flow_scalespace.scale == obj.scale && ...
                calc_feature_vec.extra_info.flow_scalespace.no_scales >= obj.no_scales, ...
                'The scale space given for UV flow in ComputeFeatureVectors is incompatible');

            % get the number of flow algorithms
            no_flow_algos = length(obj.flow_short_types);

            % initialize the output feature
            colspd = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos*obj.FEATURES_PER_PIXEL*obj.no_scales);

            % iterate for multiple scales
            for scale_idx = 1:obj.no_scales

                image_sz = size(calc_feature_vec.extra_info.flow_scalespace.ss{scale_idx});
                image_sz = image_sz([1 2]);

                % compute diagonally opposite pixel's colliding speed for each optical flow given
                colspd_temp = obj.computeCollidingSpeedForEachUV(calc_feature_vec.extra_info.flow_scalespace.ss{scale_idx}(:,:,:,algos_to_use), image_sz );

                % iterate over all the candidate flow algorithms
                for feat_idx = 1:size(colspd_temp,3)
                    % resize and store
                    colspd(:,:,((feat_idx-1)*obj.no_scales)+scale_idx) = imresize(colspd_temp(:,:,feat_idx), calc_feature_vec.image_sz);
                end
            end
        else
            assert(isfield(calc_feature_vec.extra_info, 'calc_flows'), 'The CalcFlows object has not been defined in the passed ComputeFeatureVectors');

            % compute diagonally opposite pixel's colliding speed for each optical flow given
            colspd = obj.computeCollidingSpeedForEachUV(calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,:,
```

```
                algos_to_use), calc_feature_vec.image_sz );
        end

        feature_depth = size(colspd,3);
    end


    function feature_no_id = returnNoID(obj)
    % creates unique feature number, good for storing with the file
    % name

        % create unique ID
        nos = returnNoID@AbstractFeature(obj);

        temp = (obj.no_scales^obj.scale)*numel(obj.nhood_1);
        % get first 2 decimal digits
        temp = mod(round(temp*100), 100);
        feature_no_id = (nos*100) + temp;

        feature_no_id = feature_no_id + sum(obj.flow_ids);

        for ftr_idx = 1:length(obj.FEATURES_PER_PIXEL_TYPES)
            nos = uint8(obj.FEATURES_PER_PIXEL_TYPES{ftr_idx});
            nos = double(nos) .* ([1:length(nos)].^2);
            feature_no_id = feature_no_id + sum(nos);
        end
    end


    function return_feature_list = returnFeatureList(obj)
    % creates a cell vector where each item contains a string of the
    % feature type (in the order the will be spit out by calcFeatures)

        window_size = num2str(max((max(obj.nhood_2,[],1) - min(obj.
            nhood_1,[],1))+1));
        return_feature_list = cell(obj.no_scales * obj.
            FEATURES_PER_PIXEL * length(obj.flow_short_types),1);

        for flow_id = 1:length(obj.flow_short_types)
            for feature_id = 1:obj.FEATURES_PER_PIXEL
                starting_no = ((flow_id-1)*obj.no_scales*obj.
                    FEATURES_PER_PIXEL) + ((feature_id-1)*obj.no_scales)
                    ;

                return_feature_list{starting_no+1} = {[obj.FEATURE_TYPE
                    ' using ' obj.flow_short_types{flow_id}], ...
```

```
                                        [obj.FEATURES_PER_PIXEL_TYPES{feature_id} ' feature'], ...
                                        ['window_size ' window_size], 'no_scaling'};

            for scale_id = 2:obj.no_scales
                return_feature_list{starting_no+scale_id} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], ...
                                        [obj.FEATURES_PER_PIXEL_T
                                        {feature_id} ' feature'], ...
                                        ['window size ' window_size], ['scale ' num2str(scale_id)], ...
                                        ['size ' sprintf('%.1f%%', (
```

```
                                                              obj
                                                              .
                                                              scale
                                                              ^(
                                                              scale_id
                                                              -1)
                                                              )
                                                              *100)
                                                              ]};
                end
            end
        end
    end
end


methods (Access = private)
    function [ colspd ] = computeCollidingSpeedForEachUV( obj, uv_flows,
        image_sz )

        no_flow_algos = size(uv_flows, 4);

        % initialize the output feature
        colspd = zeros(image_sz(1), image_sz(2), no_flow_algos*obj.
            FEATURES_PER_PIXEL);

        % get the nhood r and c's (each col given a neighborhood
        %  around a pixel - nhood_r is row ind, nhood_c is col ind)
        [cols rows] = meshgrid(1:image_sz(2), 1:image_sz(1));
        nhood_rep_1 = repmat(obj.nhood_1, [1 numel(rows) 1]);
        nhood_rep_2 = repmat(obj.nhood_2, [1 numel(rows) 1]);
        nhood_r_1 = nhood_rep_1(:,:,1) + repmat(rows(:)', [size(obj.
            nhood_1,1) 1]);
        nhood_c_1 = nhood_rep_1(:,:,2) + repmat(cols(:)', [size(obj.
            nhood_1,1) 1]);
        nhood_r_2 = nhood_rep_2(:,:,1) + repmat(rows(:)', [size(obj.
            nhood_2,1) 1]);
        nhood_c_2 = nhood_rep_2(:,:,2) + repmat(cols(:)', [size(obj.
            nhood_2,1) 1]);

        % get the pixel indices which are outside
        idxs_outside = nhood_r_1 <= 0 | nhood_c_1 <= 0 | nhood_r_1 >
            image_sz(1) | nhood_c_1 > image_sz(2) ...
             | nhood_r_2 <= 0 | nhood_c_2 <= 0 | nhood_r_2 > image_sz(1)
                 | nhood_c_2 > image_sz(2);

```

```
        % find how many nhood pixels are outside for each pixel
        sums_outside = sum(idxs_outside, 1);

        % find the unique no. of nhood pixels outside (will iterate
        % over these no.s)
        unique_sums = unique(sums_outside);

        % iterate over all the candidate flow algorithms
        for algo_idx = 1:no_flow_algos

            % get the flow for this candidate algorithm
            xfl = uv_flows(:,:,1,algo_idx);
            yfl = uv_flows(:,:,2,algo_idx);

            % initialize the feature to return
            features = zeros(numel(xfl), obj.FEATURES_PER_PIXEL);

            % iterate over all unique no. of pixels outside
            for s = unique_sums
                % get the pixels which fall in this category
                curr_idxs = sums_outside==s;

                % get rows and cols for for these valid pixels
                temp_r_1 = nhood_r_1(:,curr_idxs);
                temp_c_1 = nhood_c_1(:,curr_idxs);
                temp_r_2 = nhood_r_2(:,curr_idxs);
                temp_c_2 = nhood_c_2(:,curr_idxs);


                % throw away indices which fall outside (fix temp_r
                % and temp_c)
                if s ~= 0
                    % select the pixel (nhoods) which have this no. of nhood pixels outside
                    temp_idxs_outside = idxs_outside(:,curr_idxs);

                    % sort and delete the nhood pixels which are outside
                    [temp, remaining_idxs_rs] = sort(temp_idxs_outside, 1);
                    remaining_idxs_rs(end-s+1:end,:) = [];

                    % get the projection matrix A
                    curr_proj_a1 = obj.proj_a1(remaining_idxs_rs);
                    curr_proj_a2 = obj.proj_a2(remaining_idxs_rs);
                    curr_proj_a3 = obj.proj_a3(remaining_idxs_rs);
                    curr_proj_a4 = obj.proj_a4(remaining_idxs_rs);
```

```

            % get the pseudoinv distance
            curr_pinv_d_u = obj.pinv_dist_u(remaining_idxs_rs);
            curr_pinv_d_v = obj.pinv_dist_v(remaining_idxs_rs);

            % if its not a 2D array straighten the arrays
            if ~all(size(curr_proj_a1) == size(remaining_idxs_rs))
                curr_proj_a1 = curr_proj_a1';
                curr_proj_a2 = curr_proj_a2';
                curr_proj_a3 = curr_proj_a3';
                curr_proj_a4 = curr_proj_a4';

                curr_pinv_d_u = curr_pinv_d_u';
                curr_pinv_d_v = curr_pinv_d_v';
            end

            % adjust temp_r and temp_c with the indxs found which are not outside the image
            remaining_idxs_rs = sub2ind(size(temp_r_1), remaining_idxs_rs, repmat(1:size(temp_c_1,2), [size(remaining_idxs_rs,1) 1]));
            temp_r_1 = temp_r_1(remaining_idxs_rs);
            temp_c_1 = temp_c_1(remaining_idxs_rs);
            temp_r_2 = temp_r_2(remaining_idxs_rs);
            temp_c_2 = temp_c_2(remaining_idxs_rs);
        else
            % get the projection matrix A
            curr_proj_a1 = repmat(obj.proj_a1, [1 size(temp_r_1, 2)]);
            curr_proj_a2 = repmat(obj.proj_a2, [1 size(temp_r_1, 2)]);
            curr_proj_a3 = repmat(obj.proj_a3, [1 size(temp_r_1, 2)]);
            curr_proj_a4 = repmat(obj.proj_a4, [1 size(temp_r_1, 2)]);

            % get the pseudoinv distance
            curr_pinv_d_u = repmat(obj.pinv_dist_u, [1 size(temp_r_1, 2)]);
            curr_pinv_d_v = repmat(obj.pinv_dist_v, [1 size(temp_r_1, 2)]);
        end

        % get the indxs for each pixel nhood
        temp_indxs = sub2ind(size(xfl), temp_r_1, temp_c_1);
```

