Training, Evaluation and Local Adaptation in Deformable Models

Samuel D. Fenster

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2000

ABSTRACT

Training, Evaluation and Local Adaptation in Deformable Models

Samuel D. Fenster

We describe how to teach deformable models to maximize image segmentation correctness based on user-specified criteria, and we present a method for evaluating which criteria work best. We present sectored snakes, which use local learning to improve demonstrably upon traditional snakes, and those with spatially uniform training, in abdominal CT slices and echocardiograms.

A traditional deformable model ("snake" in 2D) fails to find an object's boundary when the strongest nearby image edges are not the ones sought. But we show how to instead learn, from training data, the relation between the shape and any image feature, as the probability distribution (PDF) of a function of image and shape.

An important but neglected task for the implementor has always been to select image qualities to guide a model. Because success depends on the relation of objective function (PDF) output to shape correctness, it is evaluated using a sampling of ground truth, a random model of the range of shapes tried during optimization, and a measure of shape closeness. We measure the incidence of "false positives" (shapes scoring better than ground truth) within given distances from the correct one; and we measure the extent to which the function increases with shape incorrectness, using correlation coefficient and using distance to the nearest increasing function (isotonic regression residual).

We demonstrate such evaluation on a simple "sectoring" of a snake, in which intensity and perpendicular gradient are learned separately over equal-length segments. This specific set of qualities shows a measured improvement over an objective function that is uniform around the shape, and it follows naturally from examination of the latter's failures due to consistent image nonuniformity around the organ boundary.

Contents

List of Figures

List of Tables

Acknowledgments

My deepest appreciation to John Kender and Terry Boult, my advisors. I would also like to thank Drs. Radhe Mohan and Chen Chui of the Medical Physics Computer Services Department at Memorial Sloan-Kettering Cancer Center, and Dr. Jeffrey Weisman of EchoVision in Philadelphia, for their expertise and their image and contour data.

This work was supported in part by DOD/ONR MURI Grant N00014-95-1-0601, by the New York State Science and Technology Foundation's Center for Advanced Technology, and by ARPA Contract DACA-76-92-C-007.

Chapter 1

Introduction

This work specifies and tests a way of training a machine to find outlines of a known kind of object in images from a domain of consistent appearance. It uses deformable models (or *snakes*, *active contours* or *energy-minimizing splines*), which are a very flexible method of finding such outlines. Such training extends the applicability of the method by adapting it where it would be unable to work otherwise. We develop methods to assess how well a model, trained or fixed *a priori*, can perform in a given domain. This allows the investigator to select the best of many possible formulations of the model. We assess some sample candidate models, and find that training can provide a marked improvement.

Why segment images?

We often want a machine to accomplish a task based on visual information. To use information about physical structures in the world, it must locate them in an image or images.

There are many domains where the exact shapes of objects need to be discovered from images. These include industrial and medical applications where robots need to touch or grasp things, avoid them or inspect them. Computers assist in surgery planning; they optimize medical treatments in which beams of radiation must precisely target some organs and avoid others; and they have diagnostic applications in medicine and manufacturing where volume, curvature or motion must be measured to detect abnormalities. Knowledge of precise shape also aids in the

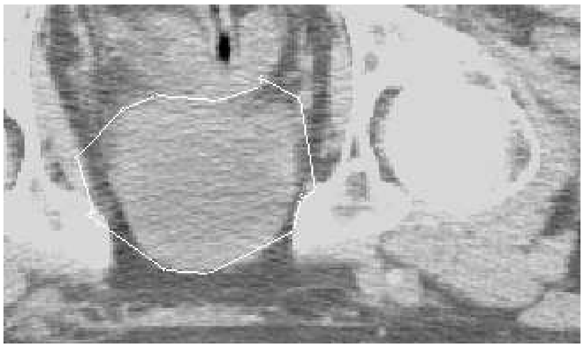


Figure 1.1: *An organ (bladder) next to stronger-edged bone (pelvis). A traditional snake, attracted to strongest edges, incorrectly clings to bone. The solution is to teach it to recognize the qualities of the correct boundary.*

many kinds of tasks involving object classification.

Why learn?

In an image full of edges of various scales and intensities, a boundary-finding algorithm must be made to find a given object. In a complicated scene, there are many possible boundaries. If a good criterion to distinguish the desired one from others is not known in advance, such a criterion must be observed and recorded, *i.e.,* learned or trained. In a deformable model, this criterion, traditionally hard-coded rather than learned, takes the form of a real-valued objective function of the image and a candidate shape.

There are domains in which traditional snakes, attracted to strongest or closest image edges, fail. The strongest edges are often not the edges of the object we seek. This is true in the domain which prompted our studies—CT images of lower abdominal organs, which are pressed up against similar organs and brighter, stronger-edged bone (Figure 1.1), and echocardiograms, which contain inner and outer heart walls, and large blobs of noise. It is not just a problem of finding suboptimal local minima—in such cases, the wrong object satisfies the objective function better than the right one does. Researchers have formulated alternative objective functions to get around the problem (see Section 2.1.1), without formally testing their properties in the domain, and without having some basis to compare the alternatives.

Let us consider what quantity should be optimized when seeking a shape in an image. To find the boundary of a specific kind of object in a cluttered scene, a shape model should know what it looks like, in a way that differentiates it from nearby object boundaries. If this is not known *a priori*, analytically or otherwise, then the model should *learn* what it looks like. That is, through training it should learn the likeliest values of selected image features associated with the boundary of that kind of object. Which features are selected determines, through training, what function will be optimized in order to find an object boundary. Some feature choices will work better than others. Though the incidence of feature values can be learned, feature selection itself cannot, and must rely on human constraint and ingenuity, since the space of projections down to features from the shape and the image (pixels near the shape and not) is too large to parameterize.

If the structure whose contours are sought does not have features (*e.g.,* brightness, edge strength) that are more extreme than those of all neighboring objects, the energy being minimized will need its parameters tailored to some domain-dependent intermediate value. That is, it cannot merely achieve an optimal value when the contour lies on, say, the strongest edges or the brightest region—it must know the *correct* feature strength to reward. This must be found through statistical measurement, which is to say, training. This requires a ground truth training set, which is a collection of images, each with the desired contour correctly specified.

In addition, the method of segmentation by deformable model uses parameters which, if not estimated from domain observations, are *ad hoc*. This has often been the case in the application of this method, and makes the meaning of segmentation results unclear. For instance, changing the weighting of prior shape energy *vs.* image energy (see Chapter 2) will change segmentation results. What result is desired? Domain observation will tell.

Why measure success?

The implementor of a deformable model must have a way to choose from the wide variety of objective functions that have been proposed and used in deformable models. If the function one uses is to be derived from training data, this variety is equivalent to the selection of features to model the distribution of, and of distribution models to use for each. Furthermore, even without a choice, it is essential that we be able to judge the potential success of our given function.

Thus far, no good methodology has been described for this. The comparison and performance evaluation of such functions is domain-specific; the standard procedure is to run the entire deformable model segmentation procedure on a series of images, comparing each resulting shape to the ground truth for that image. But the results of this method only characterize the cumulative effect of: selection of the contour's initial position; the objective function; and the optimization method to find the best nearby contour. It cannot test the effectiveness of their separate contributions to segmentation. Generally, each image is tried with only one initial contour. This method is inadequate to evaluate objective functions, whether derived from training or not.

Thus, a method is needed which can assess the suitability of the objective function alone. How? Contrary to intuition, the distribution of an objective function's values for *correct* contours gives no information about its goodness for segmentation. Consider that, whatever this distribution, if incorrect contours have the *same* distribution of values, the function cannot guide a contour to the correct shape. Qualities consistent across the body of ground truth are useless if they are also unchanged for wrong contours. We must also look at nearby *incorrect* shapes to assure that an objective function responds to increased correctness by getting closer to optimal. This criterion depends on the image environment and variability near each contour. We examine an energy function's behavior for incorrect shapes, by generating such shapes from ground-truth shapes in a sample of domain images.

Why make training spatially local?

We investigate the possibility of making a deformable model respond differently to local image qualities at different places on its boundary. The qualities that training teaches the model to seek may not be uniform everywhere. For instance, in abdominal CT scans, the bladder may be in contact with the pelvis on the sides, seminal vesicles (in men) at two regions near the back, the (highly variable) rectum directly behind, and tissue fluid elsewhere. Thus, its boundary characteristics vary with position (Figure 4.3). This calls for a spatially varying objective function. Furthermore, they may have differing degrees of variability at different places, so the function's sensitivity should vary, providing robustness where wide variation is expected. We will present sectored snakes, which address these needs within the training framework.

The rest of this document

The rest of this document will cover:

- History of segmentation by deformable model, of learning in segmentation, and of measuring segmentation success, in Chapter 2;
- My formulations for learning and for measuring, in Chapter 3;
- My choices within these frameworks, and design of software to do these, in Chapter 4;
- My experiments, in Chapter 5;
- Results, conclusions, and future work, in Chapter 6.

Chapter 2

Background

The work described here concerns training deformable models to find boundaries of known kinds of objects in images from a particular domain, and characterizing the performance of a deformable model in a domain, so as to find the best kind of training.

It is often necessary to find the precise extent of an object in an image (*i.e.,* to *segment* it). Many approaches can be taken, depending on the qualities of the image. Some methods iteratively label pixels as belonging or not belonging to the object, based on their intensity and their neighbors' intensity and labeling. Others find a shape describing the object's boundary, usually based on the output of an edge detector. The method (or class of methods) we address here, the deformable model, finds a boundary, though not necessarily using edgels. Below we describe deformable models.

Then we survey how, in the past, learning has been used to do segmentation. Finally, we look at how segmentation performance has been characterized.

2.1 Deformable models

A *deformable model* is a description of a shape whose parameters are iteratively adjusted until it best matches what is depicted in an image. "Best" is measured by an arbitrary real-valued objective function, or "energy." The process of minimizing this energy is sometimes called a "force." In the case of a 1D contour in a 2D image, the model is called a *snake*, as in the work

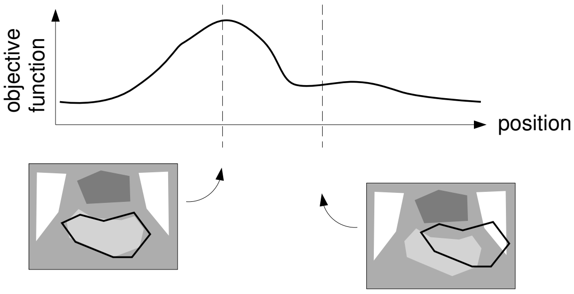


Figure 2.1: *An objective function reacts to adjustments of shape relative to image features.*

that first introduced the method [28]. It can also be used to find surfaces in 3D imagery, as first done in [43]. They, and subsequent work, used a mechanical model: The boundary is attracted by some "image force" to strong edges, and it has some "elasticity" which keeps its shape plausible. Differential equations are set up. As a method of segmentation it has many advantages. It can fit a continuous shape to discrete image data; it accommodates shape models that can change locally to conform to image details and, conversely, can constrain global shape to a given class; and prior shape expectation (local and global) can be incorporated in the objective function. Thus, it can be robust to locally bad data, yet can find local shape where data exists. It can respond to general image-processing cues (edges) or domain-specific information such as particular colors or textures.

A deformable model finds an object in an image by maximizing an objective function of image and shape:

$$\mathbf{S}^* = \arg\max_{\mathbf{S}} f(\mathbf{I}, \mathbf{S})$$

where the $M \times N$ image $\mathbf{I} \in \Re^{MN}$ and shape $\mathbf{S}$ is a vector of shape parameters $(s_1 \ldots s_n)$. $\mathbf{S}^*$ is the optimal shape in image $\mathbf{I}$, according to the measure embodied in the function f, which is usually a weighted combination of several criteria. The inputs to $f\colon \Re^{MN} \times \Re^n \mapsto \Re$ are in theory, the value of every pixel and every shape parameter. The output of f is supposed to reflect our confidence (the likelihood) that $\mathbf{S}$ depicts the shape and location of a particular object in image $\mathbf{I}$. Thus, f encodes information about what such an object is expected to look like in an image. It may be a function of image data inside, on, outside of, or far away from $\mathbf{S}$. In practice, it usually only depends on pixel values on or near the shape boundary.

Usually, f is a combination of two functions. The first, *image energy*, penalizes "unlikely" shapes—ones that vary from some standard shape, or are too bumpy, or have unevenly distributed nodes. The second, *shape energy*, responds to the strength and nearness of image edges or gradients. The shape for which f is maximal has balanced the satisfaction of these two criteria. But in general, f does not need to take this two-term form. Maximizing an objective function is a very general way of finding whatever one is looking for based on almost *any* criterion.

2.1.1 Formulation choices

To implement a deformable model, one must choose three elements. The first is a **shape model**, which can be a set of *snaxels* (separated points or a chain of adjacent pixels), or can be continuous—a polyline (also a set of points, but including the lines connecting them), a surface of polygonal patches, a spline curve or surface, a sum of "modes" based on distribution of sampled shapes [36], a sum of Fourier harmonics, or "vibration modes" [40], etc.

The second element to choose in a deformable model is an **objective function**, f, a real-valued function of the shape parameters and an image. This function choice is the focus of this thesis. Depending on how the optimal shape will be found, the implementor may have to implement not (only) the objective function ("energy") but rather its derivative ("force") with respect to the chosen shape parameters.

A variety of image energies have been used. They have included negated image gradient strength, summed over the shape boundary, which indicates how much the shape lies along image edges. Also used are negated inverse square distance of shape boundary from image edgels, which creates a gravity-like attraction of the shape to them. [11] use a texture measure to see if the expected textures lie inside and outside the shape. [12] and others [13, 11, 27] have combined quantities based on region statistics (pixel values inside *vs.* outside the shape) with image energies as above. [20], choosing very specifically for their image domain, use an energy that tends to center a curve in a constant-intensity ribbon in a cross-sectional brain image.

A variety of shape energies, too, have been employed. Simple ones penalizing variation from flatness include the sum over the shape of its curvature, *i.e.,* second derivative magnitude in the normal direction, or angle or second difference magnitude of discrete points. This works if the shape is known *a priori* to be very smooth. Otherwise, measures of deformation from some rest position, perhaps representing an *a priori* average or expected shape, can be employed. Summing first derivative or difference magnitude keeps the parameterization smooth, and prevents shape degeneracy due to points moving together. Such differential penalties are often formulated as if the shape were a mechanical entity, such as a membrane, rod, or thin plate. If the shape is a finite set of points rather than continuous, it and its energy can be represented using a Finite Difference Model, with vertices connected by springs, with an elasticity matrix penalizing deformation from rest position, with penalties possibly based on observed shape variation in the domain. If the shape model is continuous, its energy can be represented by a Finite Element Model (linear or not): a continuous shape partitioned into piecewise polynomial segments (for 2D shapes) or patches (for 3D), again with a deformation energy, this time based on the entire boundary, not discrete point positions.

Other *a priori* shape energies have been used which also do not involve the pixel data of the image being segmented. These include balloon forces [17], which force a shape outward. These work when the shape can be started inside a homogeneous region, and allow initialization far from its boundary. Also, some models allow the interactive user to specify points that are known to be on the boundary.

Finally, one must choose an **optimization technique** or algorithm to find the shape optimizing the chosen objective function (energy). The usual method is some variety of gradient descent, which requires that the objective function's derivative ("force") with respect to shape parameters be known, and not necessarily the function itself. Success requires that the function is not only minimal for the correct segmentation solution shape, but also that it be monotonic in the region searched (Section 3.2.1); otherwise shape may converge to a local minimum that is not the best in the region. Another optimization method, dynamic programming, avoids the monotonicity requirement, but it only finds the "global" best solution within a specified window around an initial guess at the shape position. One approach [1] only examines image data near user-chosen vertices; another [22] takes time proportional to (pixels in contour) $\times$ (pixels in region searched), with seven minutes reduced to 36 seconds by sacrificing optimality using a multiscale approach, and further to a still-long 8.0 seconds by arbitrarily reducing iterations from (number of vertices) to 2. Stochastic optimization methods have also been applied to this problem, such as "Brownian strings" [24]. Such methods involve decreasing random perturbations to an initial shape, with probabilities biased toward those with scores that turn out closer to optimal. It is possible to find a globally optimal shape if two points are fixed in advance [34, 16], or if absence of local minima is guaranteed by initializing in an "edgeless" region, and ballooning outward [17]. An overview of optimization issues in deformable models, including techniques to ensure convergence, may be found in [31].

All of the optimization methods require a method for making an **initial guess**, which seems to be an inherent limitation of an otherwise very flexible(!) segmentation method. The initial shape may be an *a priori* likely position, or one found by some other method or at lower resolution, or roughly specified by hand.

2.1.2 The physically-motivated approach

Shape optimization is usually treated as mechanical or "physically based" process, especially in less recent work, such as the seminal [43]. The scalar function is called an "energy," $E \in \Re$, and is minimized:

$$\mathbf{S}^* = \arg\min_{\mathbf{S}} E(\mathbf{I}, \mathbf{S})$$

where $E = E_{\text{int}} + E_{\text{ext}}$. E_{int} depends only on shape, and is thus seen as a kind of elasticity, called the *stiffness*, *internal* or *bending energy*. It penalizes unlikely shapes, often by simulating a membrane or thin plate. E_{ext} reflects how well the shape matches features in the image, and is called the *external* or *image energy*.

Iterative minimization is seen as letting a "force" $\mathbf{F} \in \Re^n$, determined by this energy, act on the shape over time:

$$\mathbf{F} = \mathbf{F}_{\text{int}} + \mathbf{F}_{\text{ext}} = -\nabla_{\mathbf{S}} E(\mathbf{I}, \mathbf{S})$$

$$\mathbf{S}_{i+1} = \mathbf{S}_i + T(\mathbf{F})$$

where T scales, and possibly caps, the force $\mathbf{F}$ to be suitably small so the iterative gradient descent will likely be stable and convergent. $\nabla_{\mathbf{S}}$ indicates the vector of derivatives of E with respect to each of the shape parameters defining $\mathbf{S}$.

Sometimes the physical analogy is adhered to more closely by setting up and iteratively solving equations of motion involving "viscosity" $\mathbb{C}$ and "mass" $\mathbb{M}$ matrices, which can help convergence. And if E_{int} is quadratic in the shape parameters, *i.e.* a squared distance, then its negative gradient is a linear force, represented by a stiffness matrix $\mathbb{K}$. So the differential equation solved is that of a damped mass on a spring, whose convergence properties are well understood:

$$\mathbb{M}\ddot{\mathbf{S}} + \mathbb{C}\dot{\mathbf{S}} + \mathbb{K}\mathbf{S} = \mathbf{F}_{\text{ext}}$$

Some people take physical analogy more seriously than others. We [10] see it as merely an analogy between the gradient-descent method of minimization and the physical principal of least action, which moves a system in the direction of greatest energy decrease. After all, the "energy" being minimized, and the parameters which must be chosen, on which it is based, are not based on any physical properties, and do not have meaningful units of measure; it is chosen to be whatever gives a good segmentation. Still, it provides a well-analyzed and implemented framework for setting up equations and making computations stable and efficient. A summary of convergence issues is found in [31].

2.1.3 The probabilistic approach

A more meaningful framework often employed is the probabilistic formulation, which allows the use of observed distributions, which is to say, training or learning.

Here, the objective function approximates the *a posteriori* probability $\mathrm{P}(\mathbf{S} \mid \mathbf{I})$ that the correct shape is $\mathbf{S}$, given that the image is $\mathbf{I}$. This Bayesian formulation is often broken into parts:

$$\mathrm{P}(\mathbf{S} \mid \mathbf{I}) = \frac{\mathrm{P}(\mathbf{S})\mathrm{P}(\mathbf{I} \mid \mathbf{S})}{\mathrm{P}(\mathbf{I})}$$

We seek the shape $\mathbf{S}^*$ most probably depicted by image $\mathbf{I}$ by maximizing $\mathrm{P}(\mathbf{S} \mid \mathbf{I})$ over $\mathbf{S}$. This is the *maximum likelihood* (ML) shape, where "likelihood" in this case means the shape that produced the *maximum a posteriori probability* (MAP). This is the shape that most likely produced the image. Of the two factors, $\mathrm{P}(\mathbf{S})$ is the *a priori* (prior) probability that the shape is $\mathbf{S}$, without knowledge of the image $\mathbf{I}$. $\mathrm{P}(\mathbf{I} \mid \mathbf{S})$ is the probability of observing the features in image $\mathbf{I}$ if it depicts shape $\mathbf{S}$. In other words, it tells if the features $\mathbf{S}$ would produce are present where expected in $\mathbf{I}$.

Maximization may ignore the division by $\mathrm{P}(\mathbf{I})$ because it does not vary with $\mathbf{S}$. This leaves two factors, which is useful here as in physical formulations, because the function is a combination (product or sum) of one that depends on the image ($\mathrm{P}(\mathbf{I} \mid \mathbf{S})$, like E_{ext}) and one that doesn't (the prior shape probability $\mathrm{P}(\mathbf{S})$, like E_{int}). These two quantities are easy to think about and model separately.

But note that $\mathrm{P}(\mathbf{I} \mid \mathbf{S})\mathrm{P}(\mathbf{S}) = \mathrm{P}(\mathbf{I} \wedge \mathbf{S})$. Thus, the optimization result is the same as the shape which maximizes the simple joint probability $\mathrm{P}(\mathbf{I} \wedge \mathbf{S})$, which is a probability distribution in a single high-dimensional space, $\Re^{MN+n}$. In many fields, the feasibility of modeling the prior probability of the outcome, $\mathrm{P}(\mathbf{S})$, is in doubt. But it is done as a matter of course by researchers using deformable models.

(Still, we will take more unified view, modeling $\mathrm{P}(\mathbf{I} \wedge \mathbf{S})$ when we use training. Also, although our work combines, and tests the effect of, probability models of multiple cues, we do not model prior shape as one of those cues; plenty of work has been done on this by others [Section 2.2].)

Researchers have often described their objective function as a Bayesian combination of probabilities without actually deriving its parameters or scaling its components to have the units (dimensions) of a probability, making it merely an analogy, as was the physically-based approach. Thus, there is no confidence that the outcome is suited to the domain it purports to model. The quantities that get optimized do not provide the guarantee of correctness or optimality that the modeling framework implies.

An equivalence is often established between "energy" and probability such that

$$\arg\min_{\mathbf{S}} E = \arg\max_{\mathbf{S}} \mathrm{P}(\mathbf{S} \mid \mathbf{I})$$

Virtually always, the monotonic $E = -\ln \mathrm{P}$ is used, which is the Gibbs energy associated with the Bolzmann distribution in physics, in which the energy of a physical configuration is related to its probability. This equivalence lacks meaning here, since the "energy" does, but sometimes it is convenient to work in negative logs. This formulation turns the maximum of a product of probabilities into the minimum of a sum of energies.

Furthermore, if E is the square of a distance in some shape space from a rest position, as it is if the stiffness is a linear restoring force in that space (elasticity matrix K), then minimization is a least-squares regression, and $\mathbb{P}$, the corresponding probability model, is a multidimensional Gaussian in that space. Its covariance matrix is the elasticity matrix $\mathbb{K}$.

2.1.4 Advantages and disadvantages

Of the various ways to find object boundaries in an image, deformable models have certain advantages:

- **Continuous boundary.** Unlike pixel-labeling methods, but like the Hough transform, segmentation by deformable model can yield the parameters of the continuous shape that best fits the discrete image data. It thus provides interpolation of the image data; it has subpixel precision and properties such as curvature and volume. Such a result can be better used for registration [25] and measurement, and better rendered for visualization.

- **Topological consistency.** The use of a shape model usually also assures connectedness, absence of holes and orientedness of the surface (a well-defined inside and outside) by fixing the manifold's topology *a priori*.

- **Local deformability.** The shape models used are usually locally adjustable, with a large number of parameters (unlike the Hough transform), so they can characterize local variations in shape.

- **Spatial variation in response.** Also, the objective function can locally vary the segmentation's responsiveness to image data of varying expected values and certainty. We have formulated such a model in Section 4.1.3.

- **Conditions on global shape.** But unlike purely local methods such as most pixel labeling algorithms, the objective function can reward closeness to a global shape expectation, such as prior observation or such as an imposed criterion like symmetry [44]; in addition, the choice of shape model can *enforce* global conditions on shape by restricting the class of shape.

- **Interpolating missing boundary cues.** Thus, unlike low-level, local methods of recovering an object boundary, deformable models can robustly handle places where the object becomes the same color as its background, becomes very diffuse or unfocused, is lost in image noise, is occluded, or where image data is otherwise uncertain, because it can keep the shape globally consistent (by any desired definition) in places where local data is absent.

- **Confidence measure.** The result of segmentation by deformable model gives a confidence level for the segmentation, as an inherent part of the method: the level is the objective function value for the optimal shape. This advantage is in common with probability-based pixel labeling, and with the Hough transform, where the number of "votes" a shape gets indicates the amount of supporting evidence.

- **Combining cues.** The objective function that dictates optimality can combine several functions indicating different kinds of evidence of shape correctness, including not just prior expectation but multiple observed kinds of correlation with image data, and even data from several image modalities.

- **Using all available evidence.** More generally, the objective function can reward any image and shape qualities that might indicate segmentation correctness, in the presence or absence of any prior knowledge about the appearance of the object sought. In its absence, a traditional objective function based on generic image cues such as edges can deform the shape to fit any clearly delineated object. But objects whose appearance *can* be modeled *a priori* can be recovered with much greater robustness and accuracy by adapting the objective function to them.

- **Domain appropriateness.** And since a deformable model explicitly maximizes a chosen function, it allows for any definition of the "best" solution.

Unfortunately, the flexibility and adaptability of deformable model segmentation, which is responsible for many of these advantages, is rarely exercised, or its possibilities examined and compared. Adjusting the model to the domain is the heart of the research reported here, and without it, some domains have resisted good automated segmentation. The traditional objective functions, unadapted to domain, can be inadequate.

Deformable models also have some disadvantages:

- **Predetermined topology** is a disadvantage if the sought object's topology is unknown, as with a 2D slice of a 3D blob. This disadvantage can be overcome by adaptively splitting models [21], by merging "bubbles" [42], and by actually deforming a function, a level set of which represents the boundary [32].

- **Not looking everywhere.** The optimization methods do not find the best shape anywhere in the image, but rather one near an initial guess. Gradient-descent methods may find a local minimum that is not optimal, even within this neighborhood.

- **Finding multiple instances is extra work.** An initial guess finds a single instance of an object. One needs a way of getting several or many initial guesses to segment many instances, in domains where images may have them, such as micrographs in which cells are outlined, or lung X-rays where holes from emphysema are outlined. The method itself does not identify multiple candidate objects in an image. And once initial approximations are found by other means, the algorithm's running time is proportional to their number.

- **Only using pixels near the boundary.** Unlike template-based methods [47], deformable models usually ignore potentially useful information in the interior of a shape, although they could be made to react to it.