```
                    temp_u_1 = xfl(temp_indxs);
                    temp_v_1 = yfl(temp_indxs);
                    temp_indxs = sub2ind(size(xfl), temp_r_2, temp_c_2);
                    temp_u_2 = xfl(temp_indxs);
                    temp_v_2 = yfl(temp_indxs);


                    %%% The main feature computation
                    fu = temp_u_1 - temp_u_2;
                    fv = temp_v_1 - temp_v_2;

                    curr_proj_a1 = curr_proj_a1.*fu + curr_proj_a3.*fv;
                    curr_proj_a2 = curr_proj_a2.*fu + curr_proj_a4.*fv;

                    t = curr_pinv_d_u.*curr_proj_a1 + curr_pinv_d_v.*
                        curr_proj_a2;

                    if ~isempty(t)
                        features(curr_idxs, 1) = max(t, [], 1);
                        features(curr_idxs, 2) = min(t, [], 1);
                        features(curr_idxs, 3) = var(t, 1, 1);
                    end
                end

                % store
                for feat_idx = 1:obj.FEATURES_PER_PIXEL
                    colspd(:,:,((algo_idx-1)*obj.FEATURES_PER_PIXEL)+
                        feat_idx) = reshape(features(:,feat_idx), image_sz);
                end
            end
        end


        function obj = extraInfo( obj )
            dist = squeeze(obj.nhood_2 - obj.nhood_1);

            n = sum(dist.^2, 2);

            obj.proj_a1 = ( dist(:,1).^2 ) ./ n;
            obj.proj_a2 = ( dist(:,1).*dist(:,2) ) ./ n;
            obj.proj_a3 = obj.proj_a2;
            obj.proj_a4 = ( dist(:,2).^2 ) ./ n;

            obj.pinv_dist_u = 1./n .* dist(:,1);
            obj.pinv_dist_v = 1./n .* dist(:,2);
        end
```

```
    end

end
```

Listing 5: OFLengthVarianceFeature class

```
classdef OFLengthVarianceFeature < AbstractFeature
    %OFLENGTHVARIANCEFEATURE computes the variance of the flow vector
    %   lengths in a small window (defined by nhood) around each pixel. The
    %   constructor takes a cell array of Flow objects which will be used
    %   for computing this feature. Second argument is of the nhood (a 5x5
    %   window [c r] = meshgrid(-2:2, -2:2); nhood = cat(3, r(:), c(:));).
    %   The constructor also optionally takes a size 2 vector for computing
    %   the feature on scalespace (first value: number of scales, second
    %   value: resizing factor). If using scalespace, ComputeFeatureVectors
    %   object passed to calcFeatures should have
    %   extra_info.flow_scalespace (the scalespace structure), apart from
    %   image_sz. Note that it is the responsibility of the user to provide
    %   enough number of scales in all scalespace structure. If not
    %   using scalespace, extra_info.calc_flows.uv_flows is required for
    %   computing this feature. If using the scalespace, usually, the
    %   output features go up in the scalespace (increasing gaussian
    %   std-dev) with increasing depth.
    %
    %   The features are first ordered by algorithms and then with their
    %   respective scale


    properties
        no_scales = 1;
        scale = 1;
        nhood;

        flow_ids = [];
        flow_short_types = {};
    end


    properties (Constant)
        FEATURE_TYPE = 'Length_Variance';
        FEATURE_SHORT_TYPE = 'LV';
    end


    methods
```

```
        function obj = OFLengthVarianceFeature( cell_flows, nhood, varargin )
            assert(~isempty(cell_flows), ['There should be atleast 1 flow algorithm to compute ' class(obj)]);

            % store the flow algorithms to be used and their ids
            for algo_idx = 1:length(cell_flows)
                obj.flow_short_types{end+1} = cell_flows{algo_idx}.OF_SHORT_TYPE;
                obj.flow_ids(end+1) = cell_flows{algo_idx}.returnNoID();
            end

            % neighborhood window provided by user
            obj.nhood = nhood;

            % store any scalespace info provided by user
            if nargin > 2 && isvector(varargin{1}) && length(varargin{1}) == 2
                obj.no_scales = varargin{1}(1);
                obj.scale = varargin{1}(2);
            end
        end


        function [ lenvar feature_depth ] = calcFeatures( obj, calc_feature_vec )
        % this function outputs the feature for this class, and the depth
        %   of this feature (number of unique features associated with this
        %   class). The size of lenvar is the same as the input image,
        %   with a depth equivalent to the number of flow algos times the
        %   number of scales

            % find which algos to use
            algos_to_use = cellfun(@(x) find(strcmp(x, calc_feature_vec.extra_info.calc_flows.algo_ids)), obj.flow_short_types);

            assert(length(algos_to_use)==length(obj.flow_short_types), ['Can''t find matching flow algorithm(s) used in computation of ' class(obj)]);

            if obj.no_scales > 1
                assert(isfield(calc_feature_vec.extra_info, 'flow_scalespace') && ...
                    ~isempty(fields(calc_feature_vec.extra_info.flow_scalespace)), ...
                    'The scale space for UV flow has not been defined in the
```

```
                    passed ComputeFeatureVectors');

            assert(calc_feature_vec.extra_info.flow_scalespace.scale ==
                obj.scale && ...
                calc_feature_vec.extra_info.flow_scalespace.no_scales >=
                    obj.no_scales, ...
                'The scale space given for UV flow in
                    ComputeFeatureVectors is incompatible');

            % get the number of flow algorithms
            no_flow_algos = length(obj.flow_short_types);

            % initialize the output feature
            lenvar = zeros(calc_feature_vec.image_sz(1),
                calc_feature_vec.image_sz(2), no_flow_algos*obj.
                no_scales);

            % iterate for multiple scales
            for scale_idx = 1:obj.no_scales

                image_sz = size(calc_feature_vec.extra_info.
                    flow_scalespace.ss{scale_idx});
                image_sz = image_sz([1 2]);

                % compute length variance for each optical flow given
                lenvar_temp = obj.computeLenVarForEachUV(
                    calc_feature_vec.extra_info.flow_scalespace.ss{
                    scale_idx}(:,:,:,algos_to_use), image_sz );

                % iterate over all the candidate flow algorithms
                for feat_idx = 1:size(lenvar_temp,3)
                    % resize and store
                    lenvar(:,:,((feat_idx-1)*obj.no_scales)+scale_idx) =
                        imresize(lenvar_temp(:,:,feat_idx),
                        calc_feature_vec.image_sz);
                end
            end
        else
            assert(isfield(calc_feature_vec.extra_info, 'calc_flows'), '
                The CalcFlows object has not been defined in the passed
                ComputeFeatureVectors');

            % compute length variance for each optical flow given
            lenvar = obj.computeLenVarForEachUV( calc_feature_vec.
                extra_info.calc_flows.uv_flows(:,:,:,algos_to_use),
                calc_feature_vec.image_sz );
```

```
            end

            feature_depth = size(lenvar,3);
        end


        function feature_no_id = returnNoID(obj)
        % creates unique feature number, good for storing with the file
        % name

            % create unique ID
            nos = returnNoID@AbstractFeature(obj);

            temp = (obj.no_scales^obj.scale)*numel(obj.nhood);
            % get first 2 decimal digits
            temp = mod(round(temp*100), 100);
            feature_no_id = (nos*100) + temp;

            feature_no_id = feature_no_id + sum(obj.flow_ids);
        end


        function return_feature_list = returnFeatureList(obj)
        % creates a cell vector where each item contains a string of the
        % feature type (in the order the will be spit out by calcFeatures)

            window_size = num2str(max((max(obj.nhood,[],1) - min(obj.nhood
                ,[],1))+1));
            return_feature_list = cell(obj.no_scales * length(obj.
                flow_short_types),1);

            for flow_id = 1:length(obj.flow_short_types)
                starting_no = (flow_id-1)*obj.no_scales;

                return_feature_list{starting_no+1} = {[obj.FEATURE_TYPE ' 
                    using ' obj.flow_short_types{flow_id}], ['window_size '
                    window_size], 'no_scaling'};

                for scale_id = 2:obj.no_scales
                    return_feature_list{starting_no+scale_id} = {[obj.
                        FEATURE_TYPE ' using ' obj.flow_short_types{flow_id
                        }], ['window_size ' window_size], ['scale ' num2str(
                        scale_id)], ['size ' sprintf('%.1f%%', (obj.scale^(
                        scale_id-1))*100)]};
                end
            end
```

```
        end
    end


    methods (Access = private)
        function [ lenvar ] = computeLenVarForEachUV( obj, uv_flows,
            image_sz )

            no_flow_algos = size(uv_flows, 4);

            % initialize the output feature
            lenvar = zeros(image_sz(1), image_sz(2), no_flow_algos);

            % get the nhood r and c's (each col given a neighborhood
            %  around a pixel - nhood_r is row ind, nhood_c is col ind)
            [cols rows] = meshgrid(1:image_sz(2), 1:image_sz(1));
            nhood_rep = repmat(obj.nhood, [1 numel(rows) 1]);
            nhood_r = nhood_rep(:,:,1) + repmat(rows(:)', [size(obj.nhood,1)
                 1]);
            nhood_c = nhood_rep(:,:,2) + repmat(cols(:)', [size(obj.nhood,1)
                 1]);

            % get the pixel indices which are outside
            idxs_outside = nhood_r <= 0 | nhood_c <= 0 | nhood_r > image_sz
                (1) | nhood_c > image_sz(2);