2.2 Learning for segmentation

Accurate contours of known objects in cluttered images are needed in many applications, including medicine and manufacturing. No segmentation method that does not use domain knowledge will be able to distinguish the sought object from neighboring ones. Since many real-world objects, including human organs, do not have qualities that are known *a priori* or analytically, these qualities must be measured. If image qualities are to guide segmentation, it is the image features of these objects which should be measured. These qualities will generally have distributions, not precisely repeated values. These distributions must be recovered from observations, *i.e.,* training images with known contours. Once they are known, the desired object may be found in a new image by finding the shape whose relationship to the image is likeliest, according to the known probability distributions.

What is *training*? It is a shorter way of saying "inductive learning," on which a book on artificial intelligence is helpful: "Learning agents can be divided conceptually into a *performance element*, which is responsible for selecting actions, and a *learning element*, which is responsible for modifying the performance element.... Learning any particular component of the performance element can be cast as a problem of learning an accurate representation of a *function*. Learning a function from examples of its inputs and outputs is called *inductive learning*" [39].

In a vision system based on a deformable model, the performance element is an objective function and a procedure to find a shape that maximizes it. The action that the performance element selects is either from a space of directions in which to guide a deformable model, or from a space of positions in which to place it. In other techniques, the idea of learning as function estimation is often not made explicit, nor set in the domain of the real numbers. But if we can somehow obtain the objective function of a deformable model by looking at many images with corresponding ground-truth (correct) shapes, we are *explicitly* doing inductive learning by determining a function.

2.2.1 Prior uses of learning in vision

Learning has been widely used for other methods of image segmentation. Among other applications, statistical observation has found thresholds for pixel classification ([41]), using statistical clustering techniques and Fisher linear discriminant analysis (allowing projection to a lower dimension with maximal class separation). Image regions have been projected onto linear eigenspaces to find (or interpolate) the nearest in a learned range of appearances of multiple objects [45, 33], or features [5].

Also common among vision algorithms are those that adopt a probability framework for their algorithms, and then use *ad hoc* formulae and parameter values to achieve their results. The Markov Random Field (MRF) image reconstruction of [23] is a good example of this. Such approaches are also common among deformable models.

With application to deformable models, work has been done on learning specialized feature and shape models, but has not been generalized. Researchers have started to use training in medical segmentation algorithms. But these always trained their models to seek observed values of a single feature picked *a priori*, and do not provide a comparison or methodology for choosing the ones that work best. Any given feature may be ignoring useful image information.

Traditional snakes, by contrast with the learning framework, are based on aligning a contour with image edges of maximum strength. But machine vision cannot always search for such preordained image features. Without knowledge of the domain, an implementor cannot always know what the right answer even is. This ambiguity is what foils traditional snakes in, for example, finding a bladder next to bone, or outlining a ventricle in an echocardiogram composed of small blobs. Thus, in some domains, a snake must be taught about correct and incorrect results.

The usual formulation of the objective function as a sum of image and shape energies lets it represent the log of a joint or conditional probability, so the two energies can model prior shape probability and probability of observed image given shape. Each of these can be a learned distribution. Although there has been plenty of work on learning the prior probability of a shape, there has been much less on learning a model of shape likelihood given an image.

Among learned prior shape models, there have been a multidimensional Gaussian distribution of vertex positions [18]; MRFs of vertex displacements with respect to neighbors [29, 30]; a Gaussian distribution of variations in “vibration modes,” in a representation using them [36]; and a Gaussian distribution in a Fourier harmonic representation [40].

Training on image features, as opposed to training on shape alone, has been done in [18], whose model, in addition to prior shape, learned a kn-dimensional Gaussian of intensities along line segments of n pixels perpendicular to a shape boundary at k feature points. Within a similar framework, [6] used a Gaussian distribution of multiscale intensity and gradient at a few points around many organs simultaneously in a human cross-section. [8] uses a 3D model of liver shape, with an objective function incorporating the observed likelihood of nearby edges being spurious, or not belonging to the object (false positives), or of true object edges not being detected.

The model described in [24] learns a histogram of pixel values on ground-truth shape boundaries. It bases its segmentation on the average of pixel values on the boundary. Finally, [11] use discriminant analysis on ground truth to choose the combination of texture and other measures that best distinguishes correct boundaries.

Figure 2.2: *The prostate is not easy to segment automatically. It is surrounded by edges of other objects, it is the same color as many of them, and its edges are fuzzy in places. Here we see a lower abdominal cross-section of a face-down man. Above the pubic bone is part of the bladder, then the prostate, then the rectum, which contains some gas. On both sides of these are a layer of thin muscle, then a thick muscle, then bone.*

At present, there are areas of real-world application of computer vision in which learning is insufficiently exploited. Medical images—cross-sectional images of patients and 3D stacks of such cross-sections—are an area of particular interest, in which some structures cannot currently be recovered accurately by a computer. In some cases, simple methods do suffice: Objects which are sharp-edged and have a very different color than their surroundings may be separated by thresholding; smooth, continuous boundaries of such objects may be recovered with deformable models; but often, such methods are foiled by nearby, unrelated edges which may be stronger than those of the object sought. And in any case, many important structures, such as the prostate, are neither sharp-edged nor reliably different in color from surrounding structures (Figure 2.2).

In the medical domain, this means that in the field, despite the research, organ contours in some parts of the body are outlined manually, slice by slice in a 3D image. This process is time-consuming and, given the volume of data and number of patients, often rushed.

2.3 Measuring domain performance

After years of neglect, and steadily rising criticism, characterization of the performance of computer vision algorithms is now being given proper attention. As more algorithms have found their way into outside fields demanding quality control, researchers have addressed systematic definition of procedures and measures for performance evaluation as an object of inquiry in itself. The last three years have seen workshops on the topic in Cambridge UK (at ECCV '96), Braunschweig, Seattle (NSF/ARPA), Saarbrücken, Santa Barbara (CVPR '98) and the Canary Islands.

Typical measures of segmentation success are based on ratios of pixels identified correctly and incorrectly as "in" or "out" [35]. For techniques that classify pixels implicitly by finding a boundary, boundary-based measures may be more efficient, or, indeed, of more relevance in the domain. Chalana [14] introduces a shape difference measure for performance evaluation that finds point correspondences between boundaries—difficult for pixel-based approaches. The work then does a statistical analysis to relate the significance of algorithm segmentation error to intra-observer variation. The ultimate domain-relevant measure of segmentation goodness is a characterization of the success of the actual task for which the segmentation is being used. An example would be patient survival rate, when segmentations are used to plan radiation treatments. One could argue that, in any domain, any other kind of measure is *ad hoc*.

Performance analysis has been well-addressed for edge detectors, with extension to other feature detectors. Ramesh [37] and those cited therein have addressed vision tasks that are low-level enough to be amenable to analytic characterization. In its general method, one models possible variations in the input that do not affect the desired output, such as pixel noise; expressions are then derived to characterize how much the output actually is affected, as a function of the degree of the specified input variability. Such characterizations include the receiver operating characteristic (ROC), or signal-to-noise ratio. Results are derived for several feature detectors. Also, this statistic has been estimated empirically [38, 48] using random perturbations of the pixel values in an image.

But sometimes failure of a detector is due not to random pixel noise, but to variation in the environment of objects to be detected (*e.g.,* abdominal organs in different patients). Then, a model of image variation is likely unknown, and complex and inexact (especially for small samples) if and when discovered. Thus, analytically deriving the effect of such variability on a vision algorithm is difficult and undependable, as is using such a model to simulate a variety of inputs from the domain universe. (The author has seen no credible attempt to statistically model intensity images of the crowded abdomen.) To assess algorithm performance here, we must sample domain images, not analytical distributions.

Baker [4] circumvents the assumption of artificial noise models by using natural scenes containing geometric constraints, and checking the conformance of detected features to them.

We have seen no proposed improvements of deformable model testing. [19] give a theoretical analysis which shows that there will be convergence in some limited situations. Generally, when deformable models are validated experimentally, each image is tried with only one initial contour. This method is inadequate to evaluate a deformable model's objective function, whether derived from training or not, for two reasons. First, it vastly underuses the information available in each image—it only yields information about one path followed by one initially guessed shape. It offers no information about how other guesses would have fared. That particular instantiation of the optimization process may have ignored other pixels, giving no information about how they would have affected the segmentation result. Second, it gives no information about the cause of a wrong result, since said result conflates the procedure for choosing an initial guess, the function being optimized, and the optimization algorithm being used.

Like the synthetic measurement of signal-to-noise mentioned above, our work uses stochastic sampling of analytical distributions to measure performance, but randomness is applied to a different part of the system: *solution* shapes are randomly perturbed to determine the behavior of a candidate objective function in real images. We measure not the *result* of the optimization, but rather the presence of a necessary qualities (local monotonicity and optimality) of its key component, and the one with the most latitude of choice: the objective function. This allows analysis of why and when this segmentation component may fail.

Chapter 3

Frameworks Formulated

In this chapter, we contribute frameworks in which to analyze and implement two key elements in the use of a learned deformable model. The first of these is the setting in which an objective function is learned from images and contours. The second, essential even in nontrained models but hitherto overlooked, is the evaluation of an objective function's performance at segmentation. Our particular application of these frameworks will be described in Chapter 4.

3.1 Training deformable models

Here, we formalize the notion of learning a general deformable model objective function based on arbitrary shape and image features. Our point of departure is the standard general statistical learning framework. [46] describes the general inductive learning problem as using a random sample of inputs, with desired outputs for each, together constituting *training data*; this is used to select a function on the space of inputs for which the expected error of the output is minimum. He identifies three main categories within this setting: pattern recognition, in which the Boolean output indicates whether the input belongs to a certain class; regression, in which the continuous output minimizes a residual error; and density estimation, in which the output is the probability of the input. We show that deformable model training falls (or should fall) into this last category.

Our problem differs from pattern recognition in seeking a real-valued figure of merit, or objective function, indicating shape goodness (or likelihood), rather than an indicator function

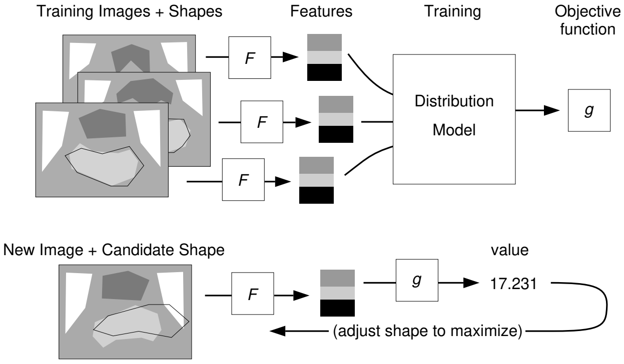


Figure 3.1: *To train a deformable model, we need to decide on a set of features, extracted from an image and shape by function F. We then need a continuous model of the density of feature values which might be observed. Features are extracted from correct image/shape pairs and their distribution is modeled as objective function g. Now, to find the desired shape in a new image, we find a contour in that image which produces the feature values that g says are likeliest.*

with a Boolean output, indicating class membership. But it shares a characteristic with the general pattern recognition framework [2, 15]: The input being classified or rated must first be reduced to a set of well-chosen *features* in a much lower-dimensional space. This is done for a sample population. A model is then trained, using these, to produce an approximation of the desired outputs.

We will see that before learning the probability distribution of image qualities associated with a contour, we must select two things: shape/image qualities (features) to observe, and a model (function space) to fit their probability distribution. The entire process that uses these is show in Figure 3.1.

3.1.1 The training data

Our inputs, from which we learn to recognize correct contours of a particular structure in an image, are a set of images $\mathbf{I}_1, \ldots, \mathbf{I}_n$ for each of which the correct shape $\mathbf{S}_1, \ldots, \mathbf{S}_n$ is known. Thus, the training data consists of *ground truth* pairs

$$\{(\mathbf{I}_1, \mathbf{S}_1), \ldots, (\mathbf{I}_n, \mathbf{S}_n)\}$$

This set should be randomly drawn from the application domain, so as to yield representative proportions of the image qualities that are correlated with the shape.

3.1.2 The features

The goal of training is to somehow produce an objective function that achieves an extremum when its shape argument best traces a particular structure in its image argument. From some class of functions, we must select a function that is extremal for shapes whose relation to the image data resembles shape-image relations seen in the training data.

The "relation" of a shape $\mathbf{S}$ to the image data $\mathbf{I}$ is any aspect $\mathbf{F}$ of the combined information $(\mathbf{I}, \mathbf{S})$; we refer to the vector or set

$$\mathbf{F}(\mathbf{I}, \mathbf{S})$$

as a collection of *features* $\mathbf{F}$ which lives in the chosen *feature space* $\mathcal{F}$. Useful features must be in some way consistent across instances in which $\mathbf{S}$ is correct for $\mathbf{I}$, and depart from this consistency for incorrect shapes. Technically, they must retain high mutual information with shape correctness.[1] If there is such a consistent quality, it will trivially be present if $\mathbf{F}$ is the identity function; but if the task of relating values of $\mathbf{F}$ to a shape's correctness is to be tractable, $\mathbf{F}$ should significantly reduce the dimension of its inputs.

Our goal is to distill an intractable amount of information (perhaps 250,000 pixels plus a complete shape description) down to a dimension in which the distribution of the training instances can be continuously modeled. If our features were the identity function, we would be modeling the distribution of training instances in their original input space; the amount of data we would need to fill this space with a representative population is exponential in the (six-digit) input dimensionality, and would be huge. The number of parameters required to characterize an even somewhat flexible distribution in such a big space would be large. Since there will not be enough training data to estimate hundreds of parameters in a stable fashion, we must project it into a smaller space $\mathcal{F}$ of features before inducing a description of its distribution.

Ideally, an objective function will never be maximal (with respect to shape) for shape-image combinations that are *not* "like" those in the ground truth training data. This would reflect the reality that no such instances had been seen. But the input does not include all the information in an image, but rather features—shape-contingent data extracted from an image. For an implausible shape, such reduced data may still assume values similar to those for plausible

[1] [15], p. 57. *Mutual information* is any measure of the predictability of one random variable, given another.

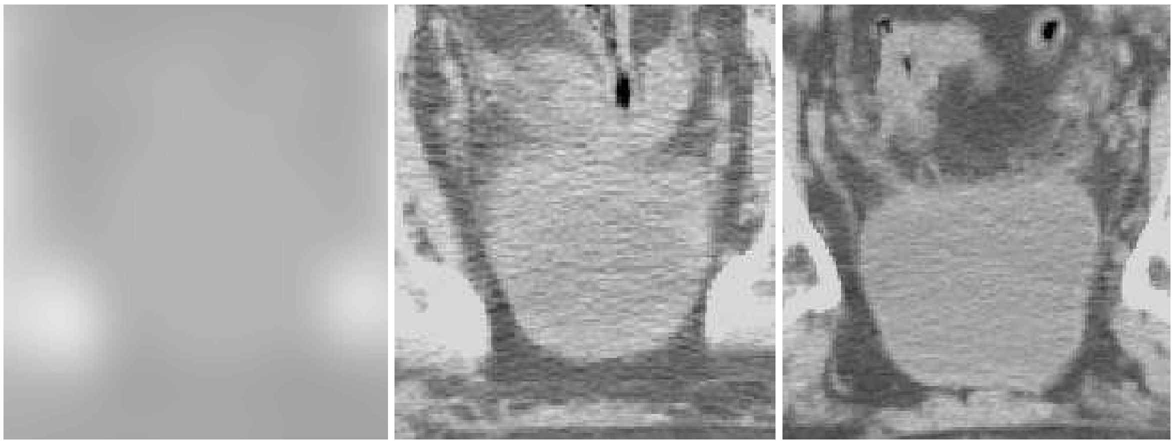


Figure 3.2: *In any domain, some learned image qualities cannot guide a shape to the object boundary,* i.e., *training will not produce a good objective function.* **Left:** *At too coarse a scale, boundary intensity or gradient strength cannot indicate a contour's correctness because distinguishing detail is gone. Tests of discriminatory ability rejected this 8-pixel Gaussian blur in the bladder domain.* **Middle and right:** *If boundary intensity varies more between images than between the object and its neighbors, no simple intensity-based objective function can respond better to the right boundary than to the wrong one.*

shapes, thus misleading the optimization and producing an incorrect segmentation. This is why feature selection is important. A well-chosen feature function **F** is one for which correct shapes will have different feature vectors than incorrect ones. Figure 3.2 illustrates what might make a feature function do poorly.

One corollary of this reasoning is that examining the values of the chosen feature for correct shapes in the image domain yields no useful information by itself. It does not matter if they are widely distributed or narrowly clustered, by some extrinsic standard. What matters is how they change when the shape becomes incorrect. This change must be distinguishable (bigger, or different) from their variation across correct shape features.

There is a combinatorial explosion of features to choose from: all possible subsets or projections of images and shapes input. Clearly, one must *a priori* choose the feature projection **F**, or some severely restricted class thereof. Section 3.2 will develop the criteria for doing this.

In traditional snake formulations where no training is done, features are not separately identified as a step on the way from image/shape to a figure of merit. This is because the entire objective function is specified *a priori*, and does not need to be induced from data; thus, such data need not be produced explicitly. Still, human intuition has "induced" such functions from assumed qualities of features at some level. We can write these explicitly. For example, snakes that respond to the sum of image gradient strengths traversed are responding to some discretization of the features

$$\mathbf{F}(\mathbf{I},\mathbf{S}) = \{||\nabla\mathbf{I}(\mathbf{x})||: \mathbf{x} \in \mathbf{S}\}$$

and those that maximize proximity to detected image edgels use the features

$$\mathbf{F}(\mathbf{I},\mathbf{S}) = \{d(e,\mathbf{S}): e \in \text{edgels}(\mathbf{I})\}$$

3.1.3 The distribution model

In a qualitative sense, we know when a shape in an image captures the boundary of the desired structure by whether the shape-image pair is similar to pairs known to captures their respective boundaries well. More precisely, this means that, projected by $\mathbf{F}$ into some (hopefully tractable) feature space $\mathcal{F}$, this pair is in a region of $\mathcal{F}$ highly populated by the training data. But this means that an arbitrary set of feature values must be mapped to some measure of populatedness in the vicinity, $g(\mathbf{F}(\mathbf{I},\mathbf{S}))$:

$$\mathbf{F}(\mathbf{I},\mathbf{S}) \quad \overset{g}{\mapsto} \quad p$$

To do this, a finite set of training points in feature space must be extended to yield density values everywhere in that space:

$$\{\mathbf{F}(\mathbf{I}_1,\mathbf{S}_1),\ldots,\mathbf{F}(\mathbf{I}_n,\mathbf{S}_n)\} \qquad \overset{\text{training}}{\longmapsto} \qquad g(\cdot)$$

Since this task is underconstrained (ill-posed), some kind of structure or regularization must be imposed on the density [46] to get a unique or stable solution.

When the shape of a known type of structure is to be found in an image $\mathbf{I}$ using a deformable model, knowledge of the structure is encoded in a feature density function g that is the result of training using $\mathbf{F}$. We want the shape $\mathbf{S}^*$ which, paired with $\mathbf{I}$, yields the features that are most "like" those in the training set, *i.e.,* the maximum density on the restriction of $\mathcal{F}$ to those features $\mathbf{F}$ that a shape can generate in image $\mathbf{I}$:

$$\mathbf{S}^* = \arg\max_{\mathbf{S}} g(\mathbf{F}(\mathbf{I},\mathbf{S}))$$

The definition of "like the training set" is embodied in the class of functions $\mathcal{G}$ from which g is drawn. $\mathcal{G}$ can be thought of as defining what it means for different values of a feature set to be close, or for a neighborhood of feature space to be highly populated. More intuitively, $\mathcal{G}$ is what the algorithm designer must choose well, so that it has a member $g \in \mathcal{G}$ that can be large for feature values like those in training and small for those produced by shapes that are wrong for an image. That is, g must be adjustable enough to capture the density variations of the training data in $\mathcal{F}$; otherwise, features unlike any observed in training may be deemed to be in a dense region, *i.e.,* "close" to correct-shape features. Conversely, the family $\mathcal{G}$ from which g is drawn must be constrained enough to generalize the data, and label other feature values in the midst of training points as being in the dense area. That is, a density function with too many parameters will overfit data—be high for values in the training, but near zero for similar features. Finally, the amount of ground truth data available must suffice to estimate the parameters of the model chosen.

The space of Gaussians has few enough parameters that it is unlikely to overfit the data. But other choices may be more able to characterize the observed distribution of feature values. A formulation could use k-means clustering, which can characterize a distribution in the same space as a Gaussian, but with more degrees of freedom—it is a mixture of Gaussians, and can representative multimodal feature distributions. Other models with even more degrees of freedom are possible; for instance a cumulative distribution can be fit by a monotonic spline.

If g is an estimate from the training data of the probability density $p(\mathbf{F})$ over feature space, then $\mathbf{S}^*$ is a *maximum likelihood* (ML) shape estimator, one whose correctness would maximize the probability of observing features $\mathbf{F}$. If $g(\mathbf{F}(\mathbf{I}, \mathbf{S}))$ has a factor that depends on $\mathbf{S}$ and not $\mathbf{I}$ then that factor is a Bayesian prior shape probability, g is an *a posteriori* probability, and $\mathbf{S}^*$ is a *MAP* estimator.

We have not yet specified how $g \in \mathcal{G}$ is determined from the training data. g itself can be chosen in an ML framework. That is, to approximate the probability density of features, we want the parameters of g which are most likely, given the training data. So we find those defining a distribution that maximizes the joint probability of observing the training set's features, $\mathrm{P}(\mathbf{F}(\mathbf{I}_1, \mathbf{S}_1) \wedge \ldots \wedge \mathbf{F}(\mathbf{I}_n, \mathbf{S}_n))$. Assuming that the training data were drawn independently, we thus find the function g that maximizes

$$\prod_{i=1}^{n} g(\mathbf{F}(\mathbf{I}_i, \mathbf{S}_i))$$

For Gaussian distributions, the mean and variances (and covariances, for a nondiagonal model) maximize this likelihood.

The ideal function g maximizes this product over the entire space of images and corresponding correct shapes in the domain; but in practice, its parameters can only be estimated based on the finite, random training set. This estimate has an uncertainty based on the size of the set. Thus, the performance resulting from training is only as good as the standard error of g's parameters.

As an example of the process, assume that features are tuples of fixed size n, and $\mathcal{G}$ is the family of nD Gaussians, having $(n^2 + 3n)/2$ parameters. Then if the training features are subjected to a principal component analysis, a.k.a. Karhunen-Loève transform, the resulting parameters are the mean, variances and principal axes of the multidimensional Gaussian most likely to have generated the data. This Gaussian is the maximal objective function g drawn from $\mathcal{G}$ above. A feature vector that falls a certain distance from the mean along a high-variance axis will fall in a higher-density region of g, and be evaluated as more like the features in ground truth examples, than a feature vector at the same distance but along a low-variance axis. A different family $\mathcal{G}$ of Gaussians, with just one variance parameter, would necessarily yield a training result g that evaluated the two feature values as falling at equal densities, and thus judge them as representing shapes that were equally correct in their respective images. This $(n + 1)$-parameter model would capture the domain's feature distribution more poorly, but the estimate of its parameters would be stable with less data.

3.2 Measuring domain performance of deformable models

In this section we describe the conditions necessary for a deformable model's success, and detail a method which follows straightforwardly from them for testing a particular model's applicability in an image segmentation domain. The conditions and method apply to the domain-dependent portion of a model, *i.e.,* the objective function f. They apply whether the function results from training, or from the traditional approach in which it is specified *a priori*; they are particularly important in trained models, where the function $f = g \circ \mathbf{F}$ is a composition of feature selection and a feature probability density model, both of which must be selected appropriately for the domain.

Performance is characterized here by measuring how the objective function reacts to near-correct shapes in actual, complex image data from a domain. It is too complex to analytically model variety in the proliferation of distracting, indistinct neighboring image structures in a natural application domain, for instance a cross-section of the human abdomen. In all likelihood there is insufficient domain data to estimate a parameterization of such complex, unknown phenomena. Yet an objective function performs well if it distinguishes these from the structure sought. Rather than impose an information-losing step of modeling domain images and then characterizing performance, we measure performance directly using the data that would have been used for the modeling. In a simpler domain, *e.g.,* theoretical edges corrupted by Gaussian noise, the input modeling approach makes sense [38].

3.2.1 Criteria for performance characterization

The primary condition that an objective function must satisfy is obvious:

- **Absolute optimality:** The ground-truth shape must score better in the image than any shape that might be tried during optimization, and that differs from it by more than some tolerance.

The tolerance depends on the precision required by the domain; and, also, on the accuracy with which the truth can even be known. There may be a degree of variation in "ground truth," *e.g.,* among segmentations drawn by different experts of the same image. No test can report accuracy finer than the ground truth's.

A second condition is of practical importance when the usual gradient-descent methods are used for optimization:

- **Relative optimality:** As a shape approaches ground truth, its score must improve, within the range of shapes tried during optimization.

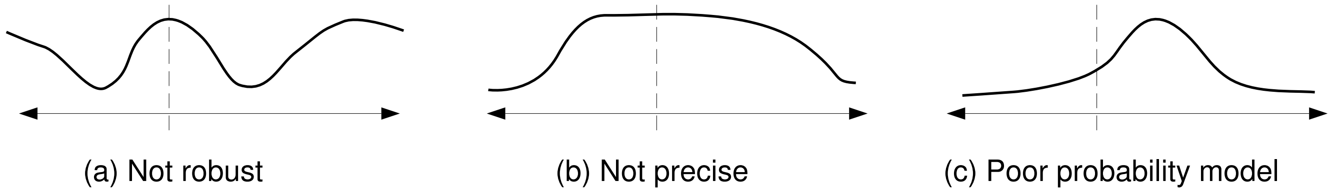


Figure 3.3: *Ways that an objective function can be bad. The horizontal axis represents small variations away from the correct object shape (the dotted vertical) in an image to be segmented. The vertical axis is the response of a (possibly trained) objective function. Function (a) falls off, as it should, but gives as big a response to a nearby incorrect shape, to which it is thus not robust. (b) is flat, and so cannot locate the correct shape with precision—only a shape that is displaced quite a bit will be judged worse. (c) is not able to capture the correct shape at all for this image, perhaps because it was trained on unrepresentative data, or because its functional form (*e.g.*, unimodal) cannot capture the distribution pattern of the data.*

Otherwise, gradient descent will return an incorrect local optimum.

Two qualities determine an objective function's success, by either criterion:

1. **Precision:** The function's flatness at its optimum—the amount by which a ground-truth shape must be perturbed, in an image, before the function significantly changes value. Optimization cannot be expected to find a shape any closer to correct than this.

2. **Robustness:** The amount of perturbation needed before other extrema (if any) are reached, or before the shape's score starts improving again, due to nearby image structures with similar feature values. Optimization must only try shapes within this distance to avoid distraction by such structures.

These attributes are qualitative, or at best statistical, since they will vary from one image to the next and require a notion of significant change, but their effects can be seen in plots of shape correctness *vs.* function value.

Precision is of interest in comparison to the tolerance that a domain demands of the shape returned by the segmentation algorithm. An example of lack of precision is provided by image features we tested which were found to be at too coarse a scale (Figure 3.2). With that amount of blurring, intensity and gradient varied less, over a region greater than the desired tolerance, than from one image to the next. This yielded a trained objective function that did not change appreciably in that region.

An example of lack of robustness is provided by traditional snakes, which find a shape that lies along maximum image gradient strength, in the CT bladder domain (Figure 1.1). Although the function falls off as the bladder outline is perturbed, it only has to be perturbed by a small amount, in some images from this domain, before it lies along other, stronger gradients (bone).

3.2.2 Method: neighborhood sampling

The qualities described above concern the relation of a shape $\mathbf{S}$'s distance from ground truth in image $\mathbf{I}_i$, denoted $D(\mathbf{S}_i, \mathbf{S})$, to its objective function value $f(\mathbf{I}_i, \mathbf{S})$. To test for these qualities, f's behavior must be known for shapes $\mathbf{S}$ close to the correct one $\mathbf{S}_i$, over the variety of images to be encountered. "Close to" must be a neighborhood N in shape space large enough to include initial guesses used for optimization. (Even optimization methods that find "global," not local, optima, only operate within some pixel distance of an initial shape estimate [22, 1]. This is not a bad thing—as well as providing efficiency, it makes such methods robust to far-away optima.)

Given that f's behavior in a finite shape region is sought for images whose theoretical qualities are not known *a priori*, it makes sense to examine f's behavior experimentally, by evaluating it for shapes sampled in the region. This procedure is applied to test images $\mathbf{I}_1, \ldots, \mathbf{I}_n$ in which the true shapes $\mathbf{S}_1, \ldots, \mathbf{S}_n$ are known. (For proper testing, these should be distinct from the training data, if any.) Thus, the basis of our test of an objective function will be m shape samples per image, generating the pairs (D, f):

$$\begin{gathered}
\{(D(\mathbf{S}_1, \mathbf{S}_{1i}), f(\mathbf{I}_1, \mathbf{S}_{1i})) \colon \quad 1 \le i \le m, \quad \mathbf{S}_{1i} \in N(\mathbf{S}_1)\} \\
\vdots \\
\{(D(\mathbf{S}_n, \mathbf{S}_{ni}), f(\mathbf{I}_n, \mathbf{S}_{ni})) \colon \quad 1 \le i \le m, \quad \mathbf{S}_{ni} \in N(\mathbf{S}_n)\}
\end{gathered}$$

The result of such testing, n datasets, each consisting of a large sample of pairs, holds all the information needed to characterize the behavior of f in the application domain.

3.2.3 Ingredients for performance characterization

There are several domain-dependent choices which must be made to evaluate the performance of a deformable model. We will now examine them individually.

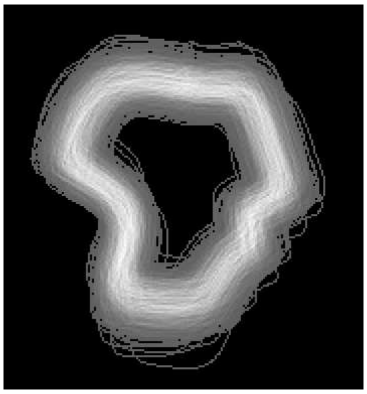


Figure 3.4: *Cloud of 1,000 perturbed versions of an inner heart wall contour. Translation had a standard deviation of 5 pixels, and scaling (independently in two random orthogonal directions) had a standard deviation of 10%.*

Range of contours: A statistical approach

We must define a process that generates perturbed versions of the ground-truth shape in an image. These deformations should be representative of shapes encountered during the process of finding the optimal shape starting with possible initial approximations or guesses. If the range of possible starting shapes is not known precisely, it is best to err by testing over a larger range, although poor test results may then, in fact, be inconsequential.

Shapes $\mathbf{S}_{ji}$ could be generated on a mesh of shape parameters near those of the ground truth $\mathbf{S}_j$, but for shapes with more than fifty parameters, *e.g.,* polylines, such a mesh would be huge. Since we cannot densely cover the high-dimensional neighborhood of all nearby or similar shapes, we must sample a subspace of deformations having an appropriately chosen basis. These can be generated on a mesh, or, as with Monte Carlo methods, the space can be sampled stochastically to get a statistical characterization of function behavior.

Thus, we generate a set of parameterized perturbations. It is chosen to expose expected kinds of flaws in the objective function(s) we are testing. We want to discover nearby shapes that, under the candidate objective function, vie with the ground truth shape for optimality. These would almost certainly include low-order deformations such as translation; they may or may not include high-frequency components, or bumps.

A measure of shape distance to ground truth

To see how $f(\mathbf{I}_j, \mathbf{S})$ behaves as $\mathbf{S}$ gets farther from $\mathbf{S}_j$, there must be a measure D of "farther." It should be such that closer is always better, as defined in the domain. Thus, it should conform to at least two properties of the mathematical definition of a distance measure: the distance of a shape to itself must be zero, and to others must be positive:

$$\forall \mathbf{S}_a \forall \mathbf{S}_b \quad D(\mathbf{S}_a, \mathbf{S}_b) \geq 0,$$
$$\forall \mathbf{S}_a \forall \mathbf{S}_b \quad D(\mathbf{S}_a, \mathbf{S}_b) = 0 \iff \mathbf{S}_a = \mathbf{S}_b$$

There does not appear to be a need for commutativity, since the roles of ground truth shape and optimization's candidate shapes need not be interchangeable; it is unclear what role the triangle inequality would play.

The shape distance measure chosen will depend on the significance of different kinds of shape incorrectness in the problem domain. For example, one application might consider a bumpy contour that repeatedly crosses the correct smooth one to be preferable to a simple small translation of the correct one. In another application, this priority might be reversed.

Several well-known measures of shape difference could be candidates in a given domain. One would be the Hausdorff distance [26], which is the farthest distance of any point on one contour to the closest point to it on the other. The Hausdorff distance treats a single, local five-pixel deviation the same as displacement by five pixels everywhere, and thus would not be a good indicator of whether an incremental shape improvement were accompanied by an improvement in objective function score.

The symmetric difference of the areas enclosed by two boundaries would be a very intuitive candidate for shape difference measure, and in fact a standard method of quantifying segmentation error is to use various ratios of the number of pixels classified correctly and incorrectly as "in" or "out" of the pictured entity [35].

The chamfer distance between two boundaries [9] is the average over one shape of distance to the closest point on the other. To calculate this, one does a distance transform on an image containing one shape, then averages the distance values intersected by the other. This measure is asymmetric; a commutative measure would be the average of the two directions. The asymmetric chamfer distance gives a result of zero in the pathological case where two curves coincide except

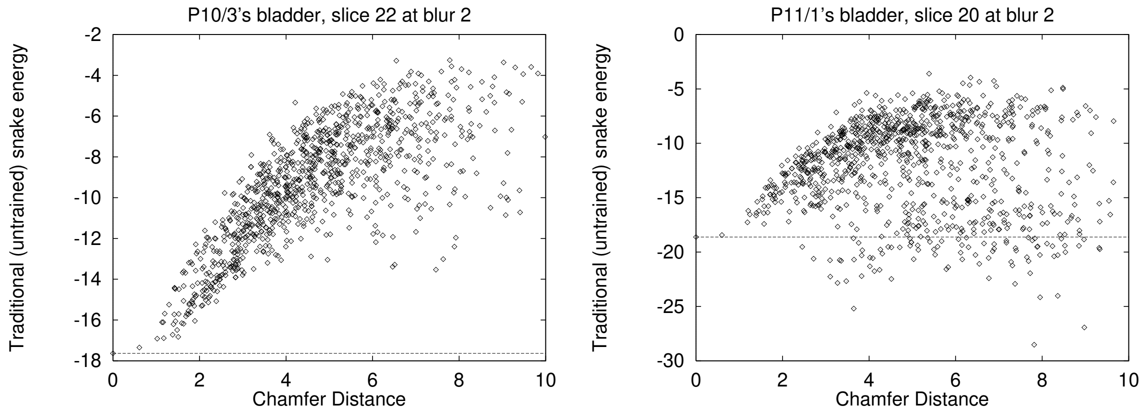


Figure 3.5: *Examples of plots showing how effective a deformable model's objective function will be in a domain. Given an image and the correct contour of a shape in it, a large set of perturbations of the shape are generated. Each perturbed contour is a point in a scatter plot of difference from correct shape* vs. *energy. A good objective function will display strong correlation with shape correctness, and will have no "false positives," contours with lower energy than the ground truth (dotted line).* Left: *a high-correlation (0.73) plot with no false positives.* Right: *A low-correlation (-0.04) plot with 8.8% false positives. Each plot characterizes an energy function's behavior in a single image. To rate the function's performance in a domain, statistics of plots from many images must be aggregated.*

for a protuberance on the one to which the distance transform is applied. It is, however, more robust to this pathology than the Hausdorff distance; also, the forward-reverse correlation over our test set (Section 4.2.2) shows that it did not arise much.

There are also shape difference measures that are based on explicitly finding correspondences between the shapes. Such an approach seems inherently unstable; but [14] is an elegant one, based on averaging the shapes and iteratively readjusting the correspondence. It is claimed to converge after a few iterations.

The shape distances above are general. If closeness in certain parts of a contour is less important than in other parts, a domain-specific distance measure could give those portions less weight. In this way, an objective function would be judged good, or monotonic, if it improved when shapes got closer in important regions but farther in unimportant regions. Such a distance measure could be recovered using many expert ratings of inputs, plus a model.

Analysis tools

The relationship of objective function to known shape correctness (distance to ground truth) over the expected range of shapes tells us whether the function will perform well. For each test image, this relationship is described exactly by a scatter plot of these two quantities for many perturbed ground truth contours. There are various ways to turn such a scattering into measures of the properties we desire.

Since these performance measures are found per image, one must combine its values for many images, whatever statistic is chosen, to measure overall goodness in the domain. One can statistically aggregate the measures; more informatively, one can plot the number of test images that fall within successively more relaxed goodness tolerances. The latter method allows one to see whether some images might be considered outliers.

The most important measure is of the absolute optimality of the ground truth. The most straightforward measure of this is the number of **false positives**, contours having energies below that of the ground truth, thus falsely indicating that they are better solutions than the correct one.

$$\text{FP}_j = \frac{1}{m}\sum_{i=1}^{m}\begin{cases} 1, & f(\mathbf{I}_j, \mathbf{S}_{ji}) < f(\mathbf{I}_j, \mathbf{S}_j) \\ 0, & \text{otherwise} \end{cases}$$

Ideally there would be none of these falling within the shape's neighborhood but outside of some tolerance ϵ. Such tolerance is determined by the use to which the shape resulting from segmentation will be put; and by the accuracy to which ground truth is known in the first place.

A measure (or plot) providing more information would relate number of false positives to distance from ground truth—this would provide information about precision, that is, how far away false positives stop appearing. Depending on the tolerance demanded by the domain, false positives for small enough perturbations would not actually be considered "false." Such tolerance could actually vary between different parts of the shape.

As well as absolute optimality, if we use a gradient descent method, we wish to test relative optimality—whether any deformation of a shape that decreases its energy actually brings it closer to ground truth. We get the strongest assurance of this from scatters which are most tightly clustered around some monotonically increasing function. (In theory, such assurance is one-sided—if shape score is not monotonic with respect to distance, all individual approaches to the true shape may still be monotonic.) There are various ways that monotonicity with distance may be quantified.

A simple test is by calculating the **correlation coefficient** for each image j,

$$\sigma_{Df} = \frac{1}{m}\sum_{i=1}^{m}\frac{(D_i - \bar{D})(f_i - \bar{f})}{\sigma_D \sigma_f}$$

where $D_i = D(\mathbf{S}_j, \mathbf{S}_{ji})$, $f_i = f(\mathbf{I}_j, \mathbf{S}_{ji})$ and $\bar{D}$, $\bar{f}$, σ_D and σ_f are their averages and standard deviations over i. The correlation coefficient σ_{Df} indicates monotonicity because, having normalized away the actual amount of variation of the D_i's and f_i's, the sum of products is bigger when larger D_i's correspond to larger f_i's. σ_{Df} quantifies how closely the data is clustered around an increasing straight line—the closer it is to +1, the better.

However, we do not require a good objective function to be close to a straight line, but just to any monotonically increasing function. Thus, the ideal indicator of robust gradient descent behavior would be how far the point set $\{(D_i, f_i) : 1 \leq i \leq n\}$ is from the **closest increasing function**. The root-mean-square residual distance, normalized by the variance of the f_i, is a measure of monotonicity which varies from 0, if the data is strictly increasing, to 1, if it is strictly decreasing. Randomly fluctuating data, with 1,000 samples, has been found to yield measures within 1% of 1.

This method of characterizing a two-dimensional data set's closeness to an increasing function is logically derived from basic principles. Assume function $f(x)$ and data $S = \{(x_i, y_i) : 1 \leq i \leq n\}$. We define the "distance" of S from f as the sum of squared differences of residuals, $\sum (y_i - f(x_i))^2$. If one increasing function f_0 minimizes this distance, then so do many others, since the distance only depends on f's values at the x_i, and, keeping these fixed, there are an infinite number of ways for f to increase between consecutive x_i. Thus, the problem is reduced to finding an increasing set of $f(x_i)$, or f_i, that minimizes the sum of residuals. Such an answer specifies an infinite set of closest increasing functions, all of which have the same residual error. This answer, and its corresponding error, do not depend on the values of the x_i, only on the ordering of the y_i. It is the residual error which gives a measure of how close the data set is to an increasing function. An algorithm to find the error only needs as input an ordered sequence of y_i.

The closest increasing sequence is the complete-ordering case of *isotonic regression*, and the efficient Pool Adjacent Violators algorithm of Ayer, Brunk *et al.*[3], among others, finds the closest increasing function to our ground truth test data. Several algorithms are discussed in Barlow *et al.*[7].

Algorithms for finding $\{f_i\}$ are based on the insight that the closest increasing sequence to a decreasing sequence of y_i is the constant sequence consisting of its mean. An initial sequence $\{f_i^0\}$, not yet made increasing, is set equal to the data, and in successive iterations $\{f_i^k\}$, decreasing subsequences are replaced by their means. These constant runs are stored efficiently. The algorithm can be proved to terminate, and the sequence, modified in place, is the closest nondecreasing sequence to the input sequence. Our algorithm implementation, in C++, is in Figure 3.6.

Putting it all together

Once we have a model of how to perturb ground-truth contours, and we have a measure of the distance between two contours, we can proceed to characterize the goodness of a deformable model's objective function. This goodness can be summarized in a plot of shape incorrectness *vs.* function value (Figure 3.5).

```
void closest_increasing (double *data, int size)
{
    int i;
    // These arrays allow us to skip elements, denoting a constant region:
    int *next = new int [size];
    int *prev = new int [size];
    // But initially, we skip nothing:
    for (i=0; i<size; i++) {
        prev[i] = i-1;
        next[i] = i+1;
    }

    // Replace nonincreasing runs with their mean:
    i=0;
    while (i < size) {
        // Find limits of nonincreasing region containing i:
        int left=i, right=i;
        while (prev[left] >= 0  &&  data[prev[left]] >= data[left])
            left = prev[left];
        while (next[right] < size  &&  data[next[right]] <= data[right])
            right = next[right];

        if (left == right) {
            i = next[i];
        } else {
            // Replace [left..right] with mean value, stored in [left]:
            double sum=0;
            for (int j=left; j<=right; j=next[j])
                sum += data[j] * (next[j] - j);
            data[left] = sum / (next[right]-left);
            next[left]=next[right];
            if (next[right] < size) prev[next[right]] = left;
            i=left;
        }
    }

    // Done!  Fill in constant runs skipped over by 'next':
    int last=0;
    for (i=0; i<size; i=next[i]) {
        for (int j=last+1; j<i; j++)
            data[j] = data[last];
        last=i;
    }
    for (int j=last+1; j<i; j++)
        data[j] = data[last];
    delete [] next;
    delete [] prev;
}
```

Figure 3.6: *Closest increasing sequence.*

Chapter 4

Implementations

Having presented a general formulation of the problem of training a deformable model to outline a given object in a given image domain, and having described the framework by which a model's objective function can be tested, we now address the specific designs and choices with which we implemented these methods.

4.1 Training framework

To actually realize software that induces an objective function from a set of images and shapes, one needs an architecture into which the range of mathematical choices can fit. We describe this, followed by the set of choices we made, which includes training based on a spatially varying probability distribution.

4.1.1 Software tools

The software tools implementing trained snakes are:

- **`xsnake`**, a visual interface allowing shape input and optimization, by gradient-descent deformation, of a selectable trained or untrained objective function;
- **`energy`**, a simple command accepting a shape, an image and an objective function name, and printing its value for those inputs;

- **`train`**, a command accepting a shape, image and objective function name and outputting feature distribution parameters for that function, observed in that image with that shape;
- **`combine-train`**, a command aggregating specified sets of feature distribution parameters.

These tools allow the production and use of a trained deformable shape model for segmentation. Tools for testing, and thus selection of the best objective function, are covered in section 4.2.1.

4.1.2 Software design

The language of our implementation was C++, which made an object-oriented approach natural, allowing "plug-and-play" flexibility in our deformable model training testbed. Interfaces were designed for a `Training` object, which stores, inputs, outputs and aggregates information observed about selected image features in training data which determines the values of parameters for a selected distribution function; and for `Force` objects, which use the information in a particular subclass of `Training` to calculate the objective (distribution) function value, and its derivatives, with respect to a selected shape parameterization.

Shape representations are currently limited by the interface design to those representable by piecewise cubics; those we have used are polylines and C^2 cubic splines. The limitation could easily be removed in favor of an abstract curve object mapping $[0, 1)$ into image space, $\mathcal{Z}^2$.

Routines using these interfaces can be passed any object implementation conforming to them. Substitution of one for another is transparent. Any one must implement the operations presented by the interface. The interfaces are shown in Figure 4.1 and Figure 4.2.

`Training` object

`Training` is an interface to an object that stores information necessary to determine the parameters of a particular feature distribution (or other objective function). The model adopted assumes that information may be collected into such an object from individual images, and that information from multiple objects may be aggregated into a new one. The **`add`** operation does this. There

```
class Training {
public:
    virtual ~Training () {}
    virtual char const *type () const = 0;
    virtual bool add (Training const &) {return true;}
    virtual bool read (const char *file_name, double &scale,
                       Array_Owner<char> &err_msg) {return true;}
    virtual bool write (const char *file_name,
                        Array_Owner<char> &err_msg) const {return true;}
    virtual void forget () {}
    virtual bool is_trained () const {return true;}
};
```

Figure 4.1: *The interface for the `Training` class, one of the two abstract classes underlying the deformable model training testbed. Subclasses of `Training` store statistics from training observations, which parameterize a PDF. They are calculated by corresponding subclasses of the containing `Force` object, from a preprocessed image and a ground-truth shape representation. Member function `add()` aggregates stored training statistics from this and another object of the same subclass, allowing incremental learning. The empty base class `Training` may be instantiated as is, and will represent null training, with no state, useful for untrained objective functions.*

are also **`read`** and **`write`** operations to save state to, and restore state from, a file.

As an example, there is a subclass of `Training` used to store data for a Gaussian distribution model of pixel intensity and perpendicular gradient along shape boundaries. This subclass stores the sum of observed intensities and of perpendicular gradient magnitudes, and their squares. `add` simply accumulates the sums of these from multiple such objects. These two sums determine a joint distribution of two Gaussians.

The `Training` object does not need to be subclassed—it can be instantiated as is, in which case its operations do nothing and it has no state. Such "training" is used by objective functions or distributions that are known *a priori*, for which no parameters are learned. One example is the traditional snake energy function, which gives the summed edge strengths around a boundary.

`Force` object

Subclasses of the **`Force`** implement different objective function models. A `Force` object relates actual images and shapes to the data determining a distribution. This interface implements objective function operations on an image and a particular shape. Some operations produce dis-

```
class Force {
public:
    virtual ~Force () {}
    virtual Training &training () = 0;
    virtual Training const &training () const = 0;

    virtual bool preprocess_image (image_t &im, double scale,
                                   char const *file_name, int slice)
        {return true;}

    virtual bool energy (double &e,
                         const Cubic_Spline &, int segment, double u) = 0;

    virtual bool force (Vector2D &dE_dp, Vector2D &dE_dn,
                        const Cubic_Spline &, int segment, double u) = 0;

    virtual bool train (const Cubic_Spline &) {return true;}

    virtual void print_training_data (ostream &, const Cubic_Spline &) {}
};
```

Figure 4.2: *The interface for `Force`, the second of the two abstract classes underlying the deformable model training testbed. Objects can only be instantiated for subclasses derived from `Force`, with actual methods filled in. Such a subclass will contain a particular `Training` subclass which parameterizes the objective function it implements. The `energy` method uses this and the (preprocessed) image and snake to calculate the objective function; the `force` method calculates its vector derivative with respect to the snake's shape parameters. Learning is done through `train`, which incrementally updates the objective function parameters based on the image and snake.*

tribution data; others use it to calculate an objective function value, or its derivatives with respect to shape parameters.

Each `Force` object contains a `Training` object whose collected data is used as the parameters of the `Force` object's objective function. An untrained objective function would use an empty `Training` object. `Force`'s operations include:

- **`train`**, which updates the `Training` object's aggregated training data to include that from the given image and shape.

- The **`energy`** operation evaluates the objective function in an image at one point on a shape, also given its normal:

$$f(u) = f(\mathbf{I}, \mathbf{S}(u), \mathbf{S}^{\perp}(u))$$

- The **`force`** operation returns the gradient (partial derivatives) of the objective function at a point on a shape, with respect to that point and the shape's normal there,

$$\partial f / \partial \mathbf{p} \qquad \text{and} \qquad \partial f / \partial \mathbf{n}$$

with

$$\mathbf{p} = \mathbf{S}(u) \qquad \text{and} \qquad \mathbf{n} = \mathbf{S}^{\perp}(u)$$

Although this design limits objective functions to those which are sums over the points in the shape,

$$f(\mathbf{I}, \mathbf{S}) = \int_u f(\mathbf{I},\, \mathbf{S}(u),\, \mathbf{S}^{\perp}(u))du$$

it makes f independent of the shape model $\mathbf{S}(u)$. This is desirable because different shape models may be used interchangeably with a `Force` object without recoding its formulae for function or derivatives.

Thus, force and energy on an entire contour are integrals—actually normalized sums of the respective quantities at one-pixel arc-length intervals, evaluated by a loop outside of the `Force` object, in an object called `Snake` which deforms according to total energy or force on the curve. An approach that includes prior shape probability (energy) is often able to calculate such energy in closed form from shape parameters. Since such efficiency is not possible using the above interface, prior shape energy can also be calculated outside of the `Force` class, in code that is shape-dependent.

To be deformed by a `Force` object, any shape model must provide operations $\mathbf{p} = \mathbf{S}(u)$ and $\mathbf{n} = \mathbf{S}^{\perp}(u)$, and also their derivatives with respect to adjustable shape parameters. For optimization, the `force` operation's shape-independent results ($\partial f / \partial \mathbf{p}$ and $\partial f / \partial \mathbf{n}$) must be multiplied by shape model dependent (but objective function independent) derivatives $\partial \mathbf{p} / \partial S_k$ and $\partial \mathbf{n} / \partial S_k$ to get the vector used to update shape parameters S_k by gradient descent:

$$\begin{aligned} \nabla f &\equiv (\ldots, \; \frac{\partial f}{\partial S_k}, \; \ldots) \\ \frac{\partial f}{\partial S_k} &= \int_u \left(\frac{\partial f}{\partial \mathbf{p}} \cdot \frac{\partial \mathbf{p}}{\partial S_k} + \frac{\partial f}{\partial \mathbf{n}} \cdot \frac{\partial \mathbf{n}}{\partial S_k} \right) du \end{aligned}$$

To assist convergence to a solution, this "force" which deforms the shape may be scaled or thresholded as part of the optimization method, for instance to limit movement to no more than one pixel, or it may be damped [31].

This design could be extended to accommodate functions of local differential properties of $\mathbf{S}$ other than $\mathbf{S}^{\perp}(u)$, such as $\ddot{\mathbf{S}}(u)$ for curvature-based "stiffness" penalties. All shape models would then have to provide these quantities and their shape-parameter derivatives. f may also depend on u, for (parametric) spatially varying response.

The availability of both `energy` and `force` allows the use of various optimization methods—those that use the objective function's value and those that use its derivatives.

4.1.3 Choices

We used the general training framework developed in section 3.1, and the training software architecture described in section 4.1.2. Within the training framework, and based on the image domains of our experiments, we chose certain functions and representations, described in this section. Choices made in testing are described in section 4.2.2.

Features and shape model

In our initial trained segmentation of bladder boundaries in CT scans, we made the following choices.

$\mathbf{S}$ was a closed polyline $\mathbf{S}(u)$, because that is how ground truth was specified (and utilized) in both domains.

Image quantities were observed at a scale s, using a blurred image $\mathbf{I}_s = \mathbf{I} * G_s^2$ gotten by convolution with a Gaussian the size of the image with variance s^2. The shape's relation to the image was summarized by perhaps the two simplest features: image intensities along the shape, $\mathbf{I}_s(\mathbf{S}(u))$; and directional image gradients normal to the shape, $\mathbf{S}^{\perp}(u) \cdot \nabla\mathbf{I}_s(\mathbf{S}(u))$, where $\mathbf{S}^{\perp}(u)$ is the unit normal to $\mathbf{S}(u)$, and gradient in each coordinate is the average of two adjacent differences. These features, measured along $\mathbf{S}$ at one-pixel arc-length intervals quantized to the nearest pixel, can be considered the output of $F(\mathbf{I}, \mathbf{S})$.

Blurring scales of $s = 2$, 4 and 8 pixels were tried. $s = 2$ was chosen as the smallest blur to test on the basis of the observation that there was observable speckling at the pixel level, so individual pixel intensities would fall in a narrower range if they were averaged with neighbors, *i.e.,* low-pass filtered. This narrower range would almost certainly improve the separation between observed feature value distributions for correct and for incorrect contours. $s = 8$ was chosen as the coarsest blur because image structures that would distinguish correct regions from incorrect ones were on the order of 8 pixels wide, and so would be obliterated by more blurring. As it turned out, $s = 8$ already obliterated useful features, rendering an objective function ineffective for segmentation (see the leftmost image in Figure 3.2).

We did not model the distribution of shapes (shape features). We had enough to test; prior shape modeling has been done by others, much more than image feature modeling. But comparisons of such shape distribution models, and investigation of their necessity and effectiveness, have not been done, and such results would be of great interest. We leave it as future work.