            % find how many nhood pixels are outside for each pixel
            sums_outside = sum(idxs_outside, 1);

            % find the unique no. of nhood pixels outside (will iterate
            % over these no.s)
            unique_sums = unique(sums_outside);


            % iterate over all the candidate flow algorithms
            for algo_idx = 1:no_flow_algos

                % get the flow for this candidate algorithm
                xfl = uv_flows(:,:,1,algo_idx);
                yfl = uv_flows(:,:,2,algo_idx);

                % initialize the feature to return
                features = zeros(numel(xfl),1);

                % iterate over all unique no. of pixels outside
                for s = unique_sums
```

```
                % get the pixels which fall in this category
                curr_idxs = sums_outside==s;

                % get rows and cols for for these valid pixels
                temp_r = nhood_r(:, curr_idxs);
                temp_c = nhood_c(:, curr_idxs);

                % throw away indices which fall outside (fix temp_r
                % and temp_c)
                if s ~= 0
                    % select the pixel (nhoods) which have this no. of
                        nhood pixels outside
                    temp_idxs_outside = idxs_outside(:, curr_idxs);

                    % sort and delete the nhood pixels which are outside
                    [temp, remaining_idxs_rs] = sort(temp_idxs_outside,
                        1);
                    remaining_idxs_rs(end-s+1:end,:) = [];

                    % adjust temp_r and temp_c with the indxs found
                        which are not outside the image
                    remaining_idxs_rs = sub2ind(size(temp_r),
                        remaining_idxs_rs, repmat(1:size(temp_c,2), [
                        size(remaining_idxs_rs,1) 1]));
                    temp_r = temp_r(remaining_idxs_rs);
                    temp_c = temp_c(remaining_idxs_rs);
                end

                % get the indxs for each pixel nhood
                temp_indxs = sub2ind(size(xfl), temp_r, temp_c);
                temp_u = xfl(temp_indxs);
                temp_v = yfl(temp_indxs);


                %%% The main feature computation
                % length variance
                len = sqrt(temp_u.^2 + temp_v.^2);
                mean_len = repmat(mean(len, 1), [size(len,1) 1]);
                len_var = mean((len - mean_len).^2, 1);
                features(curr_idxs,1) = len_var;
            end

            % store
            lenvar(:,:,algo_idx) = reshape(features, image_sz);
        end
    end
```

```
    end
end
```

Listing 6: PbEdgeStrengthFeature class

```
classdef PbEdgeStrengthFeature < AbstractFeature
    %PBEDGESTRENGTHFEATURE Summary of this class goes here
    %   Detailed explanation goes here

    properties
        threshold_pb;

        no_scales = 1;
        scale = 1;
    end


    properties (Constant)
        PRECOMPUTED_PB_FILE = 'pb.mat';

        FEATURE_TYPE = 'Pb Edge Strength';
        FEATURE_SHORT_TYPE = 'PB';
    end


    methods
        function obj = PbEdgeStrengthFeature( threshold, varargin )
            % threshold for Pb provided by user
            obj.threshold_pb = threshold;

            if nargin > 1 && isvector(varargin{2}) && length(varargin{2}) == 2
                obj.no_scales = varargin{2}(1);
                obj.scale = varargin{2}(2);
            end
        end


        function [ pbedge feature_depth ] = calcFeatures( obj, calc_feature_vec )
            CalcFlows.addPaths()

            if obj.no_scales > 1
                error('PbEdgeStrengthFeature:NoScaleSpace', 'Scale space not supported yet');
```

```
            assert(~isempty(fields(calc_feature_vec.im1_scalespace)), 'The scale space for im 1 has not been defined in the passed ComputeFeatureVectors');

            assert(calc_feature_vec.im1_scalespace.scale == obj.scale && ...
                calc_feature_vec.im1_scalespace.no_scales >= obj.no_scales, 'The scale space given for im 1 in ComputeFeatureVectors is incompatible');

            % initialize the output feature
            pbedge = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), obj.no_scales);

            % iterate for multiple scales
            for scale_idx = 1:obj.no_scales
                % get the next image in the scale space
                im_resized = calc_feature_vec.im1_scalespace.ss{scale_idx};

                % compute the probability of boundary
                if size(calc_feature_vec.im1,3) == 1
                    [ pbedge ] = pbBGTG(im2double(im_resized));
                else
                    [ pbedge ] = pbCGTG(im2double(im_resized));
                end

                % compute distance transform and resize it to the original image size
                pbedge = imresize(bwdist(pbedge > obj.threshold_pb), calc_feature_vec.image_sz);

                % resize it to the original image size
                pbedge(:,:,scale_idx) = imresize(pb, calc_feature_vec.image_sz);
            end
        else
            % if precomputed pb exists
            if exist(fullfile(calc_feature_vec.scene_dir, obj.PRECOMPUTED_PB_FILE), 'file') == 2
                load(fullfile(calc_feature_vec.scene_dir, obj.PRECOMPUTED_PB_FILE));
            else
                % compute the probability of boundary
                if size(calc_feature_vec.im1,3) == 1
                    [ pbedge ] = pbBGTG(im2double(calc_feature_vec.im1))
```

```
                ;
            else
                [ pbedge ] = pbCGTG(im2double(calc_feature_vec.im1))
                ;
            end
        end

        % compute distance transform and resize it to the original
            image size
        pbedge = imresize(double(bwdist(pbedge > obj.threshold_pb)),
            calc_feature_vec.image_sz);
    end

    feature_depth = size(pbedge,3);
end


function feature_no_id = returnNoID(obj)
% creates unique feature number, good for storing with the file
% name

    % create unique ID
    nos = returnNoID@AbstractFeature(obj);

    temp = obj.no_scales^obj.scale;
    % get first 2 decimal digits
    temp = mod(round(temp*100), 100);
    feature_no_id = (nos*100) + temp;

    % incorporate the threshold
    feature_no_id = round(obj.threshold_pb * feature_no_id);
end
end

end
```

Listing 7: PhotoConstancyFeature class

```
classdef PhotoConstancyFeature < AbstractFeature
    %PHOTOCONSTANCYFEATURE the |I1(x)-I2(x+u)| the absolute difference in
    %   pixel values of two images using the flow information. The
    %   constructor takes a cell array of Flow objects which will be used
    %   for computing this feature. The constructor also optionally takes a
    %   size 2 vector for computing the feature on scalespace (first value:
    %   number of scales, second value: resizing factor). If using
```

```
    %   scalespace, ComputeFeatureVectors object passed to calcFeatures
    %   should have im1_scalespace, im2_scalespace and
    %   extra_info.flow_scalespace (the scalespace structures), apart from
    %   image_sz. Note that it is the responsibility of the user to provide
    %   enough number of scales in all 3 scalespace structures. If not
    %   using scalespace im1_gray, im2_gray and
    %   extra_info.calc_flows.uv_flows are required for computing this
    %   feature. If using the scalespace, usually, the output features go
    %   up in the scalespace (increasing gaussian std-dev) with increasing
    %   depth.
    %
    %   The features are first ordered by algorithms and then with their
    %   respective scale


    properties
        no_scales = 1;
        scale = 1;

        flow_ids = [];
        flow_short_types = {};
    end


    properties (Constant)
        NAN_VAL = 100;
        FEATURE_TYPE = 'Photo Constancy';
        FEATURE_SHORT_TYPE = 'PC';
    end


    methods
        function obj = PhotoConstancyFeature( cell_flows, varargin )
            assert(~isempty(cell_flows), ['There should be atleast 1 flow
                algorithm to compute ' class(obj)]);

            % store the flow algorithms to be used and their ids
            for algo_idx = 1:length(cell_flows)
                obj.flow_short_types{end+1} = cell_flows{algo_idx}.
                    OF_SHORT_TYPE;
                obj.flow_ids(end+1) = cell_flows{algo_idx}.returnNoID();
            end

            % store any scalespace info provided by user
            if nargin > 1 && isvector(varargin{1}) && length(varargin{1}) ==
                2
```

```
                obj.no_scales = varargin{1}(1);
                obj.scale = varargin{1}(2);
            end
        end


        function [ photoconst feature_depth ] = calcFeatures( obj,
            calc_feature_vec )
        % this function outputs the feature for this class, and the depth
        %   of this feature (number of unique features associated with this
        %   class). The size of photoconst is the same as the input image,
        %   with a depth equivalent to the number of flow algos times the
        %   number of scales

            if obj.no_scales > 1
                assert(~isempty(fields(calc_feature_vec.im1_scalespace)) &&
                    ...
                    ~isempty(fields(calc_feature_vec.im2_scalespace)), ...
                    'The scale space for im 1 and/or im 2 has not been
                        defined in the passed ComputeFeatureVectors');

                assert(calc_feature_vec.im1_scalespace.scale == obj.scale &&
                    ...
                    calc_feature_vec.im1_scalespace.no_scales >= obj.
                        no_scales, 'The scale space given for im 1 in
                        ComputeFeatureVectors is incompatible');

                assert(calc_feature_vec.im2_scalespace.scale == obj.scale &&
                    ...
                    calc_feature_vec.im2_scalespace.no_scales >= obj.
                        no_scales, 'The scale space given for im 2 in
                        ComputeFeatureVectors is incompatible');

                assert(isfield(calc_feature_vec.extra_info, 'flow_scalespace
                    ') && ...
                    ~isempty(fields(calc_feature_vec.extra_info.
                        flow_scalespace)), ...
                    'The scale space for UV flow has not been defined in the
                        passed ComputeFeatureVectors');

                assert(calc_feature_vec.extra_info.flow_scalespace.scale ==
                    obj.scale && ...
                    calc_feature_vec.extra_info.flow_scalespace.no_scales >=
                        obj.no_scales, ...
                    'The scale space given for UV flow in
                        ComputeFeatureVectors is incompatible');
```