Feature distribution model

We chose perhaps the simplest model of the distribution of our features—a multidimensional Gaussian. If the incidence of our chosen feature values in our domain is unimodal, a Gaussian may model it well enough. Specifically, we assumed independent, identically-distributed (I.I.D.) values of intensity $\mathbf{I}_s(\mathbf{S}(u))$ and directional gradient $\mathbf{S}^{\perp}(u) \cdot \nabla \mathbf{I}_s(\mathbf{S}(u))$ at all points along $\mathbf{S}$. Thus, training recovered the parameters of two Gaussian distributions, $N(\mu_{\mathbf{I}},\, \sigma_{\mathbf{I}})$ and $N(\mu_{\nabla},\, \sigma_{\nabla})$. The joint probability of both these quantities at every point around the contour was the modeled probability of observing those features on a shape $\mathbf{S}$ in an image $\mathbf{I}$, *i.e.,* $\mathrm{P}(F(\mathbf{I}, \mathbf{S}))$. This is a product of Gaussians; its negative log is the "image energy"

$$E \;=\; \oint \frac{\left(\mathbf{I}_s(\mathbf{S}(u)) \;-\; \mu_{\mathbf{I}}\right)^2}{\sigma_{\mathbf{I}}^2} du \;+\; \oint \frac{\left(\mathbf{S}^{\perp}(u) \cdot \nabla \mathbf{I}_s(\mathbf{S}(u)) \;-\; \mu_{\nabla}\right)^2}{\sigma_{\nabla}^2} du$$

We also used an energy with one more parameter, the correlation between intensity and gradient at a point, $c_{\mathbf{I}\nabla}$. This is a 2D Gaussian rather than the joint distribution of two 1D Gaussians. We still modeled different points as independent. (Such independence may not hold, but estimating distributions from limited sample data demands the use of limited parameterizations. And we wish to see how well such simple models can do.) This energy was defined as

$$E_c \;=\; E \;-\; 2 \oint \frac{\left(\mathbf{I}_s(\mathbf{S}(u)) \;-\; \mu_{\mathbf{I}}\right) \left(\mathbf{S}^{\perp}(u) \cdot \nabla \mathbf{I}_s(\mathbf{S}(u)) \;-\; \mu_{\nabla}\right)}{\sigma_{\mathbf{I}}\, \sigma_{\nabla}\, c_{\mathbf{I}\nabla}} du$$

Minimizing any of these "energies" is equivalent to maximizing $\mathrm{P}(F(\mathbf{I}, \mathbf{S}))$. We use straightforward gradient descent minimization over shapes $\mathbf{S}$, so we must take the derivative of E with respect to the parameters of $\mathbf{S}$.

To recover the parameters $\mu_{\mathbf{I}}$, $\sigma_{\mathbf{I}}$, μ_{∇}, σ_{∇} and $c_{\mathbf{I}\nabla}$ from training data, points along the ground-truth contour in each $\mathbf{I}_s$ and $\nabla\mathbf{I}_s$ were sampled at intervals of one pixel of arc length, quantized to the nearest pixel location. Each point was treated as an independent sample, in accordance with our (simplistic) I.I.D. assumption. Since samples are not in fact independent, but are correlated according to what image they came from, this resulted (perhaps incorrectly) in shorter contours having less of an effect ("weight") in training.

Reference model

To provide comparison with traditional methods as applied in the same domain, we also used a "traditional" objective function (not based on training) that summed gradient strengths on the shape boundary:

$$E_T = -\oint ||\nabla\mathbf{I}_s(\mathbf{S}(u))||du$$

Thus, although the same features are used, no distribution model is used (except, implicitly, the *a priori* claim that the higher the gradient strength traversed by $\mathbf{S}$, the higher its probability).

Sectored Snakes: A new synthesis

The qualities that training teaches the model to seek may not be uniform everywhere on its boundary. For instance, in abdominal CT scans, the bladder may be in contact with the pelvis on the sides, seminal vesicles (in men) at two regions near the back, the (highly variable) rectum directly behind, and tissue fluid elsewhere (Figure 4.3). In using snakes trained to seek a particular boundary brightness and edge sharpness, we noticed that, unsurprisingly, a snake would behave badly where boundary qualities were locally different. Since these spatial boundary variations are often consistent across images of a particular domain object, it made sense to formulate a snake that learned what to seek separately on separate portions of its length. Furthermore, there may be differing degrees of variability at different places, so the function's sensitivity should vary, providing robustness where wide variation is expected. In this, we followed [18], although they

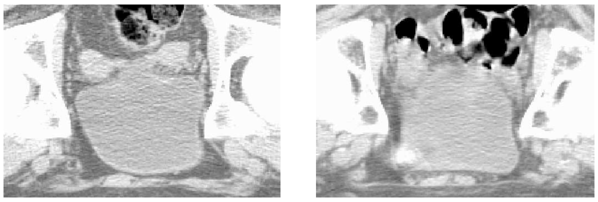


Figure 4.3: *Image variability. These are bladders of different patients. There are parts of their boundary that vary greatly between images, such as the top, pressed against the highly variable rectum. Other parts of the boundary may have very predictable appearance, like the sides, which are always near pelvic bone, and the seminal vesicles, on either side of the top.*

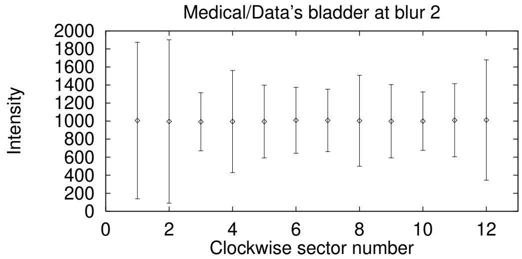


Figure 4.4: *Each sector of a sectored snake has separately learned a joint Gaussian of image intensity (shown here) and gradient in the normal to the snake direction. The intensity bars are tallest in regions 12, 1 and 2, corresponding to greatest intensity variability at the top of the bladder, over 36 images. A snake with this data would not be as strongly attracted to any particular intensity value at the top as elsewhere.*

trained a small number of preselected feature points. We seek an accurate boundary everywhere, so we wish to respond to image data everywhere along the contour, rather than just at a few points. Here we describe sectored snakes, which address these needs within the training framework.

Many approaches to a spatially varying objective function are possible; our first effort was to see if we could produce a measurable improvement. We divided the contour into a fixed number of equal-length regions (sectors), each with separate training (Figure 4.4). Not only would this allow the snake to be attracted to different conditions where different conditions were expected; it also would allow stronger objective function response in regions where there is less variability in conditions between images.

To get regions that roughly correspond among training contours, and between training data and images to be segmented, an origin for the start of the first sector, going clockwise, had to be selected. In many domains, including ours, images and the objects therein may be expected to have a consistent orientation. Thus, we chose "twelve o'clock" to be the highest contour point above the center of mass of the contour (which is quicker and easier to calculate than the center of mass of the enclosed area).

When we minimized our simple, unsectored image energy, we were finding a maximum-probability shape assuming a joint I.I.D. Gaussian distribution of image intensities and perpendicular gradients (at some scale) observed around the contour. In a snake with M sectors, this energy is modified to reflect one of M Gaussian distributions—independent, but not identical—based on position along the contour. Thus, we have a function $k(u)$ which maps the shape parameter u to a sector, $1, \ldots, M$. Instead of a single pair of Gaussians, $N(\mu_{\mathbf{I}},\ \sigma_{\mathbf{I}})$ and $N(\mu_{\nabla},\ \sigma_{\nabla})$, we have a pair for each sector: $N(\mu_{\mathbf{I}}(k),\ \sigma_{\mathbf{I}}(k))$ and $N(\mu_{\nabla}(k),\ \sigma_{\nabla}(k))$.

Our image energy still has a scale parameter s so that the image qualities it responds to are in a Gaussian-blurred image $\mathbf{I}_s = \mathbf{I} * G_s^2$. So our sectored image energy E_S of a contour $\mathbf{S}(u)$ in an image $\mathbf{I}$ at scale s is:

$$E_S = \oint \frac{[\mathbf{I}_s(\mathbf{S}(u)) - \mu_{\mathbf{I}}(k(u))]^2}{\sigma_{\mathbf{I}}^2(k(u))} du + \oint \frac{\left[\mathbf{S}^{\perp}(u) \cdot \nabla \mathbf{I}_s(\mathbf{S}(u)) - \mu_{\nabla}(k(u))\right]^2}{\sigma_{\nabla}^2(k(u))} du$$

where $\mathbf{S}^{\perp}(u)$ is the unit normal to $\mathbf{S}(u)$. The probability we are modeling, $\mathrm{P}(F(\mathbf{I}, \mathbf{S}))$, is proportional to e^{-E_S}.

The number of sectors was picked large enough to capture the number of different types of regions around the object we sought. In our tests, this was the bladder, which has pelvic bone on each side and seminal vesicles (in men) and rectum in back. If the training data were infinite, more sectors would always be better, because their training would approximate a continuously varying probability distribution of image qualities around the shape boundary, and would capture arbitrarily small variations. But since the training data is finite, more sectors also means that each sector gets less exposure to a statistically representative variety of surroundings in the image, and thus poorer training. We chose twelve sectors based on the sizes of structures that seemed consistently positioned around bladder boundaries. Further work may examine the effect of varying the number; sectors of unequal length, perhaps split or merged based on similarity; and continuous parameterizations of the spatially varying probability distribution. For our initial tests, we wanted to see if our intuition was borne out that even a simple sectoring of a snake in a consistent domain would improve our segmentation over that of spatially uniform training.

4.2 Domain performance measurement

We now present our realization of the testing methodology developed in section 3.2. First we describe the software tools created to evaluate the effectiveness of an objective function at segmentation of a chosen class of objects in an image domain; then we run down the procedural choices and parameter values we used for testing in our experiments.

4.2.1 Software tools

In keeping with Unix style, we have separate programs to perform the separate tasks in the testing pipeline. The steps in testing are:

- *Generate parameter tuples of random perturbations (once for all images).* **`perturb-gen`** generates a large set of parameter tuples, each specifying a perturbation of a shape. The command's arguments are the parameters of their random distribution.

- *Generate perturbations from these (for each image's ground-truth shape).* This is done by a library routines used in each of the next two tools. A snake is perturbed by transforming its control points, which effects the same transformation on the continuous piecewise cubic curve they define (a polyline, in our experiments).

- *Measure their distances to unperturbed shape (for each perturbation).* **`perturb-dist`**, given a ground-truth shape, finds its distance from each of a set of perturbations applied to it.

- *Run the candidate objective function (on perturbed shapes for each image).* **`perturb-eval`** finds the value of the specified objective function, for each of a set of perturbed shapes, given an image and its ground-truth shape.

- *Analyze function-*vs.*-distance scatter plot (for each image).* **`incdist`** measures monotonicity of the data by finding the RMS distance of the plot data to the closest increasing function. The implementation is discussed below. Standard **statistical and plotting tools** and command scripts calculate false positive rate per image (given ground truth objective function value) and correlation between shape distance and objective function value; and produce scatter plots.

- *Aggregate performance results from separate images.* Existing tools aggregate these statistics over the set of test images.

The model of how to perturb ground truth shapes, and with what parameters, is available to all tools via a library function.

`perturb-do` generates a particular perturbation of a ground-truth shape, for inspection along with an image, to help understand what shapes in the image incorrectly produced high or low objective function values. Such a perturbation can be picked from a scatter plot showing a nearly-correct shape with a poor score, or one distant from ground truth yet scoring well. Seeing what shapes foil the function helps to explain—and fix—its failures.

4.2.2 Choices

The perturbations we used for objective function evaluation were a combination of translation and of scaling independently around two randomly chosen orthogonal axes centered on the contour's centroid. The translations, and the logs of the scale factors, were normally distributed. The extent of the perturbations can be seen in Figure 3.4. We did not try local perturbations, and we do not yet know whether these (or others) would reveal objective function misbehavior that went unnoticed.

The magnitudes of translation and scaling were based on how far initial approximations to object shape could reasonably be expected to be. Optimization's search window around the initial guess is limited—in the case of gradient methods, by objective function gradient monotonicity; in the cases of dynamic programming and greedy algorithms, by an explicit window size that limits search to something tractable.

The objects whose outlines were sought in our experiments were inner heart wall, with widths of 40–120 pixels and heights of 30–90 pixels; and bladder, with widths of 70–90 and heights of 70–100. We considered it reasonable to cause translations by more than 5 pixels (5%–10% of the object's size) 32% of the time (the area outside one standard deviation), and likewise cause scaling by more than 10% to happen 32% of the time. Thus, the translations were given a standard deviation of 5 pixels; the logs of the scale factors were given a standard deviation of $\log(1.1)$, or 10% stretch/shrinkage.

As a measure of difference between two boundaries, we chose the chamfer distance (see Section 3.2.3). Although this measure is asymmetric, in our CT domain it has a high correlation (0.977) between $d(a, b)$ and $d(b, a)$. (This correlation is the mean for our test set of 36×1,000 perturbed contours; its standard deviation is 0.005.) This measure, the average distance of one shape to another, seems likely to approximate a doctor's characterization of closeness of fit. (No perceptual experiments to verify this were done, however.) Our measure is general, not tailored to the domain, because we did not investigate what kinds of shape difference are more and less important to doctors in our two domains.

In each of our two domains, we tested twelve different objective functions. There were four different feature models—the traditional untrained snake; one using Gaussians modeling intensity and directional gradient; the same with covariance; and the same with 12 independently modeled sectors. Each was tested at three scales, blurring with Gaussian kernels of σ=2, 4 and 8 pixels. All tests used the same 1,000 parameterized perturbations.

Results were statistically analyzed using the measures described in Section 3.2.3:

- Correlation coefficient to measure clustering around a line as an indicator of monotonicity of candidate objective functions with respect to shape distance from ground truth;
- The RMS distance of this function to the *closest* increasing function, as a more reliable measure of monotonicity, as described in Section 3.2.3;
- False positive rate to measure candidate function's optimality with respect to ground truth. We unfortunately had no data on ground-truth variability. With such data, we could establish a shape distance tolerance within which a perturbation is not "false," and therefore may legitimately score better than the supplied ground truth shape;
- Cumulative probability distribution of fraction of images falling within given false positive rates, to flag pathological images;
- Visual inspection of plots.

Chapter 5

Experiments

To the best of our knowledge, this is the first formal measurement and comparison of the effectiveness of deformable model energy functions. Here we describe the image (and task) domains in which we evaluated the performance of deformable models at finding object boundaries. We then discuss the experimental protocols. Finally, we present the results of the experiments.

5.1 Domains

To see how the suitability of given kinds of training (or no training) might vary with circumstance, experiments were conducted in two major application domains. The same set of objective function models was tested in each domain.

The domains we chose had to have certain data available to us—a reasonably sized sample of images, drawn (more or less) randomly and independently from a population of such images used in a particular process for a task involving boundary finding. Each image had to have the ground truth shape of the desired structure available, to within tolerances acceptable to human engineers (here, in both cases, doctors).

5.1.1 Bladder in abdominal CT

In this domain, object contours in the human abdomen are needed to plan radiation treatments which will destroy tumors in the prostate. In such treatments, shaped beams of radiation converge on the tumor so as to give it maximum exposure while giving surrounding healthy tissue minimal exposure. Before beam shape and direction can be optimized, the layout of all nearby organs must be known. Each of these is traditionally outlined manually, slice by slice, on CT scans, sometimes with the help of simple edge-following algorithms. This is a slow, costly and often inaccurate process. A repetitive task on which human life depends could use some verifiably accurate machine assistance. Automated segmentation would help greatly, but, because the image is crowded and complex, it is difficult.

Automated determination of organ boundaries in this domain is difficult for several reasons: boundaries are often fuzzy, and sometimes merge with those of neighboring objects, which may be the same color; edges of neighboring objects, such as bone, may be much stronger; and different points along a boundary can have very different neighboring colors and patterns, as can a given portion of a boundary from one image to the next.

Because anatomic structures are outlined manually for treatment planning, there is a large body of hand-segmented ground truth in this domain. This data may be used to train deformable contour models and to test the accuracy of the resulting models. For our experiments, we concentrated on center slice or slices of the bladder. Although this was a small subset of available images, it maintained the critical temporal and interpatient variations we were interested in. The possible proximity of a tumor gave our training data the same incidence of organ abnormalities as would be seen in the application of training results in the domain. Our images were from a study at Memorial Sloan-Kettering Cancer Center in which patients had four images taken at different stages of treatment for prostate cancer.

Upon investigation, it appeared that the "ground truth" available to us was not drawn very accurately (Figure 5.1); it is, on average, four pixels away from the visible organ boundary, in a 512×512 image. One might think that this casual approach to accuracy would make it easier for our segmentation algorithms to match human performance, which currently suffices in this application. But a poor objective function in a snake segmentation does not necessarily yield small inaccuracies within the range of ground truth variation; rather, it will produce contours that follow entirely the wrong organ in places. And inaccurate data is a disadvantage for us because it cannot provide training that will result in accurate segmentations.

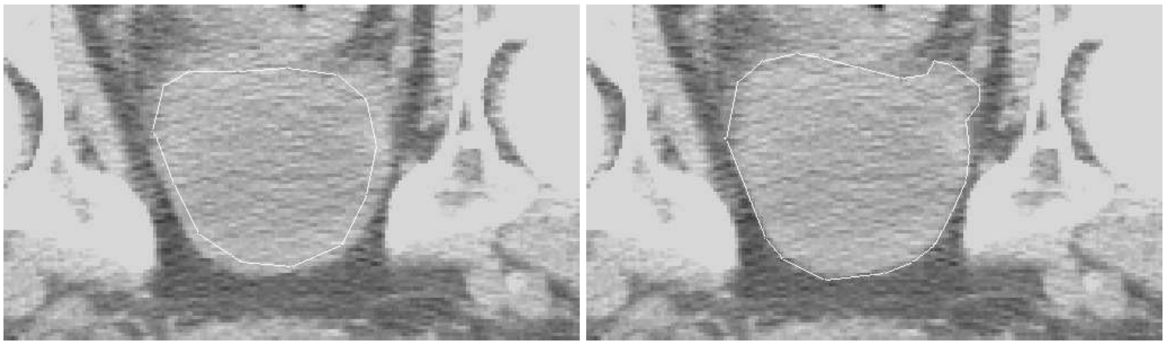


Figure 5.1: *Medical personnel did not draw contours to pixel accuracy (left). Learned segmentation can be no better than the training data, so we redrew the contours we trained on (right).*

To get accurate training data, we drew our own contours, with agreement from doctors that we could do better than the ones they routinely drew. (Economics and time constraints prevent them from doing better.) For greatest consistency, and thus trainability, we trained and tested contours from the organ's middle slice (or two, if there were an even number), where each organ had four to 14 slices (averaging between nine and ten). Up to four 3D images from each of seven patients yielded 24 organ images in 3D, upon whose middle slices we drew 36 high-quality bladder contours. We used polylines of 21 to 51 points—33 on average—for contours of 184 to 339 pixels, averaging 304. (The doctor-drawn versions used from 11 to 22 points, excepting one that used 42; 17 points were used on average. Lengths varied from 149 to 496, averaging 272.)

Both grey-level intensity values and image size are calibrated, and consistent between images from the GE 9800 CT scanner. Doctors regularly assume that particular, narrow grey-level ranges indicate particular materials (water, bone, tissue). In all images, a pixel's width was 0.0939335 cm. of tissue.

5.1.2 Heart ultrasound (echocardiograms)

The other domain in which we pursued experiments was in ultrasound images of the heart (echocardiograms). Each image is a time sequence of frames. The moment of maximum expansion of the ventricles is called end-diastoly (the expansion phase is called the diastole) and the moment of maximum contraction is called end-systoly. The time sequence is long enough to include both. The echocardiogram is used to diagnose and measure abnormalities in the heart muscle or function by tracing the motion of the inner wall of the ventricle.

The quality of these images is much poorer than of the abdominal CTs. Regions are not of homogeneous color but are rather characterized by "speckle," fuzzy white blobs whose major axes are perpendicular to the direction toward the sensor; they result from interference from reflected coherent sound waves. Speckle is present everywhere but concentrated at structure boundaries, especially those perpendicular to the sensor direction. Image clarity varies over the cycle of a heartbeat, and is best at maximum contraction.

It was much harder for us, as non-experts, to interpret these images than the abdominal CTs, so even if we had known the degree of accuracy (or variability) of the expert-drawn contours, we would not have been able to redraw them any better.

We trained deformable models on expert-drawn ground truth boundaries of the inner heart wall at end-systoly. Available views included short-axis and longitudinal; we confined ourselves to short-axis, in which the boundary does not fade into other structures. The supplied boundaries were made into 100-point polylines during the drawing process. Lengths varied from 122 to 360 pixels, averaging 233.

The 320×240 images we used, from Dr. Jeffrey Weisman of Echovision, were preprocessed using a despeckling algorithm. This made the images clearer to the eye, but probably worked to our disadvantage in doing training, as grey levels were adjusted via an expert-chosen one of three (unknown, proprietary) precomputed mappings.

Images were from 24 patients with varying heart abnormality.

5.2 Protocol

As described above, our training data in the abdominal CT domain were 36 hand-outlined 2D images of bladders from 24 abdominal 3D CT scans of seven patients; and for echocardiograms, we had 24 end-systole images.

Training and segmentation was done for four different snake formulations described in Section 4.1.3: the sectored snake; the simple trained model; the model with covariance; and a traditional, untrained snake whose energy is gradient strength traversed by the contour. Each of these was tested at three scales, $s = 2$, $s = 4$ and $s = 8$, for a total of twelve different objective functions.

We evaluated each of these energy functions, using the procedure described in Section 4.2, with 1,000 perturbations of the ground truth contour in each of the test images.

The efficacy of the different models of training was evaluated using “jackknife testing.” This means that performance in each ground-truth image was tested using models trained on all images but that one. This approach allowed us to maximize the (limited) amount of training data, while not biasing results by segmenting images that the learning algorithms had already seen answers for.

Below we look at three statistics of the results, as described in Section 3.2.3. The first is the number perturbed contours whose image energy is lower than that of the unperturbed ground truth contour (“false positives”). We would like this number to be as close to 0 as possible, as we had no information about tolerance in the ground truth, in either domain. The second is the degree of correlation between a contour’s image energy and distance from the image’s ground truth contour. A correlation coefficient close to +1 indicates an objective function which is mostly increasing, but a convex or concave monotonically increasing function will produce values less than 1. The third statistic is our measure of monotonicity, normalized distance to closest increasing sequence, for which lower values are more desirable.

Since these statistics are measures of performance in a single ground-truth image, *i.e.,* a function of a single scatter plot, we characterize the function’s performance in the domain by giving each statistic’s average, average deviation and confidence interval over the entire ground truth test set.

In addition to these statistics on shape incorrectness *vs.* objective function value, visual inspection of the scatter plots, in their full 2,000-dimensional glory, was an important analysis tool.

5.3 Results

In this section we report the results of our training and performance evaluation. For all of the performance measures reported here, we give:

- the measure for each objective function at each scale, averaged over the test images;
- the measure's average deviation over the test set;
- its 95% confidence range.

Numbers are reported to two significant digits.

The average deviation is the average of absolute differences from the observed mean, over the values of the performance measure on each test image in the domain. It is like the standard deviation but is more robust to outliers. We chose it because we suspect our sample populations, small as they are, of containing pathological cases, *i.e.,* not being "normal" in distribution. This suspicion is strengthened by the fact that the average deviation was generally 10% to 25% less than the standard deviation.

The 95% confidence range for each measure is simply the difference from its observed average within which the average for the entire (infinite) image domain is 95% certain to fall (assuming a normal distribution of values for the performance measure). Confidence intervals are based on standard deviation, not average deviation. They are worst-case, because if the measures are not in fact normally distributed, a thick tail caused by outliers may make them larger than actually needed for 95% confidence.

5.3.1 Bladder

Here we examine the results of learning and testing the various objective functions on CT images of the bladder. Overall, training provided a big improvement over simple attraction to edges in finding the correct shape.

Training

The simplest form of training used two independent Gaussians, modeling the probability of finding a given intensity $\mathbf{I}_s(\mathbf{S}(u))$ at a contour pixel with blur s, and of finding a given gradient $\mathbf{S}^{\perp}(u) \cdot \nabla\mathbf{I}_s(\mathbf{S}(u))$ perpendicular to the contour. By way of illustration, Table 5.2 shows the distribution parameters μ_I, σ_I and μ_∇, σ_∇ of these features at each scale. It also shows $\rho_{I\nabla}$, the additional parameter used by the model that learns the two Gaussians with covariance.

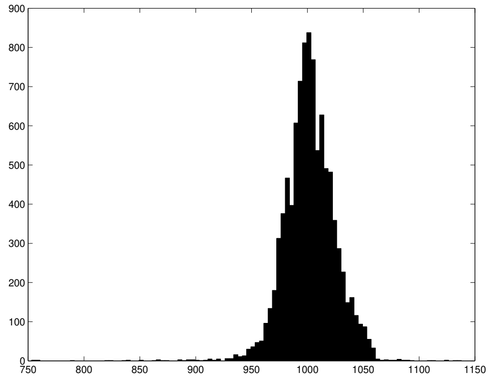
The Kolmogorov-Smirnov test of normality shows that the distributions are not in fact Gaussian (Table 5.1), though they are bell-shaped. Distributions of intensity and directional gradient over the entire test set are show in Figure 5.2.

Bladder CT: Are Blur=2 Intensities *not* Gaussian? Kolmogorov-Smirnov Test

portion of contour	# of pixels	mean	std. dev.	K-S	K-S min for confidence of 90%	95%	99%
whole	9700	1002	23	0.043	0.0082	0.0088	0.010
sector 1	773	1006	29	0.089	0.029	0.031	0.037
sector 2	732	996	30	0.137	0.030	0.032	0.038
sector 3	713	992	18	0.042	0.030	0.032	0.039
sector 4	684	995	24	0.099	0.031	0.033	0.039
sector 5	628	995	20	0.038	0.032	0.035	0.041
sector 6	664	1008	19	0.053	0.031	0.034	0.040
sector 7	811	1008	19	0.022	0.028	0.030	0.036
sector 8	1436	1004	22	0.074	0.021	0.023	0.027
sector 9	976	999	20	0.062	0.026	0.028	0.033
sector 10	775	999	18	0.043	0.029	0.031	0.037
sector 11	766	1010	20	0.025	0.030	0.031	0.037
sector 12	742	1012	26	0.041	0.030	0.032	0.038

Table 5.1: *The Kolmogorov-Smirnov test simply returns the maximum difference between two cumulative distributions. Here we test our bladder contour intensity data against Gaussians with the same mean and variance. Although the distributions are bell-shaped, K-S values usually exceed 99% confidence that distributions are* not *Gaussian.*

The variance of the gradient is high at coarse scales, which would seem to make the gradient an insignificant contributor to the objective function (knowledge of gradient strength at coarse scale should not affect the estimated probability of a shape's correctness). But at the coarsest scale (8), the function that modeled the covariance of the intensity and the gradient produced half the false positives of its closest competitor at that scale, meaning that information was present in the gradient's high correlation with intensity, $\rho_{I\nabla}$, despite the gradient's high variance on its own.



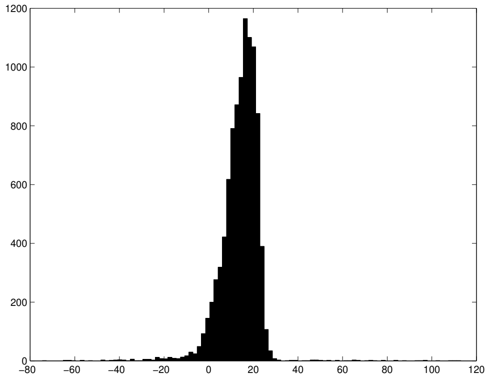


Figure 5.2: *The training we tried was all based on observed boundary pixel intensities and image gradients perpendicular to the boundary. We trained various Gaussian models on this data. These histograms are of pixel intensity distribution (left) and perpendicular gradient distribution for bladders at blur scale 2.*

Bladder CT: Learned Parameters

Scale	μ_I	σ_I	μ_∇	σ_∇	$\rho_{I\nabla}$
2	1002	23	14	8.9	-0.55
4	1004	26	5.9	7.1	-0.77
8	1010	31	1.6	4.1	-0.86

Table 5.2: *Training was based on a total of 9,700 pixels in 36 contours belonging to 24 images, taken at different times, of 7 patients. All but sectored training was based on the parameters above. With 95% confidence, the observed intensity and gradient means μ_I and μ_∇ represent those of the image domain to within 1/3 of their respective standard deviations.*

False positives

An energy function is guaranteed to produce some incorrect segmentations if it is smaller for some close but incorrect contours than it is for the ground truth. Such "false positives" occurred far less often with trained snakes than traditional snakes—in fact, the incidence was negligible when the least blur was used, for all three varieties of training. By contrast, a snake attracted to the strongest edges incorrectly gave 15% of the perturbed shapes a better score than it gave the ground truth shape. At higher blurs, it did even worse. The average incidences of false positives per image for each energy function are in Table 5.3.

At coarser scales, differences between kinds of training, formerly hidden by close-to-

Performance on Bladder CT: False Positives

Objective function model	scale (pixels)	False positive rate avg %	avg dev	95% conf
Untrained (traditional)	2	15	15	±5.9
	4	23	17	±6.3
	8	32	20	±7.7
Single intensity & gradient strength Gaussians	2	1.1	1.6	±0.98
	4	11	11	±4.1
	8	33	5.8	±2.7
Gaussians in 12 sectors	2	1.1	1.7	±1.1
	4	6.7	7.4	±3.3
	8	27	6.4	±3.0
Gaussians with covariance	2	1.6	2.5	±1.4
	4	6.6	7.9	±3.0
	8	14	12	±4.4

Table 5.3: *Incidence of "false positives" (contours scoring better than the correct one) for different objective functions on bladder images. Rates are out of 1,000 perturbed contours for each image. Each row reports the average and average deviation of this rate over the 36 test images, using the indicated objective function model on images blurred at the indicated scale. The objective function used on each image was trained on the other 35 images.*

Training produces functions vastly closer to achieving optimality at ground truth. Lower blur causes each of the function types to make far fewer mistakes. The sectored snake only outperforms the simpler trained model at coarser scales, which are overall worse.

perfect performance, became apparent. At scale 4, **sectoring** the probability distribution cut errors almost in half, from 11% to 6.7% (*e.g.,* Figure 5.3), showing that it can be important for a segmentation algorithm to take into account consistent differences in qualities at different places on an object. Taking covariance into account likewise improves performance.

Although the coarser-scale energy functions showed correlations with distance to ground truth that were just as good as at the finer scale, they have many more false positives—a factor of ten, for the best trained formulation. This disqualifies scale $s = 4$ for use in segmentation in this domain. The correlation measure is equally good at the two scales, while the false-positive measure is very bad for the coarse scale.

Table 5.4 shows the cumulative distribution of false positives among the images: what

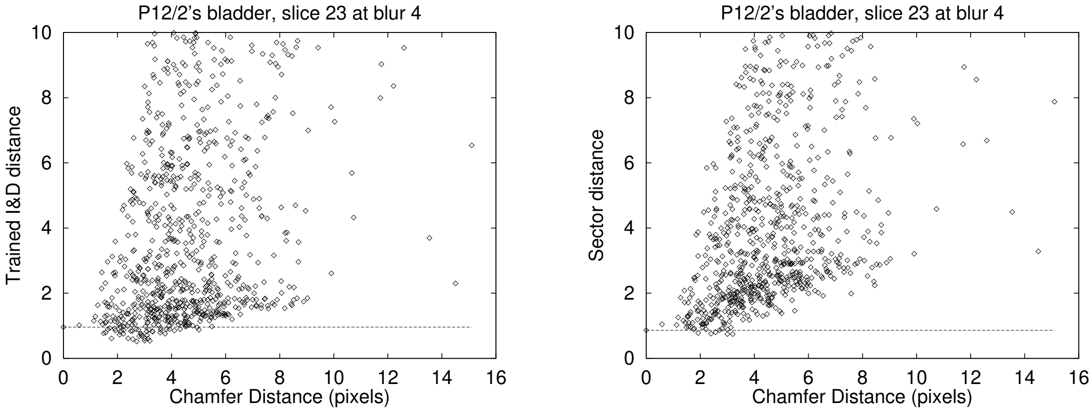


Figure 5.3: *Two objective functions—the result of unsectored and sectored training on the same dataset—applied to contours in the same image. The former has 42 false positives (below the dotted line) in 1,000; the latter has 5. They have comparable measures of increasing monotonicity—correlations of 0.50 and 0.49, and normalized distance from increasing of 0.84 for both.*

portion of the 36 slices met progressively more relaxed standards (number of false positives allowed) for each objective function. For instance, 72% of the image slices had less than 2 in 1,000 bladder contour perturbations with lower sectored-snake energies than the truth. By contrast, only 28% of the images had that few glitches with the traditional snake energy.

Objective function	Scale	no FPs	FPs $<$ 0.2%	FPs $<$ 2%	FPs $<$ 5%
Sectored and trained	2	53%	72%	94%	97%
	4	8%	11%	56%	72%
Unsectored but trained	2	42%	72%	89%	92%
	4	8%	14%	31%	56%
Untrained (traditional)	2	14%	28%	33%	42%
	4	14%	22%	31%	33%

Table 5.4: *How many images did how well, using each candidate objective function: The fraction of 36 images in which fewer than the specified percentage of the 1,000 perturbations were false positives.*

Correlation

Our aim is to check whether contours that are further from ground truth have consistently higher (less optimal) energies. One guarantee of this is if the scatter plot of distance D from ground truth *vs.* objective function value f is tightly clustered around some increasing function. A high linear correlation coefficient $\sum(D_i - \bar{D})(f_i - \bar{f})/n\sigma_D\sigma_f$ is an indicator of this, although a low value does not preclude it. This statistic is shown, for each energy function, in Table 5.5.

Most importantly, all of the trained formulations demonstrated a much higher correlation than traditional (untrained) snakes.

For all objective function models, coarser (blurrier) image scale improved correlation (but devastated false-positive rates). Fine-scale functions only respond to image data very close to the shape; their poorer correlation to distance from correct boundary demonstrates the fact that a snake cannot be attracted to a boundary it cannot "see": some blurring is necessary to make it approach the correct proximity, by seeing a positive correlation (gradient) in that direction; but such blurring will then degrade its precision. The known technique of gradual deblurring to achieve robustness is indicated.

Snakes with sectored training had slightly, but not significantly, higher correlations than unsectored trained snakes, at bigger image scales.

Monotonicity

In this domain, a very nearly inverse relationship was observed between the correlation coefficient and our novel measure of distance from monotonicity (Section 3.2.3). Thus, the function preferences indicated by the correlation coefficient are also indicated by this measure. See Table 5.6.

The consistence of the results also means that, in this domain, both are equally good indicators.

The value of this measure for large datasets with no increasing tendency are extremely close to +1, as is apparent from Table 5.7. (This table is included solely to demonstrate this property of the measure.) It shows monotonicity measures for sequences of objective function values of shapes that are not sorted by chamfer distance from ground truth, and vice versa. The sequences are simply in the random order in which perturbations were generated. The measures are uniformly between 0.99 and 1. Compare to objective functions with significant monotonicity in Table 5.6, which have much lower values.

Performance on Bladder CT: Correlation Coefficient

Objective function model	scale (pixels)	Correlation coefficient avg	avg dev	95% conf
Untrained (traditional)	2	0.15	0.34	±0.12
	4	0.21	0.37	±0.14
	8	0.22	0.38	±0.15
Single intensity & gradient strength Gaussians	2	0.55	0.14	±0.057
	4	0.58	0.14	±0.056
	8	0.44	0.089	±0.040
Gaussians in 12 sectors	2	0.55	0.12	±0.050
	4	0.60	0.14	±0.055
	8	0.51	0.083	±0.038
Gaussians with covariance	2	0.54	0.095	±0.037
	4	0.60	0.11	±0.039
	8	0.63	0.12	±0.045

Table 5.5: *Degree of correlation between contour image energies and chamfer distance to correct shape on bladder images. The learned functions are much better than the traditional snake, and for medium blur, the sectored snake's energy is slightly more correlated to shape correctness than the unsectored snake. Though the medium-blur scale gives a desirable higher correlation within each function class, Table 5.3 show that the bigger blur is to be avoided.*

Inspection of plots

Comparison of scatter plots to images (both on display in Appendix A) reveals a strong correspondence between proximity of neighboring structures and poor behavior of the untrained snake. When bone or muscle is very close to the bladder's boundary, perturbing its correct shape is as likely to make it score better as to score worse. When interfering objects are slightly further away, the scatter plot can actually be seen to arch up, then down again, indicating that the function scores the correct shape higher than shapes close by, but that shapes slightly further away are rewarded for lying on edges of other objects. Such an arch indicates a lack of robustness of the traditional snake to interference from objects commonly found near the bladder. The trained model does not arch back down.

In a few images, the bladder boundary is simply very indistinct. Such images foiled all

Performance on Bladder CT: Monotonicity

Objective function model	scale (pixels)	Obj. function monotonicity			Chamfer dist. monotonicity		
		avg	avg dev	95% conf	avg	avg dev	95% conf
Untrained (traditional)	2	0.87	0.16	±0.058	0.90	0.11	±0.041
	4	0.84	0.20	±0.075	0.85	0.17	±0.062
	8	0.85	0.18	±0.070	0.85	0.17	±0.065
Single intensity & gradient strength Gaussians	2	0.76	0.13	±0.051	0.74	0.10	±0.040
	4	0.73	0.14	±0.052	0.74	0.12	±0.048
	8	0.81	0.099	±0.046	0.83	0.044	±0.020
Gaussians in 12 sectors	2	0.76	0.12	±0.044	0.74	0.081	±0.033
	4	0.71	0.14	±0.054	0.72	0.11	±0.043
	8	0.74	0.10	±0.052	0.80	0.052	±0.022
Gaussians with covariance	2	0.79	0.079	±0.032	0.79	0.063	±0.025
	4	0.73	0.098	±0.037	0.75	0.085	±0.032
	8	0.68	0.13	±0.049	0.71	0.11	±0.041

Table 5.6: *Monotonicity of various objective functions on bladder images, demonstrating essentially the same ranking as Table 5.5. This is the distance from the shape-incorrectness*-vs.-*objective-function plot to the nearest increasing function. It is measured as the root mean square of residuals, and normalized by standard deviation. Thus, the value is zero for a function that is perfectly increasing. The biggest it can get is 1.0.*

The first two columns are the distance from sequence of objective function values, sorted by chamfer distance to ground truth, to the nearest increasing sequence. The second two columns are distance to increasingness for y vs. *x rather than x* vs. *y—contour incorrectness sorted by objective function. As is apparent, the correlation between the two measures is high—0.94.*

Monotonicity of a Random Sequence

Objective function model	scale (pixels)	Obj. function monotonicity		Chamfer dist. monotonicity	
		avg	avg dev	avg	avg dev
Traditional	2	0.9930	0.0019	0.9944	0.00048
	4	0.9945	0.0013	0.9944	0.00048
Single Gaussians	2	0.9968	0.0021	0.9944	0.00048
	4	0.9977	0.0013	0.9944	0.00048
Sectored	2	0.9969	0.0020	0.9944	0.00048
	4	0.9973	0.0014	0.9944	0.00048
With covariance	2	0.9976	0.0014	0.9944	0.00048

Table 5.7: *Monotonicity of* unsorted *shape-incorrectness and objective-function values, presented not as meaningful analysis of the data but for comparison to the values in Table 5.6. The values here are all within 1% of 1.0—the random sequences are almost as far as possible from the closest increasing function. Since the random order of pregenerated perturbed ground-truth shapes is the same for every objective function tested on an image, the second statistic is the same every time.*

objective functions.

5.3.2 Heart

The results of training and testing on echocardiograms follow. Concepts are as explained in the preceding (bladder CT) section. Results were generally poor, because of the quality of the images; still, some useful results emerge.

Training

Table 5.8 gives an illustration of parameters recovered in the simple two-Gaussian training. Variance in both features was huge, meaning that the probability distribution induced by the data was estimated with very poor certainty, relative to the range of values a pixel or gradient can assume. Correlation between the two features, though, was significant. The actual separate distributions which the Gaussians attempt to model are histogrammed in Figure 5.5.

Heart End-Systoly Ultrasound: Learned Parameters

Scale	μ_I	σ_I	μ_∇	σ_∇	$\rho_{I\nabla}$
2	869	428	-52	75	-0.48
4	823	372	-32	42	-0.53
8	764	324	-14	19	-0.48

Table 5.8: *Training was based on a total of 7,073 pixels in 24 contours belonging to 24 images of different patients. The simplest training was based on the parameters above. With 95% confidence, the observed intensity and gradient means μ_I and μ_∇ represent those of the image domain to within 40% of the respective standard deviations σ_I and σ_∇.*

As can be seen, the observed features varied widely between images. This likely explains the poor performance of models trained on this data.

False positives

False positive rates were unacceptable for all functions tested. Untrained snakes, seeking strongest edges, did better than any of the trained models. The moral is clear: in segmentation and boundary-finding tasks, one objective function does not fit all domains.

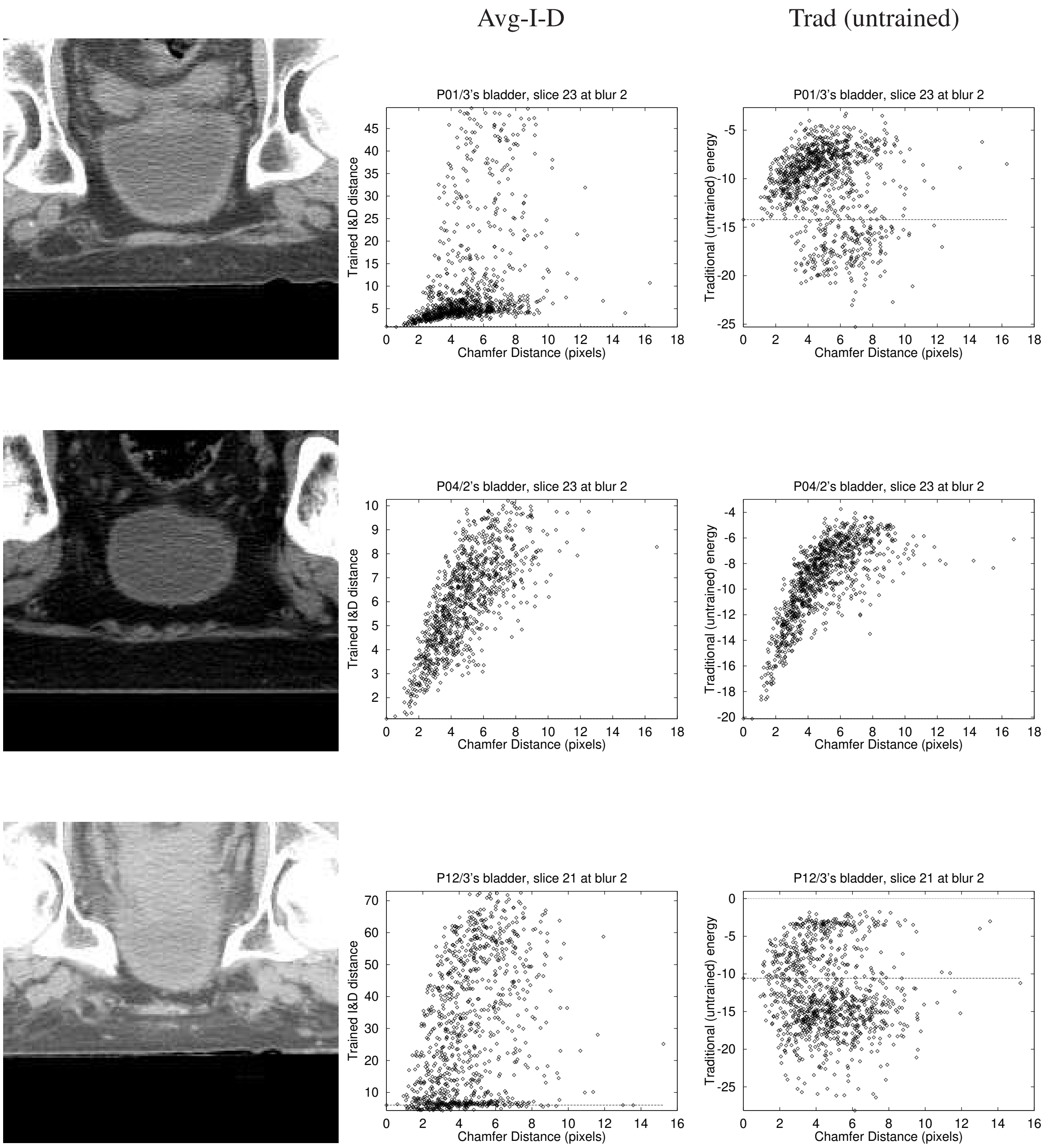


Figure 5.4: *When other objects are close to the desired one, the untrained objective function bows back down as shapes get more incorrect, because they are aligned with edges again, albeit the wrong ones; the trained model can distinguish correct edges. When nothing is very close to the desired object, the untrained and trained models both work well. When edges separating the object from others are absent, both trained and untrained models fail (many perturbed contours score below the ground truth contour; little apparent monotonicity).*

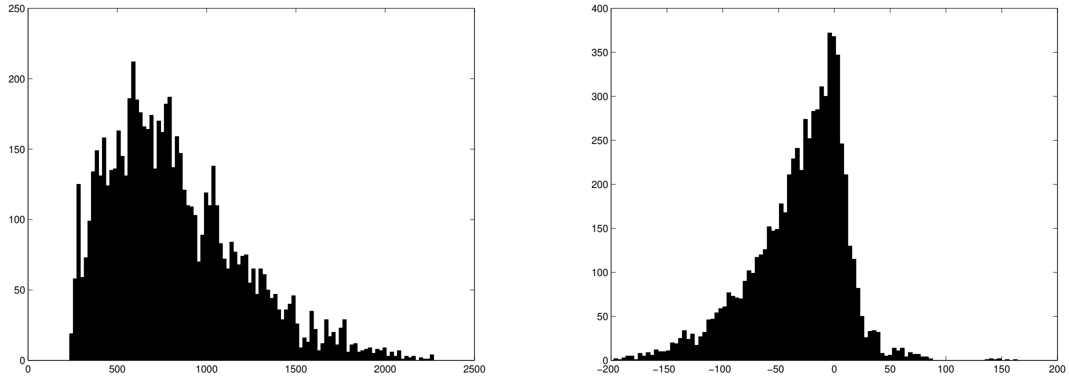


Figure 5.5: *The far-from-Gaussian distributions of contour intensity and perpendicular gradient, in ultrasound images of the inner wall of the heart at end-systole.*

One of the likely reasons that Gaussian training could not recognize correct boundaries is that both intensity and gradient along the desired boundaries are probably bimodal. (Figure 5.5 appears unimodal because it is an aggregate of features in 24 images.) Inspection of the images (Section A.2) shows that boundaries are characterized by bright "speckle," not by perpendicular gradients. This means that intensities along the boundary are alternating dark and light; and that gradient perpendicular to the boundary is not consistent, but rather zero when a speckle blob is intersected, or of unpredictable sign if it is grazed by the contour. Therefore it is not surprising that a nondirectional gradient magnitude would identify the boundary better than something looking for a consistent directional value.

The untrained snake's performance was almost equaled by the learned probability distribution that modeled covariance. The covariant model worked best at a higher blur (4) here than it did in bladder CT imagery; indeed, in stark contrast to that imagery, the finest scale (2) provided no advantage to the trained objective functions in this domain, although the untrained snake at that scale, and only at that scale, beat all other models. Blurring provides an advantage to the Gaussian models, by producing a single average intensity rather than bimodal speckle. No accuracy is sacrificed by blurring, since edges are not well-localized anyway, by the criterion of what the domain expert drew.

Performance on Heart End-Systoly Ultrasound

Objective function model	scale (pixels)	False positive rate avg %	avg dev	95% conf	Correlation coefficient avg	avg dev	95% conf
Untrained (traditional)	2	18	15	±7.0	0.55	0.15	±0.071
	4	32	21	±10.0	0.38	0.23	±0.12
	8	47	18	±8.4	0.11	0.31	±0.15
Single intensity & gradient strength Gaussians	2	31	21	±10.0	0.23	0.26	±0.13
	4	29	20	±9.2	0.34	0.26	±0.13
	8	34	18	±8.5	0.35	0.23	±0.11
Gaussians in 12 sectors	2	29	21	±10.0	0.24	0.25	±0.13
	4	28	21	±9.6	0.33	0.27	±0.13
	8	35	18	±9.0	0.33	0.25	±0.12
Gaussians with covariance	2	27	17	±8.6	0.19	0.23	±0.12
	4	23	16	±8.0	0.34	0.19	±0.11
	8	30	16	±7.9	0.40	0.20	±0.11

Table 5.9: *False positive rate and correlation coefficient of various objective functions on ultrasound images of the heart at end-systoly (maximum ventricle contraction). For reasons discussed in the text, untrained snakes at finer scales performed better than trained models, of which covariant at scale 4 did best. Unlike the bladder CT domain, a coarser scale helped reduce the (still poor) false positive rate, as well as improve the Gaussian models' ability, at larger distances, to decrease when becoming more correct.*

Monotonicity

As measured by distance to nearest increasing function, or by correlation coefficient, the untrained snake at the finest scale scored best of all the models. This may be a result of the fact that the images are already blurry, and edge cues—which in this domain did not help the trained models—get stronger gradually as one approaches the correct shape. Unlike in the CT domain, increasing the blur made the untrained model perform significantly worse, not better.

Again, with the Gaussian trained models, blur helped. At high blur, modeling covariance provided somewhat significant cues as to distance from shape correctness. False positive rate indicated the same thing for the medium and high blurs. Sectoring provided no advantage in this domain, meaning that image qualities did not differ consistently according to what portion of the contour they were on.

Performance on Heart End-Systoly Ultrasound: Monotonicity

Objective function model	scale (pixels)	Obj. function monotonicity			Chamfer dist. monotonicity		
		avg	avg dev	95% conf	avg	avg dev	95% conf
Untrained (traditional)	2	0.78	0.10	±0.049	0.78	0.088	±0.042
	4	0.85	0.12	±0.063	0.83	0.11	±0.058
	8	0.93	0.078	±0.042	0.91	0.079	±0.041
Single intensity & gradient strength Gaussians	2	0.90	0.072	±0.033	0.91	0.064	±0.033
	4	0.85	0.11	±0.052	0.85	0.093	±0.046
	8	0.82	0.13	±0.064	0.83	0.10	±0.051
Gaussians in 12 sectors	2	0.90	0.078	±0.039	0.91	0.066	±0.037
	4	0.85	0.12	±0.057	0.85	0.11	±0.052
	8	0.83	0.13	±0.060	0.84	0.094	±0.047
Gaussians with covariance	2	0.93	0.054	±0.029	0.93	0.052	±0.029
	4	0.87	0.064	±0.032	0.87	0.059	±0.030
	8	0.83	0.10	±0.052	0.82	0.088	±0.047

Table 5.10: *Monotonicity of various objective functions* vs. *distance from the correct shape. See Table 5.6 for explanation. The correlation between the former and latter measures over all ground truth data is 0.943, very close to that for the abdominal CT study.*

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have gathered evaluation data, in a method that should be highly useful to anyone employing deformable models, from datasets relating shape incorrectness to objective function value. Such data showed the effect that scale, and choice of function parameterization, had on the success of segmentation in each of two domains.

The evaluations show us that in our domain of cluttered images, the enhancements we made to snakes were necessary to get good segmentation results. The simple statistics we extracted from energy-distance plots showed that in one of the domains tested, a standard, untrained snake would have failed most of the time, but the trained model would succeed. In the ultrasound domain, more sophistocated features than we tried are needed for acceptable segmentations. Our data also provide evidence that heterogeneous training (sectoring) reduces the learned objective function's false positive rate significantly over simple unsectored training, in a domain (abdominal CT) where there was systematic variation around the object being outlined.

The CT results show much better performance for energies based on the fine image scale than for those on the coarse scale; the ultrasound results show the opposite, demonstrating that domain-dependent testing is necessary to choose the best objective function. Also in domains in which object boundaries were less accurately drawn, or for other reasons not drawn on visible edges, a coarser scale might in fact provide a better correlation between a contour's correctness and image properties along it. This is because image properties on the boundary would be "visible" to the objective function further away if they were blurred.

Unfortunately, although fine scale may provide better discrimination in some domains, when using gradient-descent optimization, coarser scales are necessary to get a nonzero gradient (derivative of energy with respect to shape parameters) when the shape is not extremely close to correct. An unblurred energy quickly reaches zero (plus noise) as a contour departs from the correct shape, so gradient descent will move a contour toward the incorrect solution. Since blurring seems to often prevent energy functions from being able to distinguish right and wrong boundaries, and yet blurring may be necessary for gradient descent optimization, gradual deblurring is indicated—a standard robust optimization technique recommended in [28].

The features and distribution models we selected to train segmentation worked well for abdominal CT scans, but failed for echocardiograms. We see that such selections do not work equally well across image domains. In echocardiograms, our features may be multimodal or skewed, but a multimodal distribution model would not help on its own, because the combined observations from all training images are unimodal. Some per-image transformation is indicated for the features we used, such as normalization by overall image brightness, to make variation of boundary qualities between images be less than between correct and incorrect boundaries within an image. Or different features than boundary intensity and perpendicular gradient must be used, such as texture or vector gradient. Fine-scale features, which worked best for CT, do not always work best in the echocardiograms, where they distinguish not the structure to be segmented, but individual blobs of speckle.

6.1.1 Contributions

We have presented a systematized framework for the training of deformable models based on any chosen shape and image features and objective function model. We hope that this framework will allow unified analysis of the diverse deformable model formulations that have existed up to now. Within this framework, we have presented several results in the segmentation of imagery by means of deformable models.

The first result is straightforward training of a continuous shape model, which appears to be especially useful in imagery characterized by "clutter." Its most telling advantage is that standard approaches would be provably inaccurate in these domains.

The second result is a methodology that allows us (and others) to quantify how good deformable models are, trained or not. The method fully utilizes available ground truth to characterize the model's response to a domain, and its results allow statistical and visual insight into causes of failure. We have tested this approach on what we hope is a believably large dataset of images, using novel but natural criteria for "goodness," and a means of anticipating segmentation failures that improves current practice in both efficiency and thoroughness.

As part of this method, we have suggested measuring a dataset's proximity to the nearest increasing function. Such monotonicity is a necessary condition for correct segmentations using gradient descent.

We have introduced sectoring, a modification of snakes that incorporates the virtues of local adaptability. The method appears to give a quantifiable improvement, at only a small increment in complexity. We have shown that modeling the covariance between different features gives a similar improvement.