```
            no_flow_algos = length(obj.flow_short_types);

            % initialize the output feature
            photoconst = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos*obj.no_scales);

            % iterate for multiple scales
            for scale_idx = 1:obj.no_scales
                % get the next image in the scale space
                im1_resized = calc_feature_vec.im1_scalespace.ss{scale_idx};
                im2_resized = calc_feature_vec.im2_scalespace.ss{scale_idx};

                [cols rows] = meshgrid(1:size(im1_resized, 2), 1:size(im1_resized, 1));

                % iterate over all the candidate flow algorithms
                for algo_idx = 1:no_flow_algos
                    algo_id = strcmp(obj.flow_short_types{algo_idx}, calc_feature_vec.extra_info.calc_flows.algo_ids);

                    assert(nnz(algo_id) == 1, ['Can''t find matching flow algorithm used in computation of ' class(obj)]);

                    % get the next flow image in the scale space
                    uv_resized = calc_feature_vec.extra_info.flow_scalespace.ss{scale_idx}(:,:,:,algo_id);

                    % project the second image to the first according to the flow
                    proj_im = interp2(im2_resized, cols + uv_resized(:,:,1), rows + uv_resized(:,:,2), 'cubic');

                    % compute the error in the projection
                    proj_im = abs(im1_resized - proj_im);
                    proj_im(isnan(proj_im)) = PhotoConstancyFeature.NAN_VAL;

                    % store
                    photoconst(:,:,((algo_idx-1)*obj.no_scales)+
```

```
                                scale_idx) = imresize(proj_im, calc_feature_vec.
                                image_sz);
                        end
                    end
                else
                    assert(isfield(calc_feature_vec.extra_info, 'calc_flows'), '
                        The CalcFlows object has not been defined in the passed 
                        ComputeFeatureVectors');

                    no_flow_algos = length(obj.flow_short_types);

                    % initialize the output feature
                    photoconst = zeros(calc_feature_vec.image_sz(1),
                        calc_feature_vec.image_sz(2), no_flow_algos);

                    [cols rows] = meshgrid(1:calc_feature_vec.image_sz(2), 1:
                        calc_feature_vec.image_sz(1));

                    % iterate over all the candidate flow algorithms
                    for algo_idx = 1:no_flow_algos
                        algo_id = strcmp(obj.flow_short_types{algo_idx},
                            calc_feature_vec.extra_info.calc_flows.algo_ids);

                        assert(nnz(algo_id) == 1, ['Can''t find matching flow 
                            algorithm used in computation of ' class(obj)]);

                        % project the second image to the first according to the
                             flow
                        proj_im = interp2(calc_feature_vec.im2_gray, ...
                            cols + calc_feature_vec.extra_info.calc_flows.
                                uv_flows(:,:,1,algo_id), ...
                            rows + calc_feature_vec.extra_info.calc_flows.
                                uv_flows(:,:,2,algo_id), 'cubic');

                        % compute the error in the projection
                        proj_im = abs(calc_feature_vec.im1_gray - proj_im);
                        proj_im(isnan(proj_im)) = PhotoConstancyFeature.NAN_VAL;

                        % store
                        photoconst(:,:,algo_idx) = proj_im;
                    end
                end

                feature_depth = size(photoconst,3);
            end

```

```

        function feature_no_id = returnNoID(obj)
        % creates unique feature number, good for storing with the file
        % name

            % create unique ID
            nos = returnNoID@AbstractFeature(obj);

            temp = obj.no_scales^obj.scale;
            % get first 2 decimal digits
            temp = mod(round(temp*100), 100);
            feature_no_id = (nos*100) + temp;

            feature_no_id = feature_no_id + sum(obj.flow_ids);
        end


        function return_feature_list = returnFeatureList(obj)
        % creates a cell vector where each item contains a string of the
        % feature type (in the order the will be spit out by calcFeatures)

            return_feature_list = cell(obj.no_scales * length(obj.flow_short_types),1);

            for flow_id = 1:length(obj.flow_short_types)
                starting_no = (flow_id-1)*obj.no_scales;

                return_feature_list{starting_no+1} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], 'no_scaling'};

                for scale_id = 2:obj.no_scales
                    return_feature_list{starting_no+scale_id} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], ['scale ' num2str(scale_id)], ['size ' sprintf('%.1f%%', (obj.scale^(scale_id-1))*100)]};
                end
            end
        end
    end

end
```

Listing 8: ReverseFlowAngleDiffFeature class

```
classdef ReverseFlowAngleDiffFeature < AbstractFeature
```

```
    %REVERSEFLOWCONSTANCYFEATURE computes:
    %   x' = round(x + u_{12}(x))
    %   \theta = \pi - acos(u_{12}(x).u_{21}(x'))
    %   which in short is the angle difference between the forward vector
    %   and the reverse vector (from the advected position). The
    %   constructor takes a cell array of Flow objects which will be used
    %   for computing this feature. The constructor also optionally takes a
    %   size 2 vector for computing the feature on scalespace (first value:
    %   number of scales, second value: resizing factor). If using
    %   scalespace, ComputeFeatureVectors object passed to calcFeatures
    %   should have extra_info.flow_scalespace (the flow scalespace
    %   structures) and extra_info.flow_scalespace_r (the reverse flow
    %   scalespace structures), apart from image_sz. Note that it is the
    %   responsibility of the user to provide enough number of scales in
    %   both the scalespace structures. If not using scalespace,
    %   extra_info.calc_flows.uv_flows and
    %   extra_info.calc_flows.uv_flows_reverse are required for computing
    %   this feature. If using the scalespace, usually, the output features
    %   go up in the scalespace (increasing gaussian std-dev) with
    %   increasing depth.
    %
    %   The features are first ordered by algorithms and then with their
    %   respective scale



    properties
        no_scales = 1;
        scale = 1;

        flow_ids = [];
        flow_short_types = {};
    end


    properties (Constant)
        NAN_VAL = pi;
        FEATURE_TYPE = 'Reverse Flow Angle Difference';
        FEATURE_SHORT_TYPE = 'RA';
    end


    methods
        function obj = ReverseFlowAngleDiffFeature( cell_flows, varargin )
            assert(~isempty(cell_flows), ['There should be atleast 1 flow
                algorithm to compute ' class(obj)]);
```

```

            % store the flow algorithms to be used and their ids
            for algo_idx = 1:length(cell_flows)
                obj.flow_short_types{end+1} = cell_flows{algo_idx}.OF_SHORT_TYPE;
                obj.flow_ids(end+1) = cell_flows{algo_idx}.returnNoID();
            end

            % store any scalespace info provided by user
            if nargin > 1 && isvector(varargin{1}) && length(varargin{1}) == 2
                obj.no_scales = varargin{1}(1);
                obj.scale = varargin{1}(2);
            end
        end


        function [ revangdiff feature_depth ] = calcFeatures( obj, calc_feature_vec )
        % this function outputs the feature for this class, and the depth
        %   of this feature (number of unique features associated with this
        %   class). The size of revangdiff is the same as the input image,
        %   with a depth equivalent to the number of flow algos times the
        %   number of scales

            if obj.no_scales > 1
                assert(isfield(calc_feature_vec.extra_info, 'flow_scalespace') && ...
                    ~isempty(fields(calc_feature_vec.extra_info.flow_scalespace)) && ...
                    ~isempty(fields(calc_feature_vec.extra_info.flow_scalespace_r)), ...
                    'The scale space for UV flow (or/and its reverse) has not been defined in the passed ComputeFeatureVectors');

                assert(calc_feature_vec.extra_info.flow_scalespace.scale == obj.scale && ...
                    calc_feature_vec.extra_info.flow_scalespace.no_scales >= obj.no_scales && ...
                    calc_feature_vec.extra_info.flow_scalespace_r.scale == obj.scale && ...
                    calc_feature_vec.extra_info.flow_scalespace_r.no_scales >= obj.no_scales, ...
                    'The scale space given for UV flow (or/and its reverse) in ComputeFeatureVectors is incompatible');
```

```
        no_flow_algos = length(obj.flow_short_types);

        % initialize the output feature
        revangdiff = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos*obj.no_scales);

        % iterate for multiple scales
        for scale_idx = 1:obj.no_scales
            im_sz = size(calc_feature_vec.extra_info.flow_scalespace_r.ss{scale_idx}(:,:,1,1));

            [cols rows] = meshgrid(1:im_sz(2), 1:im_sz(1));

            % iterate over all the candidate flow algorithms
            for algo_idx = 1:no_flow_algos
                algo_id = strcmp(obj.flow_short_types{algo_idx}, calc_feature_vec.extra_info.calc_flows.algo_ids);

                assert(nnz(algo_id) == 1, ['Can''t find matching flow algorithm used in computation of ' class(obj)]);

                % get the next flow image in the scale space
                uv_resized = calc_feature_vec.extra_info.flow_scalespace.ss{scale_idx}(:,:,:,algo_id);
                uv_resized_reverse = calc_feature_vec.extra_info.flow_scalespace_r.ss{scale_idx}(:,:,:,algo_id);

                % compute x' = round(x + u_{12}(x)) (advected point)
                r_dash = rows + uv_resized(:,:,2);
                c_dash = cols + uv_resized(:,:,1);
                r_dash = round(r_dash);
                c_dash = round(c_dash);