The experimental results, using these methods, make clear that performance characterization of a deformable model's objective function is essential; and that the objective function—and any image scale parameters, features and distribution models folded into it—must be tailored to the task domain.

Taken together, we believe these constitute a more robust and more quantitative approach to a difficult but critical image task. This approach can quite easily be shown to improve not only on human results, but on existing computerized approaches.

Since the method is really a class of methods, perhaps the major contribution is that it permits knowledgeable experimentation over many image features, allowing researchers to compare results more accurately, determine causes of objective function failure, and devise more precise measures of model accuracy and reliability.

6.2 Shortcomings of the work

The methods presented here leave some problems unsolved.

First principles do not completely determine the details of the procedures. Several decisions about how to go about training a deformable model and testing it must be made, and depend on expert intuition about the domain images being segmented, and about the application to which the recovered shape boundary is put.

For example, the training methodology does not automatically choose image/shape features, or probability distribution models of them, from the infinite set of possibilities. Candidate features and PDFs, and candidate values of any free parameters, must be chosen by domain experts based on their insight into the domain. These can then be tested and compared.

The testing methodology itself can only proceed once the experimenter guesses the range of incorrect shapes that a deformable model segmentation will try near the correct one during optimization. The experimenter must estimate the worst displacement that initial guesses might have from the correct shape; and also what tractable subspace of deformations of it will reveal most incorrect scorings that candidate objective functions might make. (Is it necessary to introduce/eliminate corners? Wiggles?) And the experimenter must pick a shape distance measure which reflects he acceptability of different kinds of shape incorrectness in the domain. (Displacement? Edge orientation errors? Do such things matter more on some parts of the shape than on others?)

Our work showed the usefulness of generalized training and testing using only simple choices. Image features were the simplest; we used none of the more sophisticated filters available, such as steerable filters. Our probability distribution models were all Gaussian, for no reasons other than unimodality and simplicity. Likewise, during testing, significant information was revealed using only a simple shape distance measure and perturbation model.

We do not claim to know the single best way to quantitatively analyze the performance plots resulting from testing. The statistics we used certainly enabled us to make choices and provided insight, but we are sure better ones exist.

We could have interpreted our results better with additional domain information that we lacked, such as multiple doctor-drawn boundaries for each testing and training image, which would tell us the limits on the accuracy of training and testing based on them [14], and would provide a tolerance within which it is not undesirable for perturbations to score as well as ground truth. More information about the medical processes that use the segmentation results would also determine such tolerances.

Also, doctor feedback on the accuracy of perturbed shapes would allow us to model a domain-dependent shape distance metric, and thus evaluate objective functions better.

There are many additional features, models and statistics that beg to be tried, some of which are mentioned in the next section.

6.3 Future work

We intend to extend the work in several mutually supportive directions:

- The class of image features is easy to extend beyond intensity and gradient measures; in addition to higher-order measures such as texture, this method easily accommodates the use of non-standard measures such as boundary placement relative to distant landmarks. The heart data, whose intensity is uncalibrated, may exhibit more-learnable characteristics if image brightnesses are normalized. And finally, it would be valuable to test the usefulness of the many extant prior shape models.

- We must check if some other probability models fit the distributions of some features better than a Gaussian. For instance, a multidimensional spline could be fit to training feature density data. This, of course, requires even more experimentation on more hand-drawn ground truth; fitting more parameters requires more data.

- The definition of a "sector" needs to be explored, particularly with respect to whether image statistics can specify some "natural" sectoring more robustly than our 12 points of the clock face. Also to be explored are continuously spatially varying training, as opposed to discrete sectors; and the existence of dependence between expected features at different locations.

- In the most ambitious setting, and if large amounts of ground truth are available, this method can allow even dynamic and automatic selection of image features, sectoring, and feature distribution model selection, as a kind of "meta-learning."

Bibliography

[1] A. A. Amini, T. E. Weymouth, and R. C. Jain. Using dynamic programming for solving variational problems in vision. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 12(9):855–867, September 1990.

[2] Harry C. Andrews. *Introduction to Mathematical Techniques in Pattern Recognition*. Wiley, New York, NY, USA, 1972.

[3] M. Ayer, H. D. Brunk, G. M. Ewing, W. T. Reid, and E. Silverman. An empirical distribution function for sampling with incomplete information. *Annals of Mathematical Statistics*, 26:641–647, 1955.

[4] Simon Baker. *Design and Evaluation of Feature Detectors*. PhD thesis, Columbia University, 1998.

[5] Simon Baker and Shree K. Nayar. Parametric feature detection. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, San Francisco, CA, USA, June 1996. IEEE Computer Society.

[6] Bernard Baldwin. *Multiscale Snakes*. PhD thesis, Courant Institute of New York University, 1997.

[7] R. E. Barlow, D. J. Bartholomew, D. J. Bremner, and H. D. Brunk. *Statistical Inference Under Order Restrictions*. Wiley, New York, 1972.

[8] Jennifer L. Boes, Charles R. Meyer, and Terry E. Weymouth. Liver definition in CT using a population-based shape model. In *Proc. 1st Int'l Conf. on Computer Vision, Virtual Reality, and Robotics in Medicine (CVRMed '95)*, pages 506–512, Nice, France, April 1995. Springer.

[9] Gunilla Borgefors. Hierarchical chamfer matching: A parametric edge matching algorithm. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 10(6):849–865, November 1988.

[10] Terrance E. Boult, Samuel D. Fenster, and Thomas O'Donnell. Reinterpreting physically-motivated modeling. In *Proc. ARPA Image Understanding Workshop*, Monterey, CA, USA, November 1994. Morgan Kaufmann.

[11] M. J. Byrne and J. Graham. Application of model based image interpretation methods to diabetic neuropathy. In *Proc. European Conf. on Computer Vision*, volume II, pages 272–282, Cambridge, UK, April 1996. Springer.

[12] A. Chakraborty, L.H. Staib, and J.S. Duncan. Deformable boundary finding influenced by region homogeneity. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 624–627, Seattle, WA, USA, June 1994. IEEE Computer Society.

[13] Vikram Chalana, W. S. Costa, and Yongmin Kim. Integrating region growing and edge detection using regularization. In *Image processing*, volume 2434 of *Medical imaging*, pages 262–271, San Diego CA USA, March 1995. SPIE–the Int'l Society for Optical Engineering, SPIE.

[14] Vikram Chalana and Yongmin Kim. Methodology for evaluation of boundary detection algorithms on medical images. *IEEE Tran. on Medical Imaging*, 16(5):642–652, October 1997.

[15] C. H. (Chi-hau) Chen. *Statistical pattern recognition*. Hayden, Rochelle Park, NJ, USA, 1973.

[16] Laurent D. Cohen and Ron Kimmel. Global minimum for active contour models: A minimal path approach. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 666–673, San Francisco CA USA, June 1996. IEEE Computer Society.

[17] L.D. Cohen. On active contour models and balloons. *Computer Vision, Graphics and Image Processing: Image Understanding*, 53(3):211–218, March 1991.

[18] T.F. Cootes, A. Hill, C.J. Taylor, and J. Haslam. Building and using flexible models incorporating grey-level information. In *Proc. IEEE Int'l Conf. on Computer Vision*, pages 242–246, Berlin, May 1993. IEEE Computer Society.

[19] C. Davatzikos and J.L. Prince. Global minimum for active contour models: A minimal path approach. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 674–679, San Francisco CA USA, June 1996. IEEE Computer Society.

[20] Chris A. Davatzikos and Jerry L. Prince. An active contour model for mapping the cortex. *IEEE Tran. on Medical Imaging*, 14(1):65–80, March 1995.

[21] Douglas DeCarlo and D. Metaxas. Blended deformable models. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 566–572, Seattle, WA, USA, June 1994. IEEE Computer Society.

[22] Davi Geiger, Alok Gupta, and Luiz A. Costa. Dynamic programming for detecting, tracking, and matching deformable contours. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 17(3):294–302, March 1995.

[23] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 6(6):721–741, November 1984.

[24] Robert P. Grzeszczuk and David N. Levin. "Brownian strings": Segmenting images with stochastically deformable contours. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 19(10):1100–1114, October 1997.

[25] André Gueziec. Large deformable splines, crest lines and matching. In *Proc. 4th Int'l Conf. on Computer Vision*, pages 650–657, Berlin, Germany, May 1993. IEEE Computer Society.

[26] D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 15(9):850–863, September 1993.

[27] J. Ivins and J. Porrill. Statistical snakes: Active region models. In *Proceedings of the 5th British Machine Vision Conference*, volume 2, pages 377–386, York, UK, September 1994. BMVA Press.

[28] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. In *Proc. IEEE Int'l Conf. on Computer Vision*, pages 259–268, London, 1987. IEEE Computer Society.

[29] Charles Kervrann and Fabrice Heitz. A hierarchical statistical framework for the segmentation of deformable objects in image sequences. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 724–728, Seattle, WA, USA, June 1994. IEEE Computer Society.

[30] K.F. Lai and R.T. Chin. Deformable contours: Modeling and extraction. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 17(11):1084–1090, November 1995.

[31] F. Leymarie and M.D. Levine. Tracking deformable objects in the plane using an active contour model. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 15(6):617–633, June 1993.

[32] R. Malladi, J. A. Sethian, and B. C. Vemuri. Evolutionary fronts for topology-independent shape modeling and recovery. In *Proc. 3rd European Conf. on Computer Vision*, volume I, pages 3–13, Stockholm, Sweden, May 1994. Springer.

[33] Hiroshi Murase and Shree K. Nayar. Learning object models from appearance. In *Proc. AAAI National Conference on Artificial Intelligence*, pages 836–843, Washington, D.C., USA, July 1993. American Association for Artificial Intelligence.

[34] W. Neuenschwander, P. Fua, G. Székely, and O. Kübler. Initializing snakes. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 658–663, Seattle, WA, USA, June 1994. IEEE Computer Society.

[35] P. P. Ohanian and R. C. Dubes. Performance evaluation for four classes of natural textures. *Pattern Recognition*, 25(8):819–833, August 1992.

[36] A. P. Pentland and S. Sclaroff. Modal matching for correspondence and recognition. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 17(6):545–561, June 1995.

[37] Visvanathan Ramesh. *Performance Evaluation of Image Understanding Algorithms*. PhD thesis, University of Washington, 1995.

[38] Visvanathan Ramesh and Robert M. Haralick. Performance characterization of edge detectors. In *Applications of artificial intelligence X : machine vision and robotics*, volume 1708, pages 252–266, Bellingham, WA, USA, 1992. SPIE.

[39] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, USA, 1995.

[40] Lawrence H. Staib and James S. Duncan. Deformable Fourier models for surface finding in 3D images. In *Visualization in Biomedical Computing*, volume 1808, pages 90–104, Chapel Hill, NC, USA, October 1992. SPIE–the Int'l Society for Optical Engineering, SPIE.

[41] T. Taxt and A. Lundervold. Multispectral analysis of the brain using magnetic resonance imaging. *IEEE Tran. on Medical Imaging*, 13(3):470–481, September 1994.

[42] H. Tek and B. B. Kimia. Image segmentation by reaction-diffusion bubbles. In *Proc. 5th Int'l Conf. on Computer Vision*, pages 156–162, Cambridge, MA, USA, June 1995. IEEE Computer Society.

[43] D. Terzopoulos and D. Metaxas. Dynamic 3D models with local and global deformations: Deformable superquadrics. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 13(7):703–714, July 1991.

[44] D. Terzopoulos, A. Witkin, and M. Kass. Symmetry-seeking models and 3D object reconstruction. *International Journal of Computer Vision*, 1(3):211–221, 1987.

[45] Matthew A. Turk and Alex P. Pentland. Face recognition using eigenfaces. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 586–591, Maui, Hawaii, USA, June 1991. IEEE Computer Society.

[46] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, NY, USA, 1995.

[47] A.L. Yuille, D.S. Cohen, and P.W. Hallinan. Feature extraction from faces using deformable templates. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 104–109, San Diego CA USA, JUN 1989. IEEE Computer Society.

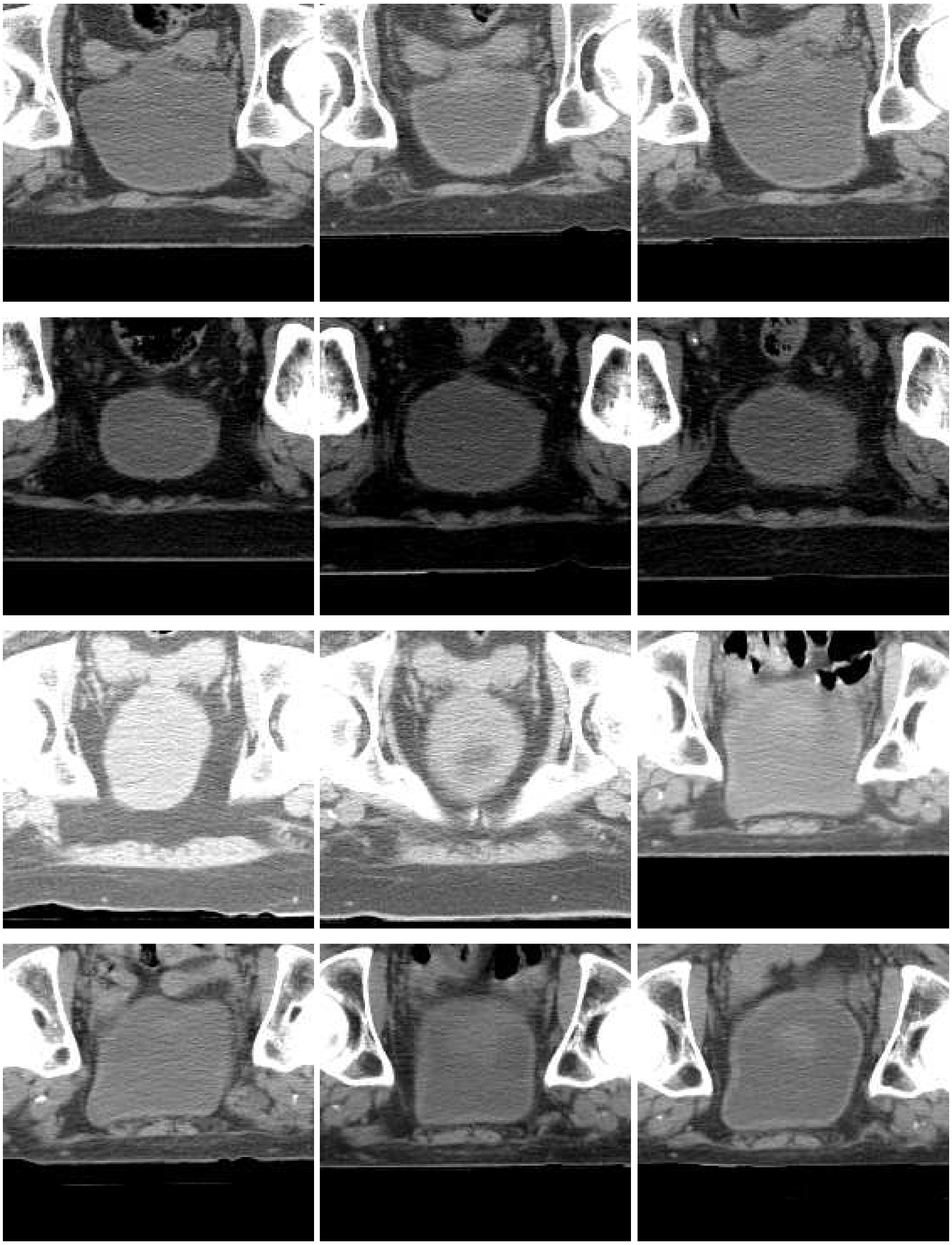
[48] Xining Zhang, Robert M. Haralick, Visvanathan Ramesh, and Anand S. Bedekar. A Bayesian corner detector: theory and performance evaluation. In *Proc. ARPA Image Understanding Workshop*, volume 1, pages 703–715, Monterey, CA, USA, November 1994. Morgan Kaufmann.

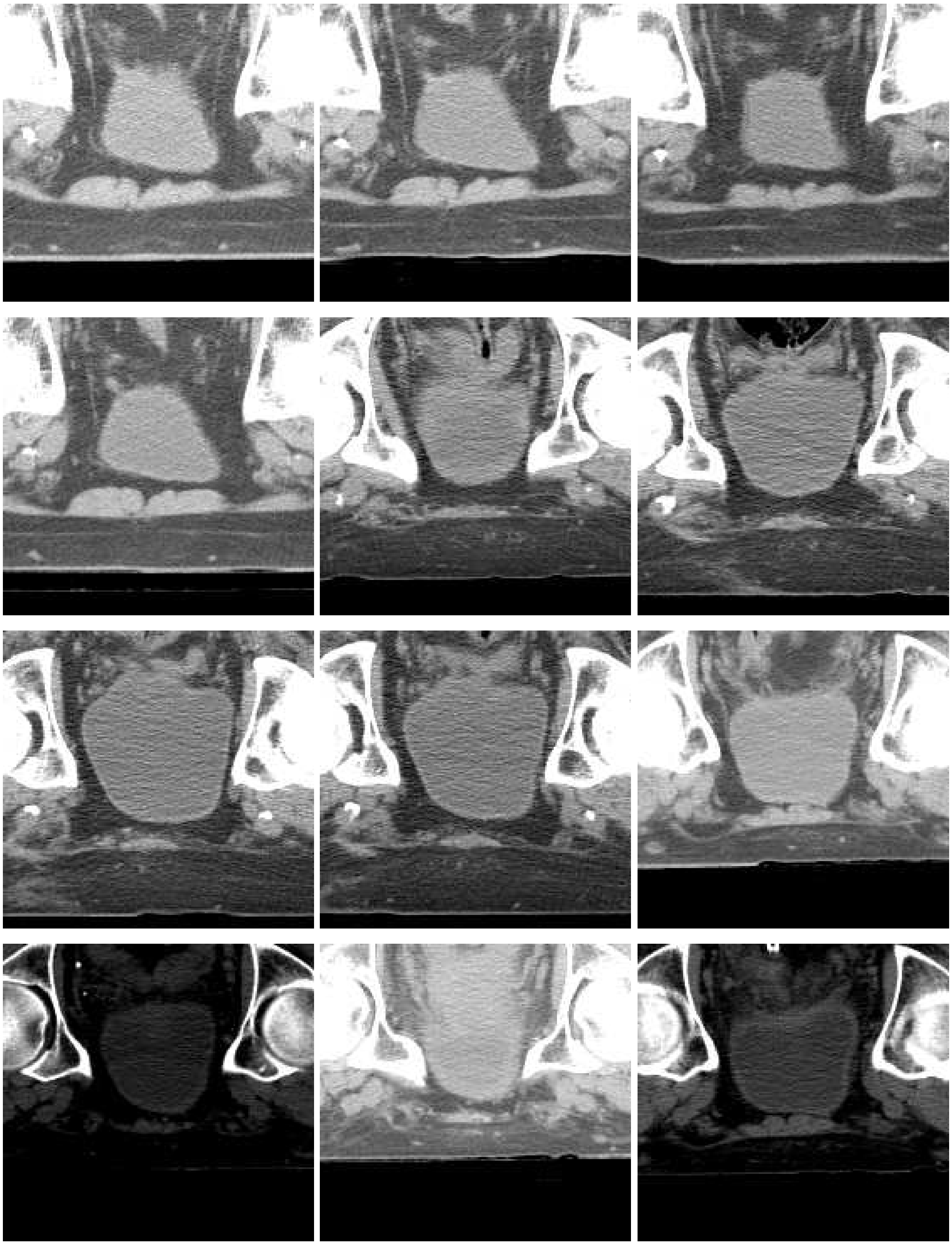
Appendix A

Images, Contours and Plots

A.1 Bladder images

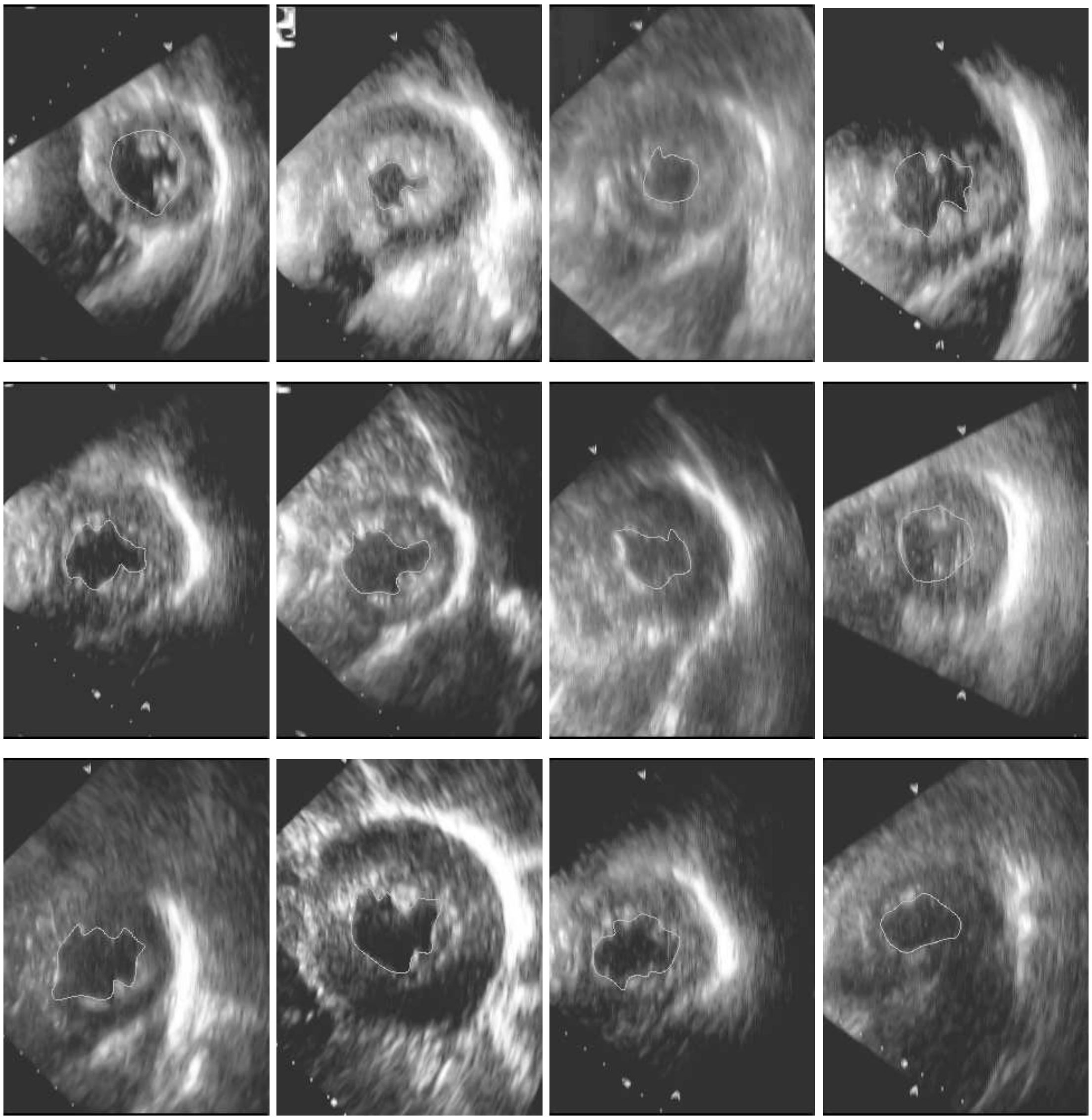
The patients in the study from which we got our bladder images had CT scans taken at four stages of treatment for prostate cancer. Not all patients or all images had the ground truth for the organ we needed available. On the next page are middle bladder slices from each of the scans we used, comprising seven patients. All are at the same scale and brightness level, and all have been cropped. Figure 2.2 shows an example of an uncropped image.

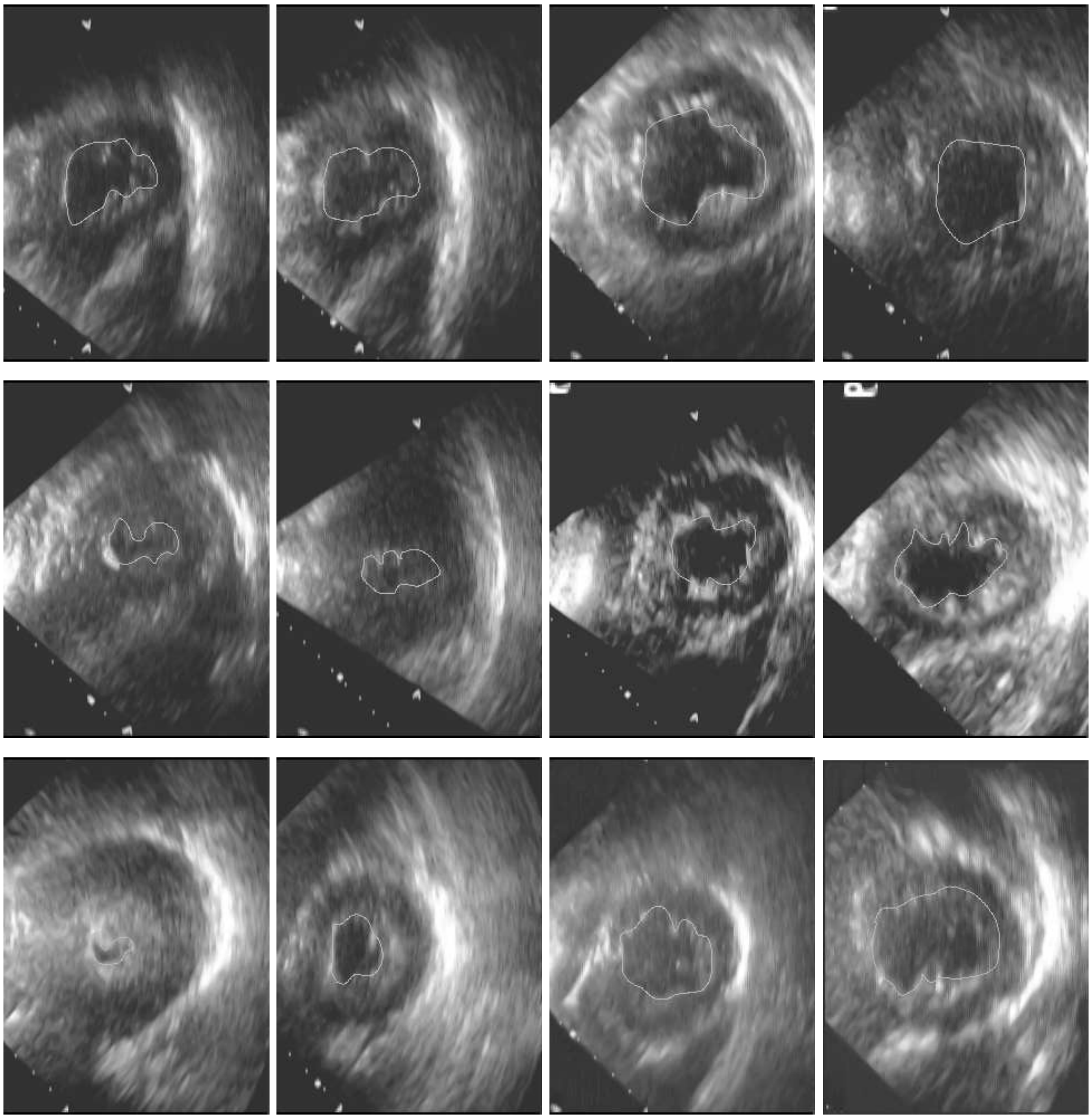




A.2 Heart images

All 24 echocardiograms used in our study follow.