                % find the points which have fallen outside the image
                outside_idcs = r_dash < 1 | r_dash > im_sz(1) | c_dash < 1 | c_dash > im_sz(2);
                r_dash(outside_idcs) = 1;
                c_dash(outside_idcs) = 1;

                ind_dash = sub2ind(im_sz, r_dash, c_dash);
```

```

                    % normalize uv vector
                    norm_val = hypot(uv_resized(:,:,1), uv_resized(:,:,2));
                    u_n = uv_resized(:,:,1) ./ norm_val;
                    v_n = uv_resized(:,:,2) ./ norm_val;

                    % get the reverse flow
                    rev_v = uv_resized_reverse(:,:,2);
                    rev_u = uv_resized_reverse(:,:,1);

                    % normalize uv reverse vector
                    norm_val = hypot(rev_u(ind_dash), rev_v(ind_dash));
                    rev_u_n = rev_u(ind_dash) ./ norm_val;
                    rev_v_n = rev_v(ind_dash) ./ norm_val;

                    % compute u_{12}(x).u_{21}(x')
                    temp = (rev_v_n.*v_n) + (rev_u_n.*u_n);
                    ang_diff = pi - acos(temp);

                    ang_diff(outside_idcs) = ReverseFlowAngleDiffFeature.NAN_VAL;

                    % store
                    revangdiff(:,:,((algo_idx-1)*obj.no_scales)+scale_idx) = imresize(real(ang_diff), calc_feature_vec.image_sz);
                end
            end
        else
            assert(isfield(calc_feature_vec.extra_info, 'calc_flows'), 'The CalcFlows object has not been defined in the passed ComputeFeatureVectors');
            assert(~isempty(calc_feature_vec.extra_info.calc_flows.uv_flows_reverse), 'The reverse flow in CalcFlows object has not been defined in the passed ComputeFeatureVectors');

            no_flow_algos = length(obj.flow_short_types);

            % initialize the output feature
            revangdiff = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos);

            [cols rows] = meshgrid(1:calc_feature_vec.image_sz(2), 1:calc_feature_vec.image_sz(1));
```

```

        % iterate over all the candidate flow algorithms
        for algo_idx = 1:no_flow_algos
            algo_id = strcmp(obj.flow_short_types{algo_idx}, calc_feature_vec.extra_info.calc_flows.algo_ids);

            assert(nnz(algo_id) == 1, ['Can''t find matching flow algorithm used in computation of ' class(obj)]);

            % compute x' = round(x + u_{12}(x)) (advected point)
            r_dash = rows + calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,2,algo_id);
            c_dash = cols + calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,1,algo_id);
            r_dash = round(r_dash);
            c_dash = round(c_dash);

            % find the points which have fallen outside the image
            outside_idcs = r_dash < 1 | r_dash > calc_feature_vec.image_sz(1) | c_dash < 1 | c_dash > calc_feature_vec.image_sz(2);
            r_dash(outside_idcs) = 1;
            c_dash(outside_idcs) = 1;

            ind_dash = sub2ind(calc_feature_vec.image_sz, r_dash, c_dash);

            % normalize uv vector
            norm_val = hypot(calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,1,algo_id), calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,2,algo_id));
            u_n = calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,1,algo_id) ./ norm_val;
            v_n = calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,2,algo_id) ./ norm_val;

            % get the reverse flow
            rev_v = calc_feature_vec.extra_info.calc_flows.uv_flows_reverse(:,:,2,algo_id);
            rev_u = calc_feature_vec.extra_info.calc_flows.uv_flows_reverse(:,:,1,algo_id);

            % normalize uv reverse vector
            norm_val = hypot(rev_u(ind_dash), rev_v(ind_dash));
            rev_u_n = rev_u(ind_dash) ./ norm_val;
            rev_v_n = rev_v(ind_dash) ./ norm_val;
```

```

                % compute u_{12}(x).u_{21}(x')
                temp = (rev_v_n.*v_n) + (rev_u_n.*u_n);
                ang_diff = pi - acos(temp);

                ang_diff(outside_idcs) = ReverseFlowAngleDiffFeature.NAN_VAL;

                % store
                revangdiff(:,:,algo_idx) = real(ang_diff);
            end
        end

        feature_depth = size(revangdiff,3);
    end


    function feature_no_id = returnNoID(obj)
    % creates unique feature number, good for storing with the file
    % name

        % create unique ID
        nos = returnNoID@AbstractFeature(obj);

        temp = obj.no_scales^obj.scale;
        % get first 2 decimal digits
        temp = mod(round(temp*100), 100);
        feature_no_id = (nos*100) + temp;

        feature_no_id = feature_no_id + sum(obj.flow_ids);
    end


    function return_feature_list = returnFeatureList(obj)
    % creates a cell vector where each item contains a string of the
    % feature type (in the order the will be spit out by calcFeatures)

        return_feature_list = cell(obj.no_scales * length(obj.flow_short_types),1);

        for flow_id = 1:length(obj.flow_short_types)
            starting_no = (flow_id-1)*obj.no_scales;

            return_feature_list{starting_no+1} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], 'no_scaling'};

```

```
                for scale_id = 2:obj.no_scales
                    return_feature_list{starting_no+scale_id} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], ['scale ' num2str(scale_id)], ['size ' sprintf('%.1f%%', (obj.scale^(scale_id-1))*100)]};
                end
            end
        end
    end

end
```

Listing 9: ReverseFlowConstancyFeature class

```
classdef ReverseFlowConstancyFeature < AbstractFeature
    %REVERSEFLOWCONSTANCYFEATURE computes:
    %   x' = round(x + u_{12}(x))
    %   ||x - (x' + u_{21}(x'))||
    %   which in short is the distance between the original position and
    %   the position advected by the forward flow followed by the reverse
    %   flow. The constructor takes a cell array of Flow objects which will
    %   be used for computing this feature. The constructor also optionally
    %   takes a size 2 vector for computing the feature on scalespace
    %   (first value: number of scales, second value: resizing factor). If
    %   using scalespace, ComputeFeatureVectors object passed to
    %   calcFeatures should have extra_info.flow_scalespace (the flow
    %   scalespace structures) and extra_info.flow_scalespace_r (the
    %   reverse flow scalespace structures), apart from image_sz. Note that
    %   it is the responsibility of the user to provide enough number of
    %   scales in both the scalespace structures. If not using scalespace,
    %   extra_info.calc_flows.uv_flows and
    %   extra_info.calc_flows.uv_flows_reverse are required for computing
    %   this feature. If using the scalespace, usually, the output features
    %   go up in the scalespace (increasing gaussian std-dev) with
    %   increasing depth.
    %
    %   The features are first ordered by algorithms and then with their
    %   respective scale


    properties
        no_scales = 1;
        scale = 1;

        flow_ids = [];
```

```
            flow_short_types = {};
        end


        properties (Constant)
            NAN_VAL = 10;
            FEATURE_TYPE = 'Reverse Flow Constancy';
            FEATURE_SHORT_TYPE = 'RC';
        end


        methods
            function obj = ReverseFlowConstancyFeature( cell_flows, varargin )
                assert(~isempty(cell_flows), ['There should be atleast 1 flow algorithm to compute ' class(obj)]);

                % store the flow algorithms to be used and their ids
                for algo_idx = 1:length(cell_flows)
                    obj.flow_short_types{end+1} = cell_flows{algo_idx}.OF_SHORT_TYPE;
                    obj.flow_ids(end+1) = cell_flows{algo_idx}.returnNoID();
                end

                % store any scalespace info provided by user
                if nargin > 1 && isvector(varargin{1}) && length(varargin{1}) == 2
                    obj.no_scales = varargin{1}(1);
                    obj.scale = varargin{1}(2);
                end
            end


            function [ revflowconst feature_depth ] = calcFeatures( obj, calc_feature_vec )
            % this function outputs the feature for this class, and the depth
            %   of this feature (number of unique features associated with this
            %   class). The size of revflowconst is the same as the input image,
            %   with a depth equivalent to the number of flow algos times the
            %   number of scales

                if obj.no_scales > 1
                    assert(isfield(calc_feature_vec.extra_info, 'flow_scalespace') && ...
                        ~isempty(fields(calc_feature_vec.extra_info.flow_scalespace)) && ...
                        ~isempty(fields(calc_feature_vec.extra_info.
```

```
                flow_scalespace_r)), ...
            'The scale space for UV flow (or/and its reverse) has not been defined in the passed ComputeFeatureVectors');

        assert(calc_feature_vec.extra_info.flow_scalespace.scale == obj.scale && ...
            calc_feature_vec.extra_info.flow_scalespace.no_scales >= obj.no_scales && ...
            calc_feature_vec.extra_info.flow_scalespace_r.scale == obj.scale && ...
            calc_feature_vec.extra_info.flow_scalespace_r.no_scales >= obj.no_scales, ...
            'The scale space given for UV flow (or/and its reverse) in ComputeFeatureVectors is incompatible');


        no_flow_algos = length(obj.flow_short_types);

        % initialize the output feature
        revflowconst = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos*obj.no_scales);

        % iterate for multiple scales
        for scale_idx = 1:obj.no_scales
            im_sz = size(calc_feature_vec.extra_info.flow_scalespace_r.ss{scale_idx}(:,:,1,1));

            [cols rows] = meshgrid(1:im_sz(2), 1:im_sz(1));

            % iterate over all the candidate flow algorithms
            for algo_idx = 1:no_flow_algos
                algo_id = strcmp(obj.flow_short_types{algo_idx}, calc_feature_vec.extra_info.calc_flows.algo_ids);

                assert(nnz(algo_id) == 1, ['Can''t find matching flow algorithm used in computation of ' class(obj)]);

                % get the next flow image in the scale space
                uv_resized = calc_feature_vec.extra_info.flow_scalespace.ss{scale_idx}(:,:,:,algo_id);
                uv_resized_reverse = calc_feature_vec.extra_info.flow_scalespace_r.ss{scale_idx}(:,:,:,algo_id);
```

```

                % compute x' = round(x + u_{12}(x)) (advected point)
                r_dash = rows + uv_resized(:,:,2);
                c_dash = cols + uv_resized(:,:,1);
                r_dash = round(r_dash);
                c_dash = round(c_dash);

                % find the points which have fallen outside the image
                outside_idcs = r_dash < 1 | r_dash > im_sz(1) | c_dash < 1 | c_dash > im_sz(2);
                r_dash(outside_idcs) = 1;
                c_dash(outside_idcs) = 1;

                % get the reverse flow
                rev_v = uv_resized_reverse(:,:,2);
                rev_u = uv_resized_reverse(:,:,1);

                %  compute ||x - (x' + u_{21}(x'))||
                ind_dash = sub2ind(im_sz, r_dash, c_dash);

                r_dash = r_dash + rev_v(ind_dash);
                c_dash = c_dash + rev_u(ind_dash);

                reverse_dist = sqrt((rows - r_dash).^2 + (cols - c_dash).^2);
                reverse_dist(outside_idcs) = ReverseFlowConstancyFeature.NAN_VAL;

                % store
                revflowconst(:,:,((algo_idx-1)*obj.no_scales)+scale_idx) = imresize(reverse_dist, calc_feature_vec.image_sz);
            end
        end
    else
        assert(isfield(calc_feature_vec.extra_info, 'calc_flows'), 'The CalcFlows object has not been defined in the passed ComputeFeatureVectors');
        assert(~isempty(calc_feature_vec.extra_info.calc_flows.uv_flows_reverse), 'The reverse flow in CalcFlows object has not been defined in the passed ComputeFeatureVectors');

        no_flow_algos = length(obj.flow_short_types);

```

```
        % initialize the output feature
        revflowconst = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos);

        [cols rows] = meshgrid(1:calc_feature_vec.image_sz(2), 1:calc_feature_vec.image_sz(1));

        % iterate over all the candidate flow algorithms
        for algo_idx = 1:no_flow_algos
            algo_id = strcmp(obj.flow_short_types{algo_idx}, calc_feature_vec.extra_info.calc_flows.algo_ids);

            assert(nnz(algo_id) == 1, ['Can''t find matching flow algorithm used in computation of ' class(obj)]);

            % compute x' = round(x + u_{12}(x)) (advected point)
            r_dash = rows + calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,2,algo_id);
            c_dash = cols + calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,1,algo_id);
            r_dash = round(r_dash);
            c_dash = round(c_dash);

            % find the points which have fallen outside the image
            outside_idcs = r_dash < 1 | r_dash > calc_feature_vec.image_sz(1) | c_dash < 1 | c_dash > calc_feature_vec.image_sz(2);
            r_dash(outside_idcs) = 1;
            c_dash(outside_idcs) = 1;

            % get the reverse flow
            rev_v = calc_feature_vec.extra_info.calc_flows.uv_flows_reverse(:,:,2,algo_id);
            rev_u = calc_feature_vec.extra_info.calc_flows.uv_flows_reverse(:,:,1,algo_id);

            %  compute ||x - (x' + u_{21}(x'))||
            ind_dash = sub2ind(calc_feature_vec.image_sz, r_dash, c_dash);

            r_dash = r_dash + rev_v(ind_dash);
            c_dash = c_dash + rev_u(ind_dash);

            reverse_dist = sqrt((rows - r_dash).^2 + (cols - c_dash).^2);
            reverse_dist(outside_idcs) = ReverseFlowConstancyFeature
```

```
                        .NAN_VAL;

                    % store
                    revflowconst(:,:,algo_idx) = reverse_dist;
                end
            end

            feature_depth = size(revflowconst,3);
        end


        function feature_no_id = returnNoID(obj)
        % creates unique feature number, good for storing with the file
        % name

            % create unique ID
            nos = returnNoID@AbstractFeature(obj);

            temp = obj.no_scales^obj.scale;
            % get first 2 decimal digits
            temp = mod(round(temp*100), 100);
            feature_no_id = (nos*100) + temp;

            feature_no_id = feature_no_id + sum(obj.flow_ids);
        end


        function return_feature_list = returnFeatureList(obj)
        % creates a cell vector where each item contains a string of the
        % feature type (in the order the will be spit out by calcFeatures)

            return_feature_list = cell(obj.no_scales * length(obj.flow_short_types),1);

            for flow_id = 1:length(obj.flow_short_types)
                starting_no = (flow_id-1)*obj.no_scales;

                return_feature_list{starting_no+1} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], 'no_scaling'};

                for scale_id = 2:obj.no_scales
                    return_feature_list{starting_no+scale_id} = {[obj.FEATURE_TYPE ' using ' obj.flow_short_types{flow_id}], ['scale ' num2str(scale_id)], ['size ' sprintf('%.1f%%', (obj.scale^(scale_id-1))*100)]};
                end
```

```
            end
        end
    end

end
```

Listing 10: SparseSetTextureFeature class

```
classdef SparseSetTextureFeature < AbstractFeature
    %SPARSESETTEXTUREFEATURE computes the difference in texture given by
    %   Sparse Set of Texture features as proposed in:
    %         Brox, T., From pixels to regions: partial differential
    %         equations in image analysis, April 2005
    %   Given the advected position of each pixel x' = round(x +
    %   u_{12}(x)), it computes the mahalanobis distance between T1(x) and
    %   T2(x'), where T1 is the texture feature for frame 1, and T2 is for
    %   frame 2. The constructor takes a cell array of Flow objects which
    %   will be used for computing this feature. The constructor also
    %   optionally takes a size 2 vector for computing the feature on
    %   scalespace (first value: number of scales, second value: resizing
    %   factor). If using scalespace, ComputeFeatureVectors object passed
    %   to calcFeatures should have im1_scalespace, im2_scalespace and
    %   extra_info.flow_scalespace (the scalespace structures), apart from
    %   image_sz. Note that it is the responsibility of the user to provide
    %   enough number of scales in all 3 scalespace structures. If not
    %   using scalespace im1_gray, im2_gray and
    %   extra_info.calc_flows.uv_flows are required for computing this
    %   feature. If using the scalespace, usually, the output features go
    %   up in the scalespace (increasing gaussian std-dev) with increasing
    %   depth.
    %
    %   The features are first ordered by flow algorithms and then with
    %   their respective scale


    properties
        no_scales = 1;
        scale = 1;

        flow_ids = [];
        flow_short_types = {};
    end


    properties (Constant)
```

```
        PRECOMPUTED_ST_FILE = 'sparsetextures.mat';

        NAN_VAL = 100;
        FEATURE_TYPE = 'Sparse Set Texture Difference (Mahalanobis)';
        FEATURE_SHORT_TYPE = 'STm';
    end


    methods
        function obj = SparseSetTextureFeature( cell_flows, varargin )
            assert(~isempty(cell_flows), ['There should be atleast 1 flow algorithm to compute ' class(obj)]);

            % store the flow algorithms to be used and their ids
            for algo_idx = 1:length(cell_flows)
                obj.flow_short_types{end+1} = cell_flows{algo_idx}.OF_SHORT_TYPE;
                obj.flow_ids(end+1) = cell_flows{algo_idx}.returnNoID();
            end

            % store any scalespace info provided by user
            if nargin > 1 && isvector(varargin{1}) && length(varargin{1}) == 2
                obj.no_scales = varargin{1}(1);
                obj.scale = varargin{1}(2);
            end
        end


        function [ texturediff feature_depth ] = calcFeatures( obj, calc_feature_vec )
        % this function outputs the feature for this class, and the depth
        %   of this feature (number of unique features associated with this
        %   class). The size of texturediff is the same as the input image,
        %   with a depth equivalent to the number of flow algos times the
        %   number of scales

            if obj.no_scales > 1
                assert(~isempty(fields(calc_feature_vec.im1_scalespace)) && ...
                    ~isempty(fields(calc_feature_vec.im2_scalespace)), ...
                    'The scale space for im 1 and/or im 2 has not been defined in the passed ComputeFeatureVectors');

                assert(calc_feature_vec.im1_scalespace.scale == obj.scale && ...
```

```
            calc_feature_vec.im1_scalespace.no_scales >= obj.no_scales, 'The scale space given for im 1 in ComputeFeatureVectors is incompatible');

        assert(calc_feature_vec.im2_scalespace.scale == obj.scale && ...
            calc_feature_vec.im2_scalespace.no_scales >= obj.no_scales, 'The scale space given for im 2 in ComputeFeatureVectors is incompatible');

        assert(isfield(calc_feature_vec.extra_info, 'flow_scalespace') && ...
            ~isempty(fields(calc_feature_vec.extra_info.flow_scalespace)), ...
            'The scale space for UV flow has not been defined in the passed ComputeFeatureVectors');

        assert(calc_feature_vec.extra_info.flow_scalespace.scale == obj.scale && ...
            calc_feature_vec.extra_info.flow_scalespace.no_scales >= obj.no_scales, ...
            'The scale space given for UV flow in ComputeFeatureVectors is incompatible');


        no_flow_algos = length(obj.flow_short_types);