A.3 Bladder and heart plots

Scatter plots characterize the goodness of an objective function on an image. Here we show an instructive subset of the objective functions tested.

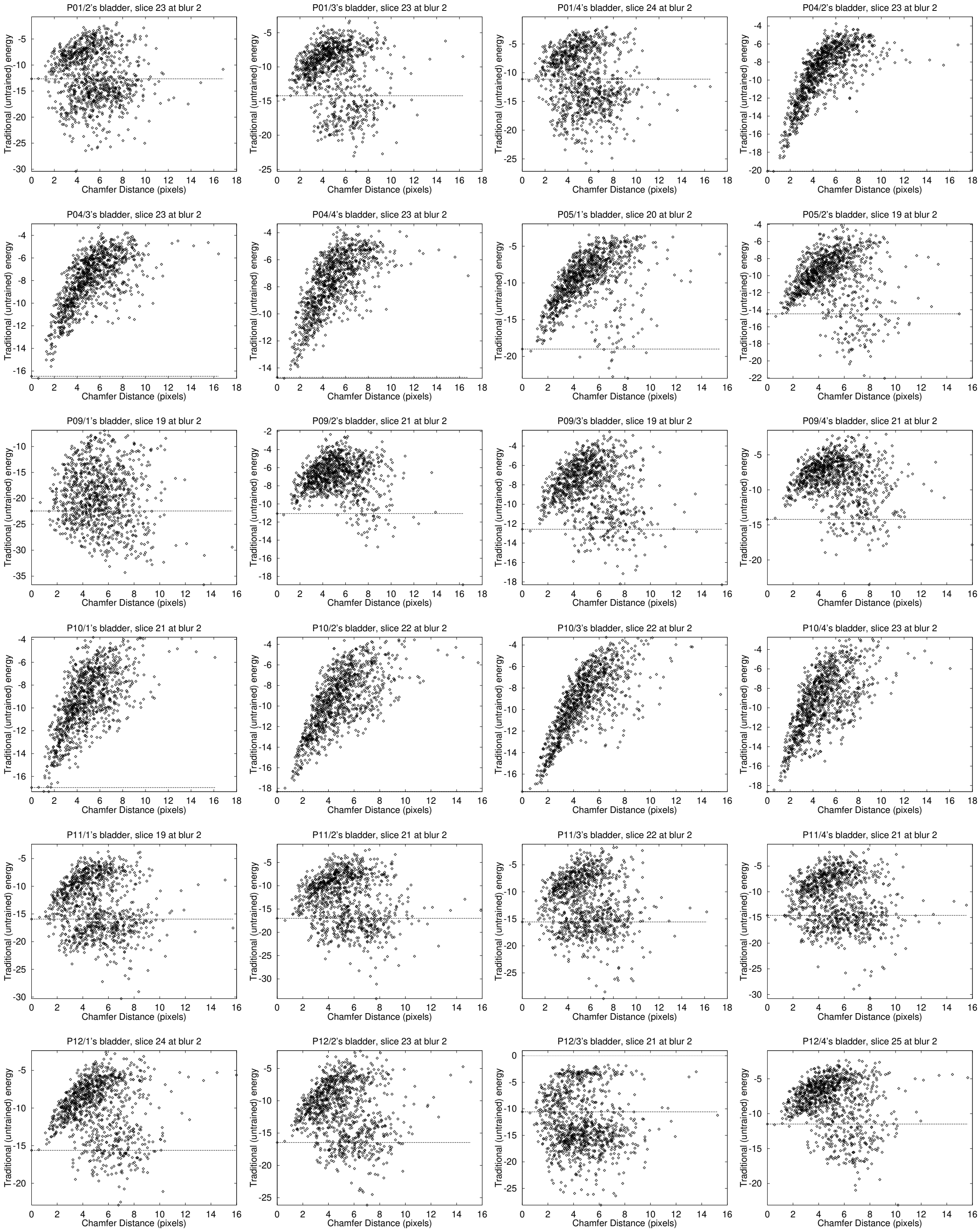
Each bladder page has a plot for one slice from each of the 24 3D CT scans used in our experiments. Each heart page has plots for all 24 heart end-systole images. On each page, reading from left to right, top to bottom, the plots correspond to the images in the previous sections, read the same way.

The following data allows comparison of:

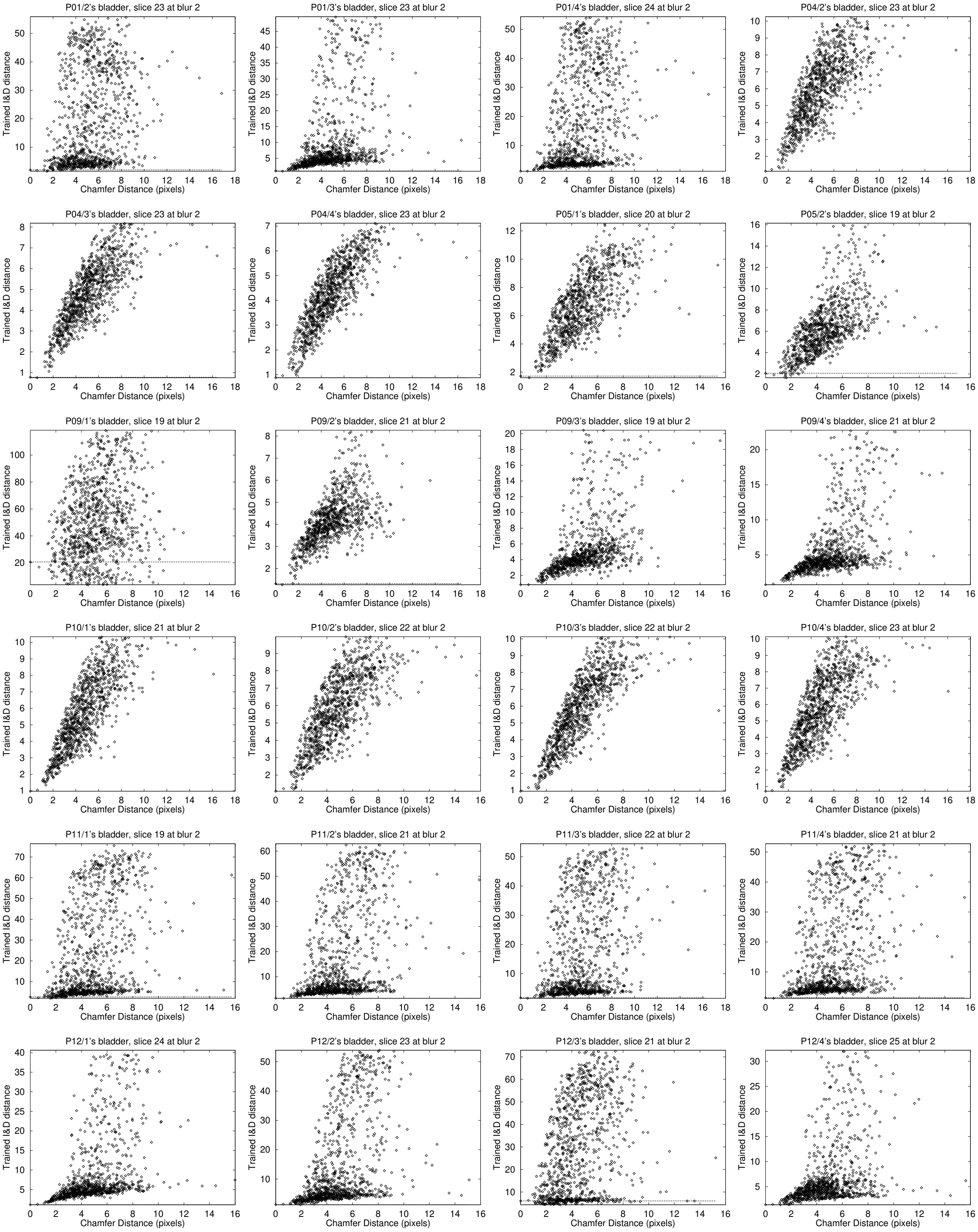
- objective functions in the CT domain:
  - traditional untrained snakes with our simplest trained model;
  - the trained model at two different scales;
  - the simple trained model with the sectored model, which incorporated spatially varying training;
- objective functions in the ultrasound domain:
  - traditional snakes with trained ones;
  - three scales, with the middle one performing best.

Plots are autoscaled because the units of the y axis, or the absolute extent in the y direction of the data, is not important in evaluating an objective function. Performance is affected by fraction of data points below the ground truth score (horizontal dotted line), and by tightness of clustering around an increasing function.

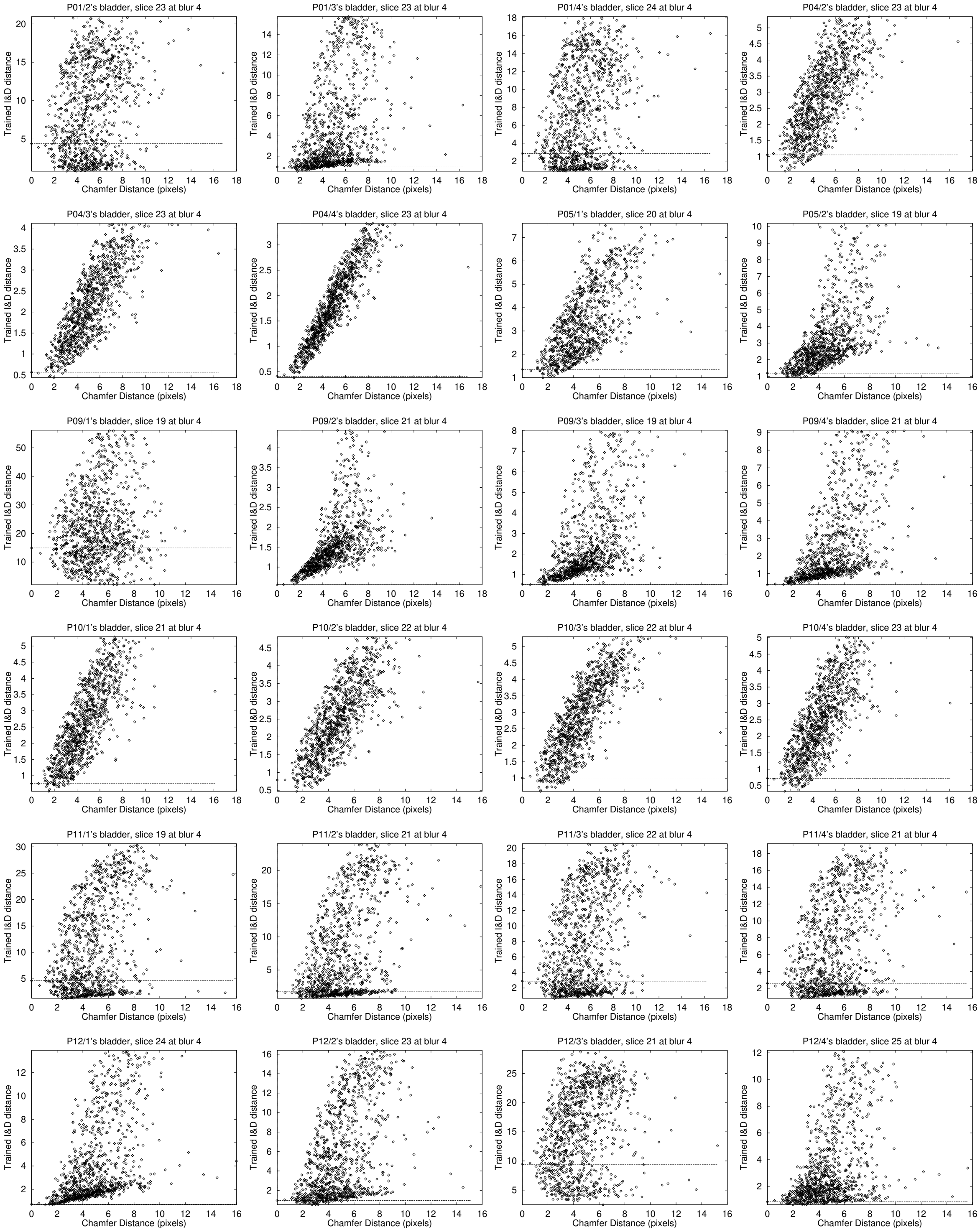
Bladder, untrained, best scale (s=2)



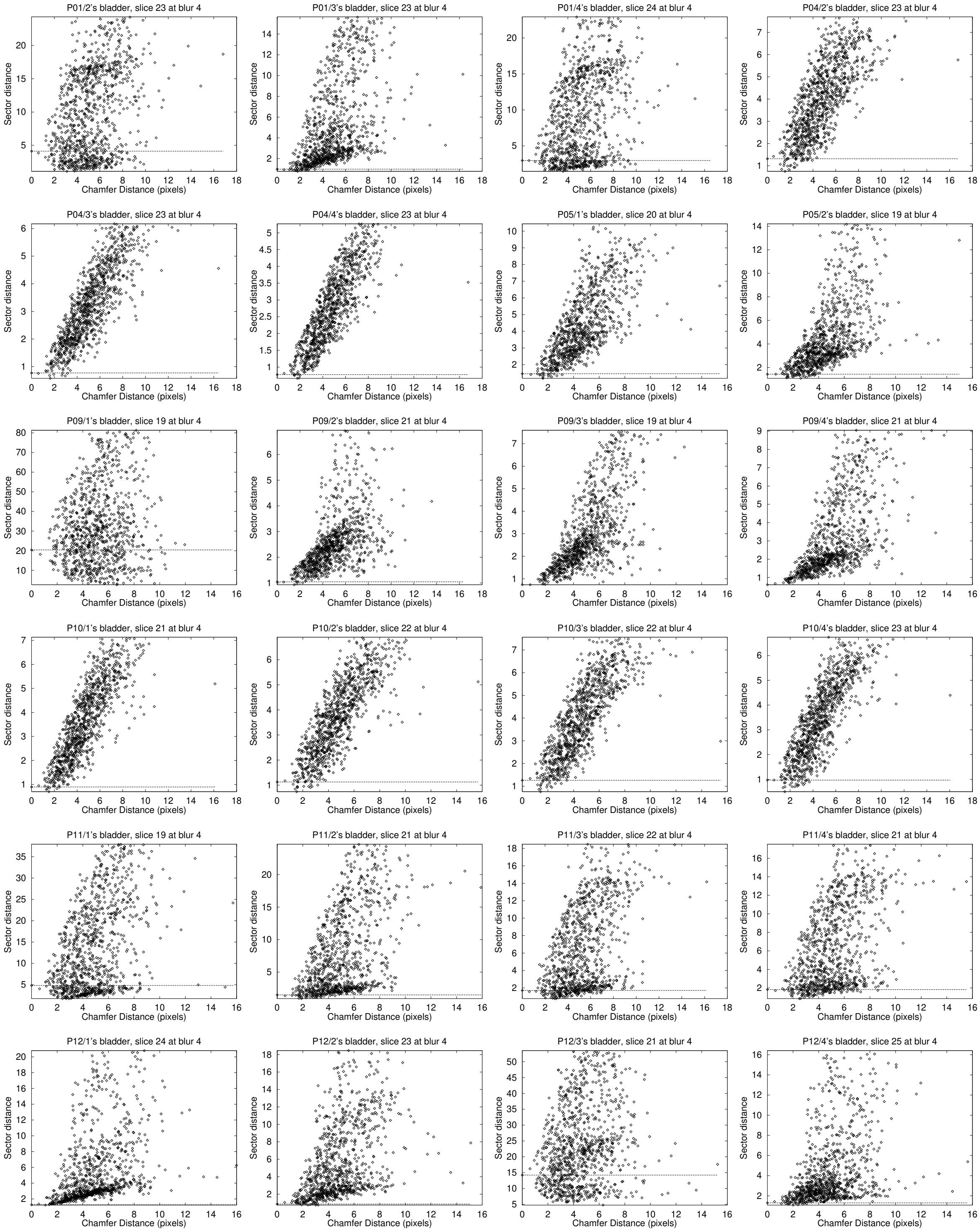
Bladder, intensity and gradient Gaussians, best scale (s=2)



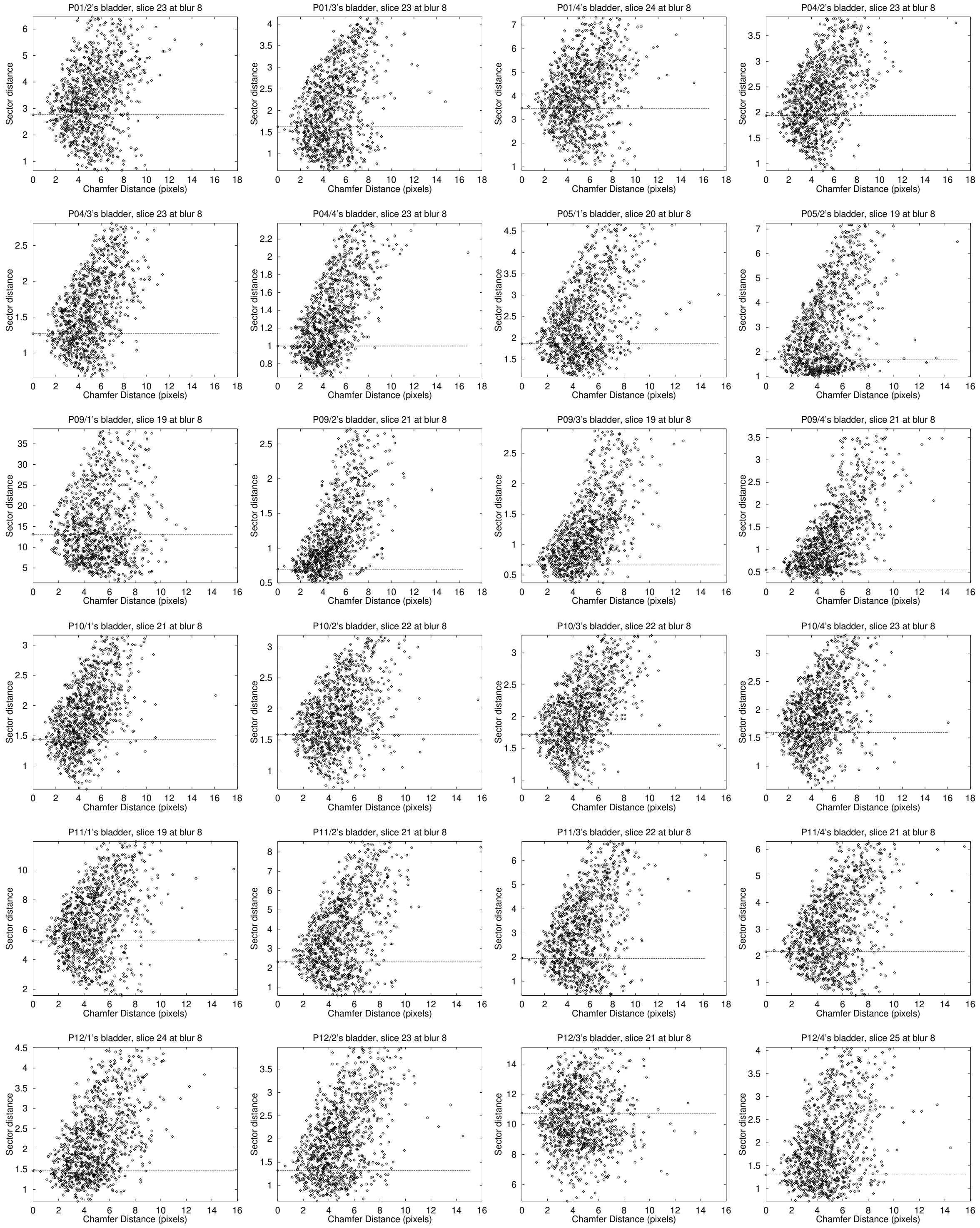
Bladder, intensity and gradient Gaussians, coarser scale (s=4)



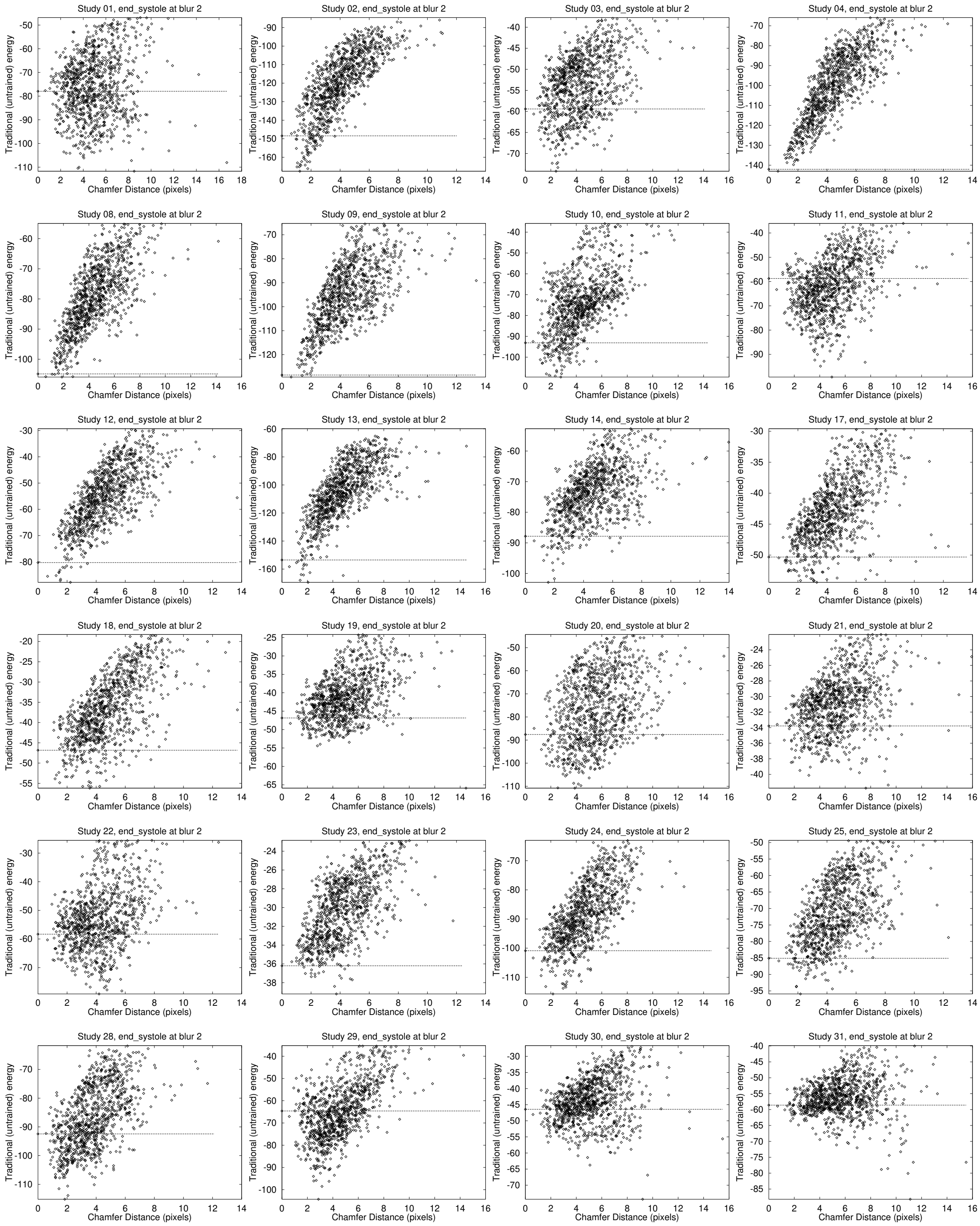
Bladder, Gaussians in 12 sectors, coarser scale (s=4)



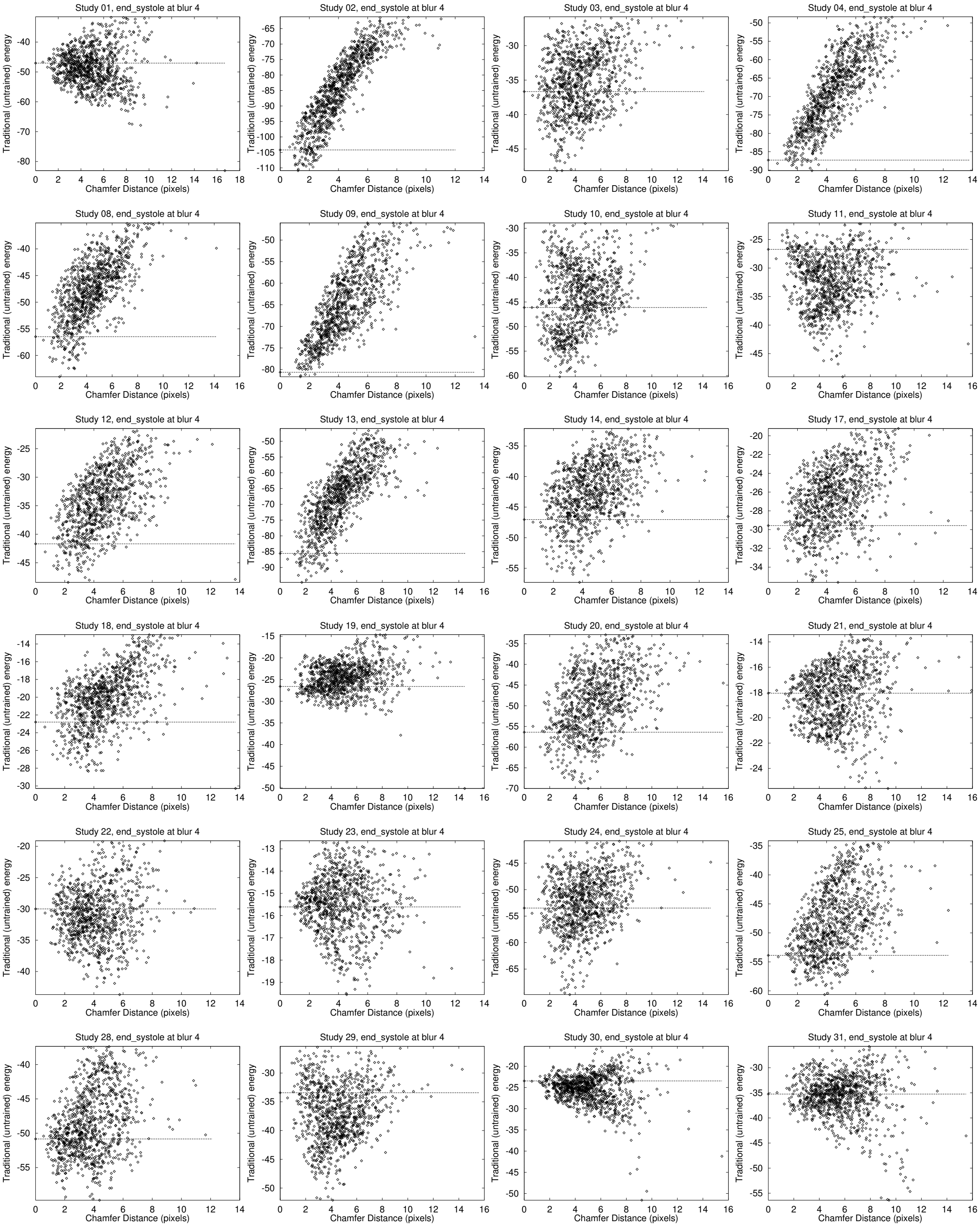
Bladder, Gaussians in 12 sectors, scale 8