        % initialize the output feature
        texturediff = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos*obj.no_scales);

        % iterate for multiple scales
        for scale_idx = 1:obj.no_scales
            % get the next image in the scale space
            im1_resized = calc_feature_vec.im1_scalespace.ss{scale_idx};
            im2_resized = calc_feature_vec.im2_scalespace.ss{scale_idx};

            % compute sparse set of texture features for both images
            sparsesettext1 = obj.computeSparseSetTexture( im1_resized );
            sparsesettext2 = obj.computeSparseSetTexture( im2_resized );

```

```
        [cols rows] = meshgrid(1:size(im1_resized, 2), 1:size(im1_resized, 1));

        % iterate over all the candidate flow algorithms
        for algo_idx = 1:no_flow_algos
            algo_id = strcmp(obj.flow_short_types{algo_idx}, calc_feature_vec.extra_info.calc_flows.algo_ids);

            assert(nnz(algo_id) == 1, ['Can''t find matching flow algorithm used in computation of ' class(obj)]);

            % get the next flow image in the scale space
            uv_resized = calc_feature_vec.extra_info.flow_scalespace.ss{scale_idx}(:,:,:,algo_id);

            proj_texture = zeros(size(sparsesettext2));
            texture_var = zeros(1,1,size(sparsesettext2,3));

            for text_idx = 1:size(proj_texture,3)
                % project the second image's texture feature to the first according to the flow
                proj_texture(:,:,text_idx) = interp2(sparsesettext2(:,:,text_idx), ...
                    cols + uv_resized(:,:,1), ...
                    rows + uv_resized(:,:,2), 'cubic');

                % compute variance of each feature
                temp = [sparsesettext1(:,:,text_idx) sparsesettext2(:,:,text_idx)];
                texture_var(text_idx) = var(temp(:));
            end

            texture_var = repmat(texture_var, [size(im1_resized,1) size(im1_resized,2)]);

            % compute the Mahalanobis distance for the texture features
            proj_texture = (sparsesettext1 - proj_texture).^2;
            proj_texture = sqrt(sum(proj_texture ./ texture_var, 3));

            proj_texture(isnan(proj_texture)) = SparseSetTextureFeature.NAN_VAL;

```

```
                    % store
                    texturediff(:,:,((algo_idx-1)*obj.no_scales)+scale_idx) = imresize(proj_texture, calc_feature_vec.image_sz);
                end
            end
        else
            assert(isfield(calc_feature_vec.extra_info, 'calc_flows'), 'The CalcFlows object has not been defined in the passed ComputeFeatureVectors');

            no_flow_algos = length(obj.flow_short_types);

            % if precomputed pb exists
            if exist(fullfile(calc_feature_vec.scene_dir, obj.PRECOMPUTED_ST_FILE), 'file') == 2
                load(fullfile(calc_feature_vec.scene_dir, obj.PRECOMPUTED_ST_FILE));
                sparsesettext1 = T1;
                sparsesettext2 = T2;
            else
                % compute sparse set of texture features for both images
                sparsesettext1 = obj.computeSparseSetTexture(calc_feature_vec.im1);
                sparsesettext2 = obj.computeSparseSetTexture(calc_feature_vec.im2);
                T1 = sparsesettext1;
                T2 = sparsesettext2;
                save(fullfile(calc_feature_vec.scene_dir, obj.PRECOMPUTED_ST_FILE), 'T1', 'T2');
            end

            % initialize the output feature
            texturediff = zeros(calc_feature_vec.image_sz(1), calc_feature_vec.image_sz(2), no_flow_algos);

            [cols rows] = meshgrid(1:calc_feature_vec.image_sz(2), 1:calc_feature_vec.image_sz(1));

            % iterate over all the candidate flow algorithms
            for algo_idx = 1:no_flow_algos
                algo_id = strcmp(obj.flow_short_types{algo_idx}, calc_feature_vec.extra_info.calc_flows.algo_ids);

                assert(nnz(algo_id) == 1, ['Can''t find matching flow algorithm used in computation of ' class(obj)]);
```

```
                    proj_texture = zeros(size(sparsesettext2));
                    texture_var = zeros(1,1,size(sparsesettext2,3));

                    for text_idx = 1:size(proj_texture,3)
                        % project the second image's texture feature to the first according to the flow
                        proj_texture(:,:,text_idx) = interp2(sparsesettext2(:,:,text_idx), ...
                            cols + calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,1,algo_id), ...
                            rows + calc_feature_vec.extra_info.calc_flows.uv_flows(:,:,2,algo_id), 'cubic');

                        % compute variance of each feature
                        temp = [sparsesettext1(:,:,text_idx) sparsesettext2(:,:,text_idx)];
                        texture_var(text_idx) = var(temp(:));
                    end

                    texture_var = repmat(texture_var, calc_feature_vec.image_sz);

                    % compute the Mahalanobis distance for the texture features
                    proj_texture = (sparsesettext1 - proj_texture).^2;
                    proj_texture = sqrt(sum(proj_texture ./ texture_var, 3));

                    proj_texture(isnan(proj_texture)) = SparseSetTextureFeature.NAN_VAL;

                    % store
                    texturediff(:,:,algo_idx) = proj_texture;
                end
            end

            feature_depth = size(texturediff,3);
        end


        function feature_no_id = returnNoID(obj)
        % creates unique feature number, good for storing with the file
        % name

```

```
            % create unique ID
            nos = returnNoID@AbstractFeature(obj);

            temp = obj.no_scales^obj.scale;
            % get first 2 decimal digits
            temp = mod(round(temp*100), 100);
            feature_no_id = (nos*100) + temp;

            feature_no_id = feature_no_id + sum(obj.flow_ids);
        end


        function return_feature_list = returnFeatureList(obj)
        % creates a cell vector where each item contains a string of the
        % feature type (in the order the will be spit out by calcFeatures)

            return_feature_list = cell(obj.no_scales * length(obj.
                flow_short_types),1);

            for flow_id = 1:length(obj.flow_short_types)
                starting_no = (flow_id-1)*obj.no_scales;

                return_feature_list{starting_no+1} = {[obj.FEATURE_TYPE ' 
                    using ' obj.flow_short_types{flow_id}], 'no scaling'};

                for scale_id = 2:obj.no_scales
                    return_feature_list{starting_no+scale_id} = {[obj.
                        FEATURE_TYPE ' using ' obj.flow_short_types{flow_id
                        }], ['scale ' num2str(scale_id)], ['size ' sprintf('
                        %.1f%%', (obj.scale^(scale_id-1))*100)]};
                end
            end
        end
    end


    methods (Access = private)
        function sparsesettext = computeSparseSetTexture( obj, im )
            if size(im,3) == 3
                F = discriminative_texture_feature(double(im),6,[],1);
            else
                F = discriminative_texture_feature(double(im),6,[],0);
            end

            sparsesettext = reshape(F', [size(im,1),size(im,2),size(F,1)]);
        end
```

```
    end

end
```

The remaining code is in the accompanying DVD

Bibliography

[1] S. Baker, D. Scharstein, J.P. Lewis, S. Roth, M.J. Black, and R. Szeliski. A database and Evaluation Methodology for Optical Flow. Technical Report MSR-TR-2009-179, Microsoft Research, Redmond, WA, dec 2009. v, vii, 28, 35, 36, 45, 49, 50, 55, 56

[2] J.L. Barron, D.J. Fleet, and S.S. Beauchemin. Performance of optical flow techniques. *International Journal of Computer Vision*, 12(1):43–77, 1994. 49, 50

[3] J. Bergen, P. Anandan, K. Hanna, and R. Hingorani. Hierarchical model-based motion estimation. In *Computer Vision ECCV '92*, pages 237–252. Springer, 1992. 11

[4] M. Black. Recursive non-linear estimation of discontinuous flow fields. *Computer Vision ECCV '94*, pages 138–145, 1994. 8

[5] M.J. Black and P. Anandan. The robust estimation of multiple motions: Parametric and piecewise-smooth flow fields. *Computer Vision and Image Understanding*, 63(1):75–104, 1996. 28

[6] MJ Black and AD Jepson. Estimating optical flow in segmented images using variable-order parametric models with local deformations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(10):972–986, 1996. 8

[7] G. Bradski and A. Kaehler. *Learning OpenCV*. O'Reilly Media Inc., 2008. URL `http://oreilly.com/catalog/9780596516130`. 18, 23

[8] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996. 21

[9] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. 14, 17, 20, 21, 22

[10] L. Breiman and A. Cutler. Random Forests. `http://http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm`. 23

[11] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and Regression Trees.* Chapman and Hall/CRC, Monterey, CA, 1984. ISBN 0412048418. 17, 19, 20

[12] P. Brodatz. *Textures; a Photographic Album for Artists and Designers.* Dover Publications New York, 1966. ISBN 0486216691. v, 26

[13] T. Brox. *From pixels to regions: partial differential equations in image analysis.* PhD thesis, Faculty of Mathematics and Computer Science, Saarland University, Germany, April 2005. v, 26

[14] T. Brox and J. Malik. Large Displacement Optical Flow: Descriptor Matching in Variational Motion Estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 99, 2010. 28

[15] M.M. Chang, A.M. Tekalp, and M.I. Sezan. Simultaneous motion estimation and segmentation. *IEEE Transactions on Image Processing*, 6(9):1326 –1333, sep 1997. 6, 8

[16] M. Collins, R.E. Schapire, and Y. Singer. Logistic regression, AdaBoost and Bregman distances. *Machine Learning*, 48(1):253–285, 2002. 5

[17] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. 23

[18] Y. Freund and R. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995. 24

[19] M. Galun, A. Apartsin, and R. Basri. Multiscale segmentation by combining motion and intensity cues. In *Proceedings of the 2005 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '05)*, volume 1, pages 256–263, june 2005. 7, 9, 10

[20] B. Gillam. Shape and meaning in the perception of occlusion. *Journal of Vision*, 7(9):608, 2007. 1

[21] C. Goutte. Note on Free Lunches and Cross-Validation. *Neural Computation*, 9:1245–1249, 1997. 34

[22] S.E. Grigorescu, N. Petkov, and P. Kruizinga. Comparison of texture features based on Gabor filters. *IEEE Transactions on Image Processing*, 11(10):1160–1167, 2002. 25

[23] F. Heitz and P. Bouthemy. Multimodal estimation of discontinuous optical flow using Markov random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(12):1217–1232, 1993. 6

[24] B.K.P. Horn and B.G. Schunck. Determining optical flow. *Artificial intelligence*, 17(1-3):185–203, 1981. 6, 28

[25] M. Irani and P. Anandan. Factorization with uncertainty. *Computer Vision ECCV '00*, pages 539–553, 2000. 10

[26] M.P. Kumar, P.H.S. Torr, and A. Zisserman. Learning layered motion segmentations of video. *International Journal of Computer Vision*, 76(3):301–319, 2008. 11

[27] C. Liu, W.T. Freeman, E.H. Adelson, and Y. Weiss. Human-assisted motion annotation. In *Proceedings of the 2008 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '08)*, pages 1–8. IEEE Computer Society, 2008. vii, 56, 57

[28] W.-Y. Loh. Classification and regression tree methods. In *Encyclopedia of Statistics in Quality and Reliability*, pages 315–323. Wiley, Chichester, UK, 2008. URL `http://www.stat.wisc.edu/~loh/treeprogs/guide/eqr.pdf`. 17, 19

[29] O. Mac Aodha, G.J. Brostow, and M. Pollefeys. Segmenting Video Into Classes of Algorithm-Suitability. In *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '10)*, pages 1054–1061, june 2010. vii, 5, 14, 23, 24, 28, 31, 35, 36, 45, 51, 52

[30] S. Markovich. Amodal completion in visual perception. In *Visual Mathematics*, volume 4, 2002. 1

[31] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A Database of Human Segmented Natural Images and its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics. In *Proceedings of the Eighth IEEE International Conference on Computer Vision, (ICCV '01)*, volume 2, pages 416–423, July 2001. 32

[32] D.R. Martin, C.C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5):530–549, 2004. 12

[33] D.R. Martin, C.C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5):530–549, 2004. 32

[34] R. Megret and D. DeMenthon. A Survey of Spatio-Temporal Grouping Techniques. Technical Report CS-TR-4403, Language and Media Processing, University of Maryland, College Park, MD, August 2002. 10

[35] M. Muja and D.G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Applications (VISSAPP '09)*, volume 331–340, 2009. 12

[36] A.S. Ogale and Y. Aloimonos. A roadmap to the integration of early visual modules. *International Journal of Computer Vision*, 72(1):9–25, 2007. 1, 4

[37] A.S. Ogale and Y. Aloimonos. A roadmap to the integration of early visual modules. *International Journal of Computer Vision*, 72(1):9–25, 2007. 47

[38] A.S. Ogale, C. Fermuller, and Y. Aloimonos. Motion segmentation using occlusions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(6):988–992, 2005. 1, 11

[39] N. Paragios and R. Deriche. Geodesic active regions for motion estimation and tracking. In *Proceedings of the Seventh IEEE International Conference on Computer Vision, (ICCV '99)*, volume 1, pages 688–694, 1999. 9

[40] N. Paragios and R. Deriche. Geodesic Active Regions: A New Framework to Deal with Frame Partition Problems in Computer Vision. *Journal of Visual Communication and Image Representation*, 13(1-2):249–268, 2002. 9

[41] V.C. Raykar, S. Yu, L.H. Zhao, A. Jerebko, C. Florin, G.H. Valadez, L. Bogoni, and L. Moy. Supervised Learning from Multiple Experts: Whom to trust when everyone lies a bit. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 889–896. ACM, 2009. 14

[42] Ronald A. Rensink and James T. Enns B. Early completion of occluded objects. *Vision Research*, 38:2489–2505, 1998. 1

[43] M.G. Ross. Exploiting Texture-Motion Duality in Optical Flow and Image Segmentation. Master's thesis, Massachusetts Institute of Technology, Aug 2000. 2

[44] D. Scharstein and R. Szeliski. High-accuracy stereo depth maps using structured light. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 1:195, 2003. vii, 55, 56

[45] E. Sharon, A. Brandt, and R. Basri. Segmentation and boundary detection using multiscale intensity measurements. In *Proceedings of the 2001 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '01)*, volume 1, pages 469–476, 2001. 10

[46] J. Shi and J. Malik. Motion segmentation and tracking using normalized cuts. In *Proceedings of the Sixth IEEE International Conference on Computer Vision, (ICCV '98)*, pages 1154–1160, jan 1998. 3, 7, 9

[47] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000. 9

[48] J. Shi and C. Tomasi. Good features to track. In *Proceedings of the 1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '94)*, pages 593–600, 1994. 15

[49] C. Stauffer and W.E.L. Grimson. Learning patterns of activity using real-time tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):747–757, 2000. 10

[50] A.N. Stein and M. Hebert. Occlusion Boundaries from Motion: Low-Level Detection and Mid-Level Reasoning. *International Journal of Computer Vision*, 82(3):325–357, 2009. iii, iv, viii, 5, 12, 13, 51, 53, 54, 55

[51] B. Stenger, T. Woodley, and R. Cipolla. Learning to track with multiple observers. In *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '09)*, pages 2647–2654, june 2009. 14

[52] C. Strecha, R. Fransens, and L. Van Gool. A probabilistic approach to large displacement optical flow and occlusion detection. *Statistical methods in video processing*, pages 71–82, 2004. 1, 47

[53] C. Strecha, R. Fransens, and L. Van Gool. Combined Depth and Outlier Estimation in Multi-View Stereo. In *Proceedings of the 2006 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '06)*, volume 2, pages 2394–2401, 2006. 1, 47

[54] C. Strecha, A.M. Bronstein, M.M. Bronstein, and P. Fua. LDAHash: Improved matching with smaller descriptors. Technical report, Ecole Polytechnique Fedrale de Lausanne, 2010. iv, 4

[55] D. Sun, S. Roth, and M.J. Black. Secrets of optical flow estimation and their principles. In *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '10)*, pages 2432–2439, june 2010. 28

[56] E. Tola, V. Lepetit, and P. Fua. Daisy: an Efficient Dense Descriptor Applied to Wide Baseline Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(5):815–830, May 2010. 1, 47

[57] J.Y.A. Wang and E.H. Adelson. Representing moving images with layers. *IEEE Transactions on Image Processing*, 3(5):625–638, Sept. 1994. 11

[58] A. Wedel, T. Pock, C. Zach, H. Bischof, and D. Cremers. An improved algorithm for TV-L1 optical flow. *Statistical and Geometrical Approaches to Visual Motion Analysis*, pages 23–45, 2009. 28

[59] M. Werlberger, W. Trobin, T. Pock, A. Wedel, D. Cremers, and H. Bischof. Anisotropic Huber-L1 Optical Flow. In *Proceedings of the British Machine Vision Conference (BMVC)*, September 2009. 28

[60] R. Williams, Lance R. Williams, Edward M. Riseman, Steven W. Zucker, W. Richards Adrion, and Department Head. Perceptual completion of occluded surfaces. *Computer Vision and Image Understanding*, 64:1–20, 1994. 1

[61] K.Y. Wong and M.E. Spetsakis. Motion segmentation and tracking. In *Proceedings of 15th International Conference on Vision Interface*, pages 80–87, 2002. 7, 9

[62] J. Xiao and M. Shah. Motion layer extraction in the presence of occlusion using graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1644–1659, 2005. 12

[63] Q. Yang, L. Wang, R. Yang, H. Stewenius, and D. Nister. Stereo Matching with Color-Weighted Correlation, Hierarchical Belief Propagation, and Occlusion Handling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(3):492–504, mar. 2009. 47

[64] X. Yong, D. Feng, Z. Rongchun, and M. Petrou. Learning-based algorithm selection for image segmentation. *Pattern Recognition Letters*, 26(8):1059–1068, 2005. 14

[65] L. Zelnik-Manor, M. Machline, and M. Irani. Multi-body factorization with uncertainty: Revisiting motion consistency. *International Journal of Computer Vision*, 68(1):27–41, 2006. 10

[66] C.L. Zitnick, S.B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski. High-quality video view interpolation using a layered representation. *ACM Transactions on Graphics*, 23(3):600–608, 2004. 11

[67] C.L. Zitnick, S.B. Kang, and N. Jojic. Simultaneous optical flow estimation and image segmentation. US Patent 7,522,749, Apr 2009. Filed: July 2005. 11

[68] C.W. Zitnick, N. Jojic, and Sing Bing Kang. Consistent segmentation for optical flow estimation. In *Proceedings of the Tenth IEEE International Conference on Computer Vision, (ICCV '05)*, volume 2, pages 1308–1315, oct 2005. vii, 11, 55, 56