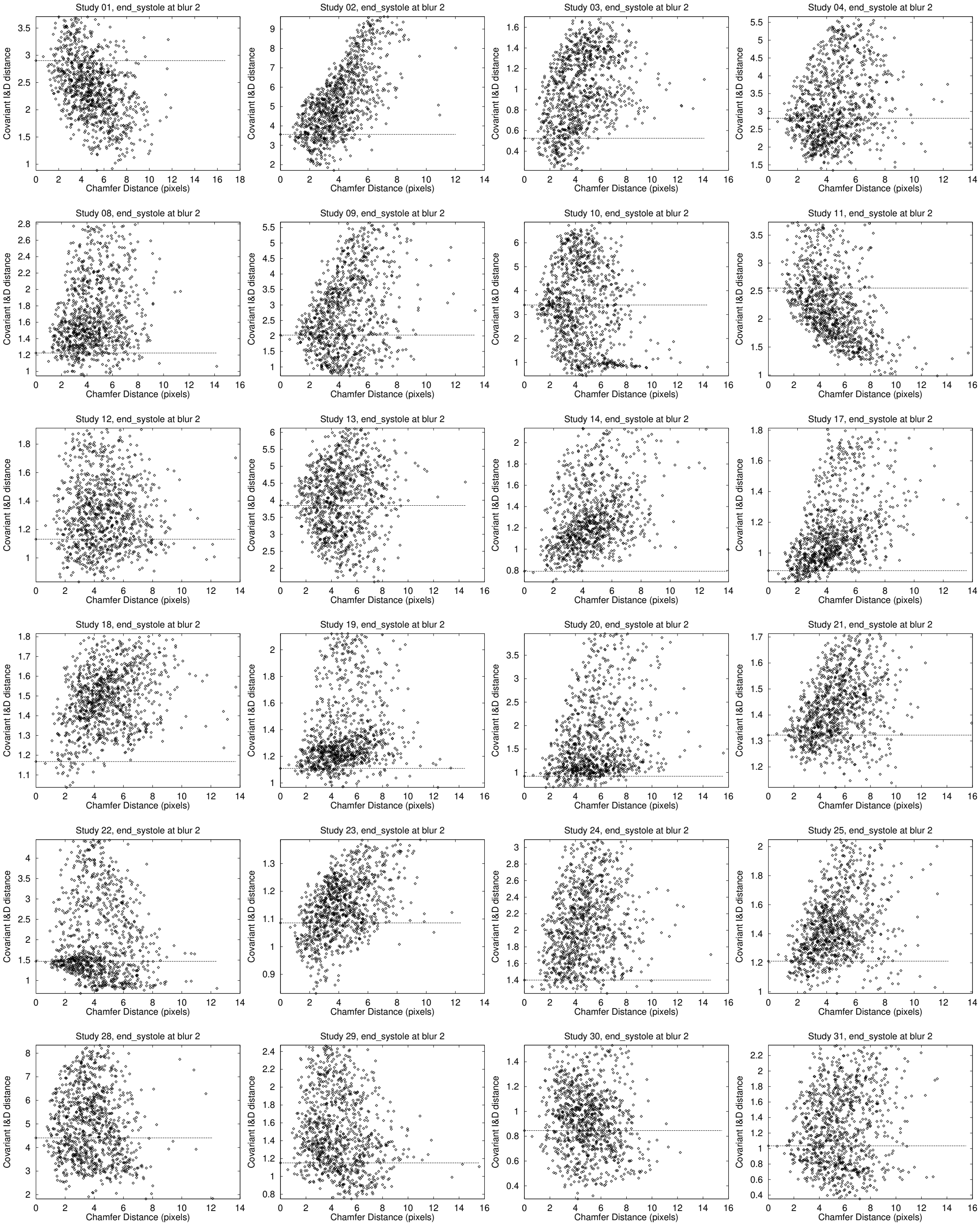
Heart, untrained, scale 2



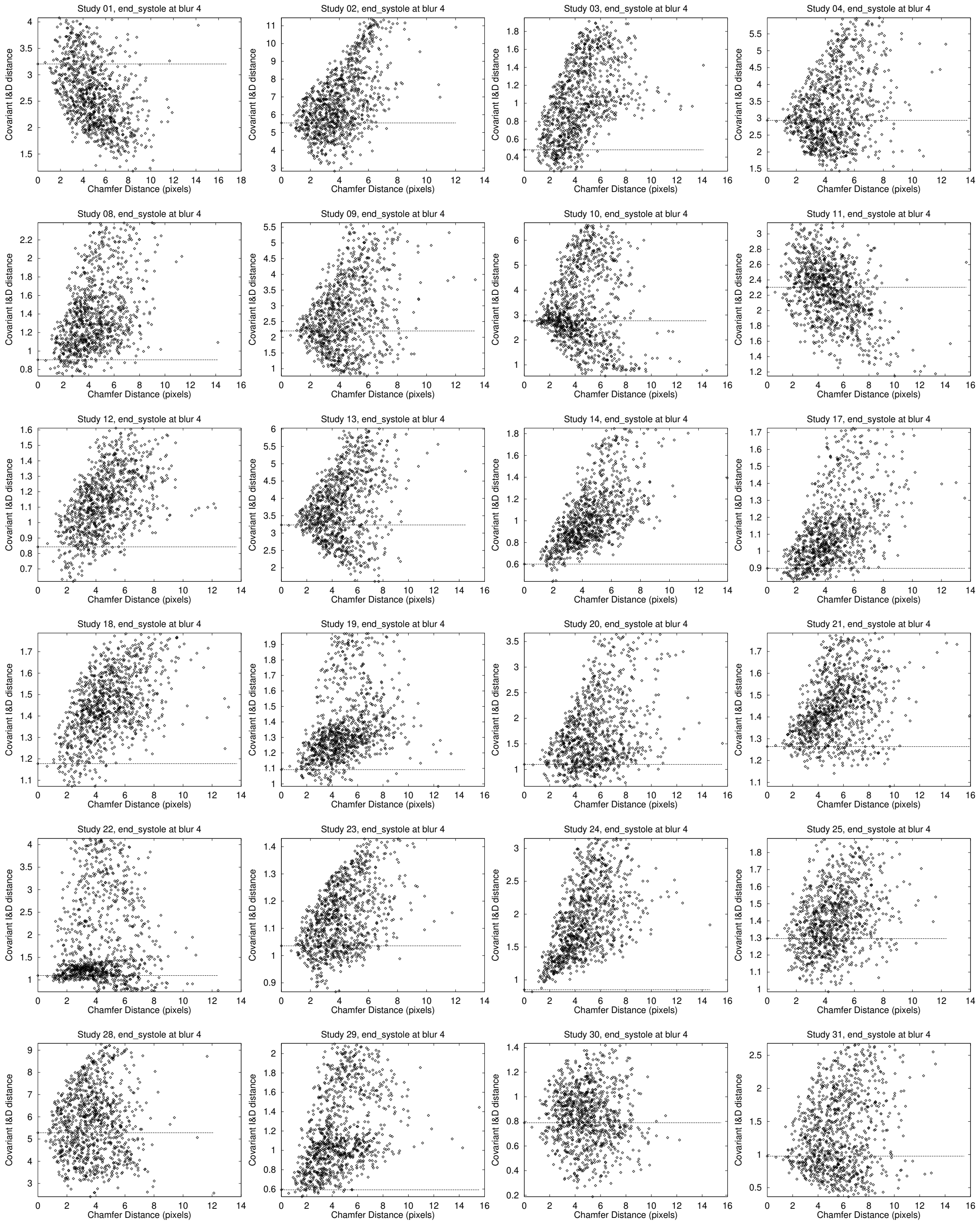
Heart, untrained, scale 4



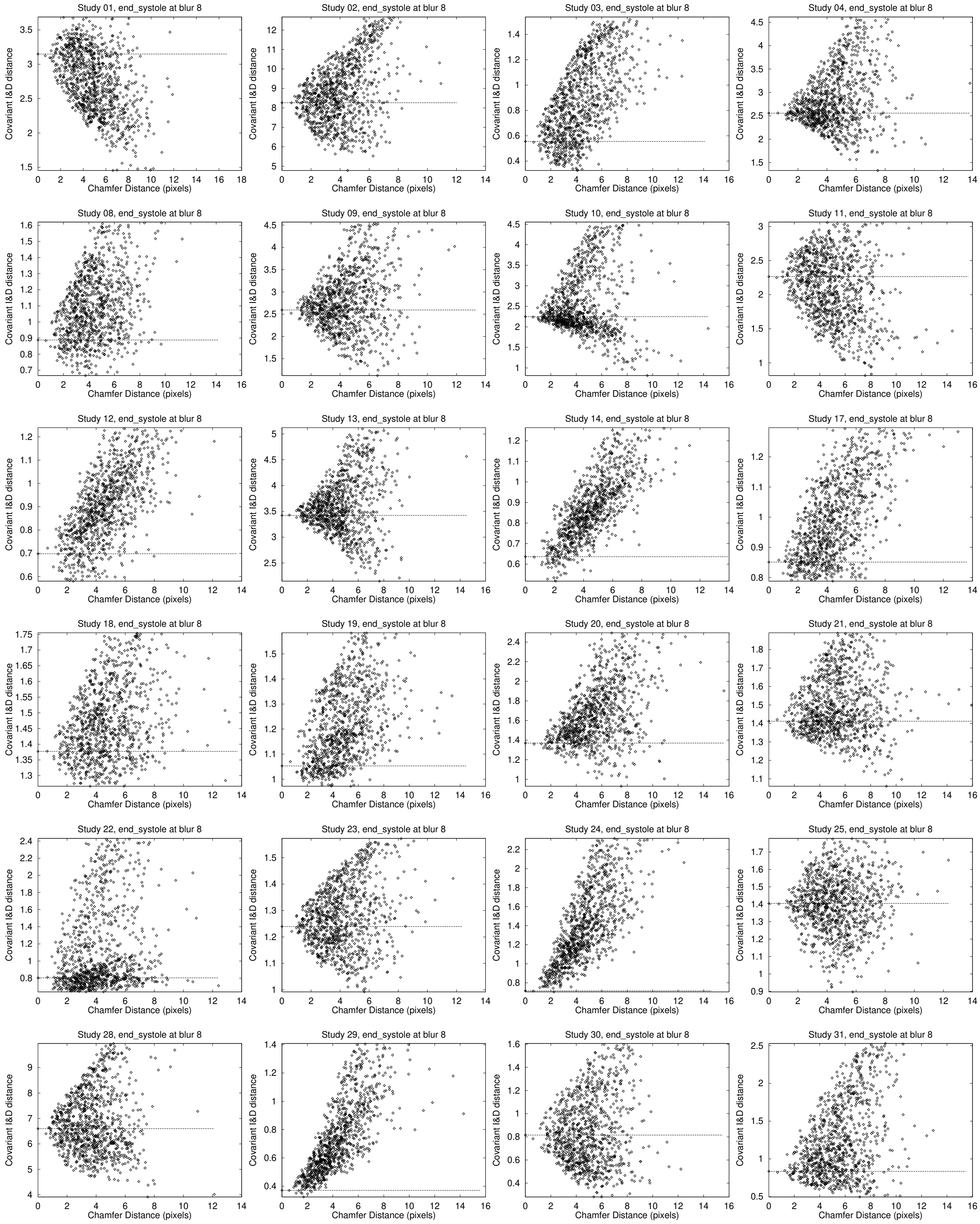
Heart, covariant intensity and gradient Gaussians, scale 2



Heart, covariant intensity and gradient Gaussians, scale 4



Heart, covariant intensity and gradient Gaussians, scale 8



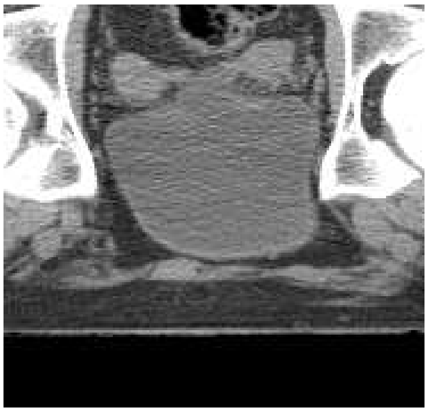
A.4 Images with corresponding plots

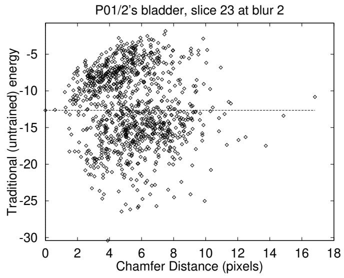
Each bladder image is shown with a plot for the following objective functions:

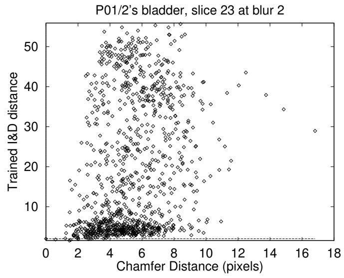
- Upper left: traditional at scale 2,
- Upper right: simple trained at scale 2,
- Lower left: sectored trained at scale 2,
- Lower right: sectored trained at scale 4.

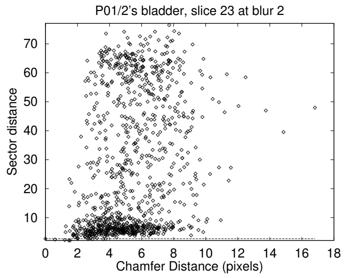
Each heart image is shown with a plot for the following objective functions:

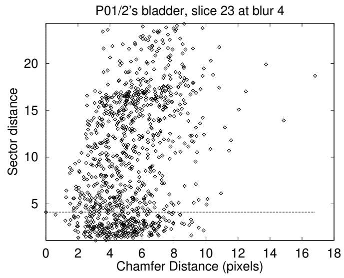
- Upper left: traditional at scale 2,
- Upper right: traditional at scale 4,
- Lower left: covariant trained at scale 2,
- Lower right: covariant trained at scale 4.

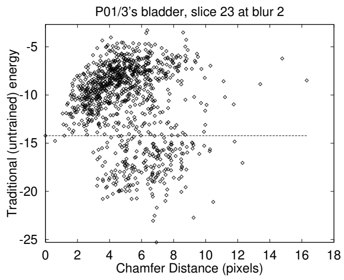


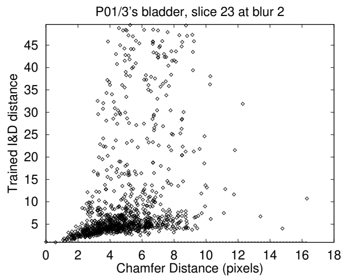

P01/2's bladder, slice 23 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)

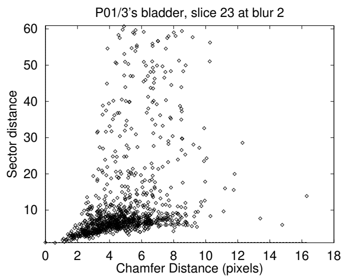

P01/2's bladder, slice 23 at blur 2
Trained I&D distance
Chamfer Distance (pixels)

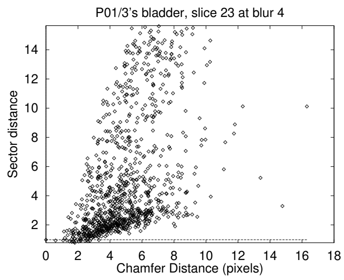

P01/2's bladder, slice 23 at blur 2
Sector distance
Chamfer Distance (pixels)

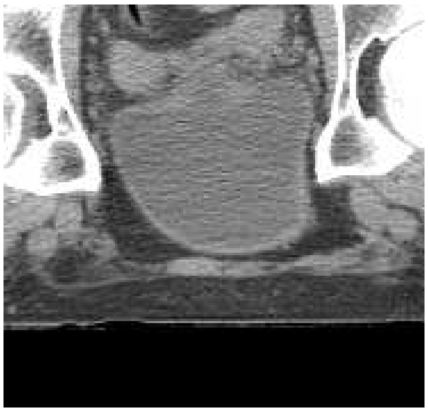

P01/2's bladder, slice 23 at blur 4
Sector distance
Chamfer Distance (pixels)

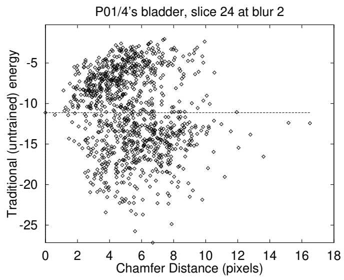

P01/3's bladder, slice 23 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)

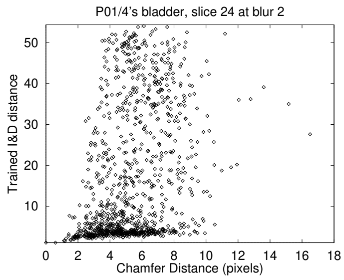

P01/3's bladder, slice 23 at blur 2
Trained I&D distance
Chamfer Distance (pixels)

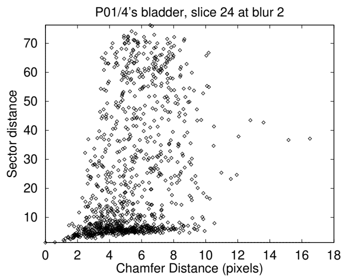

P01/3's bladder, slice 23 at blur 2
Sector distance
Chamfer Distance (pixels)

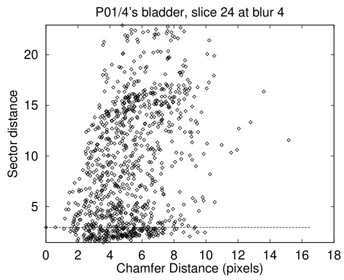

P01/3's bladder, slice 23 at blur 4
Sector distance
Chamfer Distance (pixels)

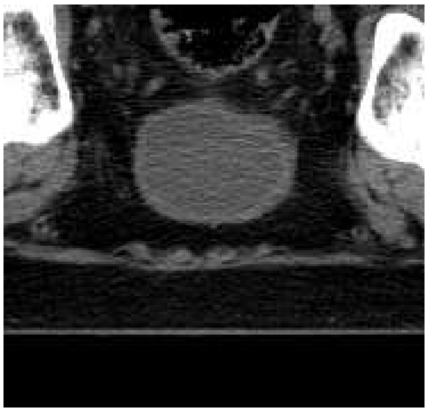


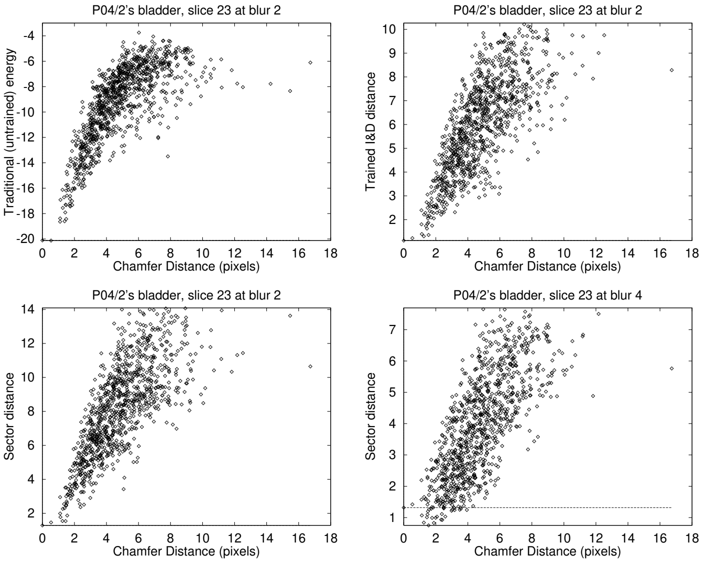

P01/4's bladder, slice 24 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)

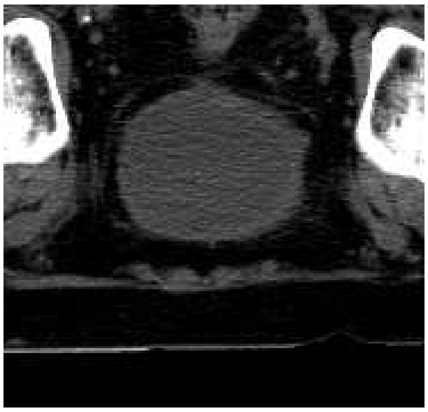

P01/4's bladder, slice 24 at blur 2
Trained I&D distance
Chamfer Distance (pixels)

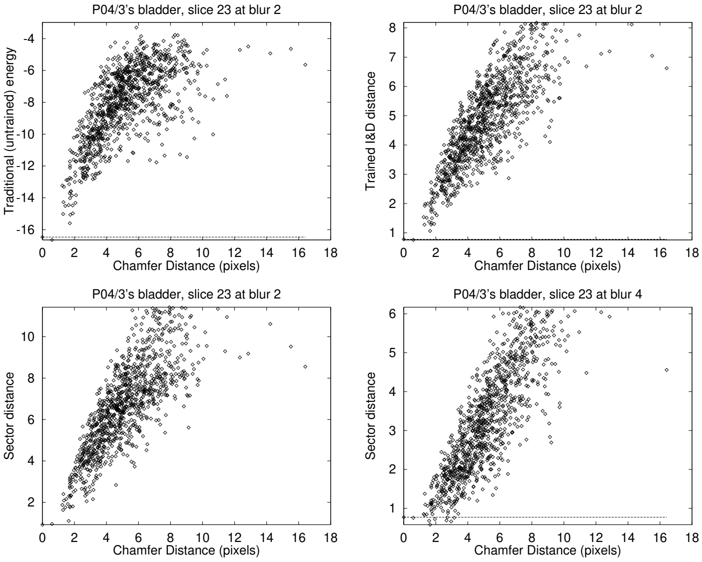

P01/4's bladder, slice 24 at blur 2
Sector distance
Chamfer Distance (pixels)

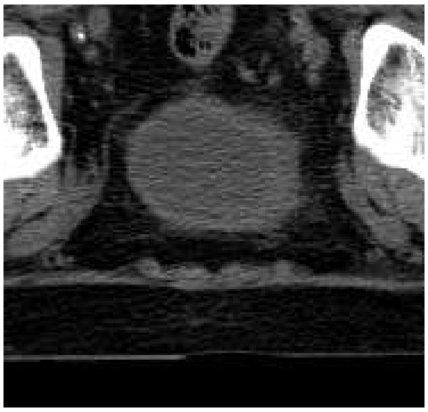

P01/4's bladder, slice 24 at blur 4
Sector distance
Chamfer Distance (pixels)

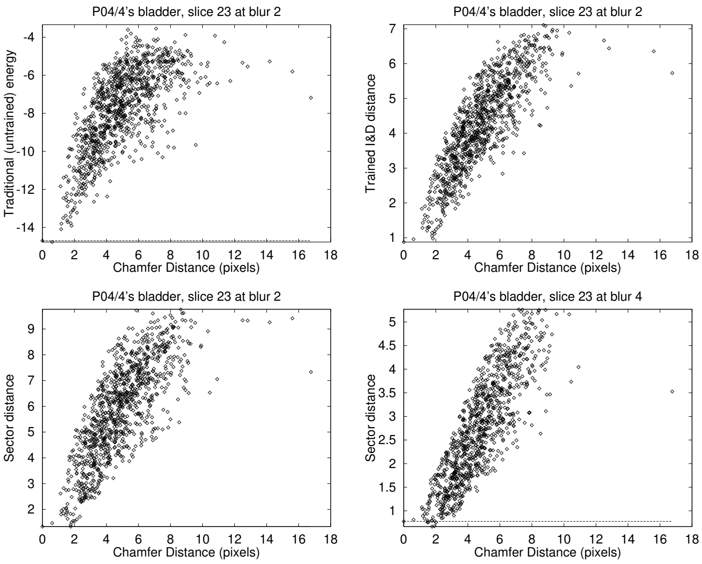


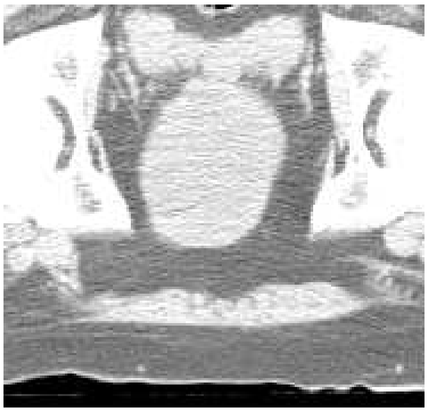

P04/2's bladder, slice 23 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P04/2's bladder, slice 23 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P04/2's bladder, slice 23 at blur 2
Sector distance
Chamfer Distance (pixels)
P04/2's bladder, slice 23 at blur 4
Sector distance
Chamfer Distance (pixels)

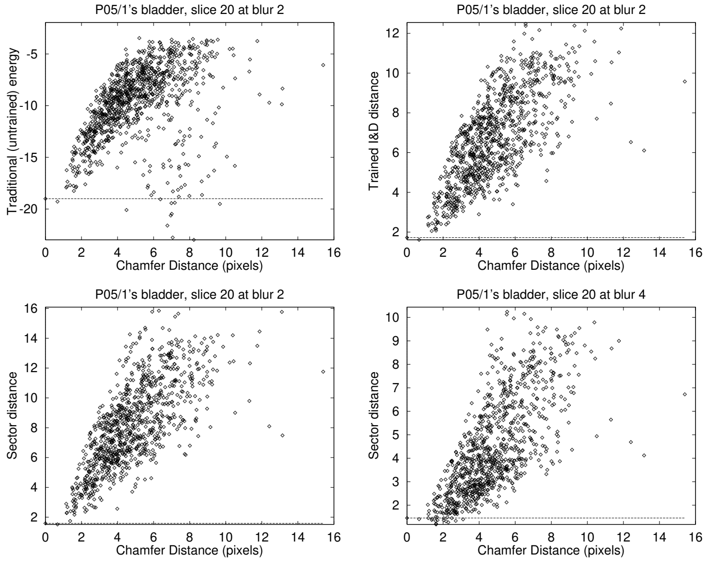



P04/3's bladder, slice 23 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P04/3's bladder, slice 23 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P04/3's bladder, slice 23 at blur 2
Sector distance
Chamfer Distance (pixels)
P04/3's bladder, slice 23 at blur 4
Sector distance
Chamfer Distance (pixels)

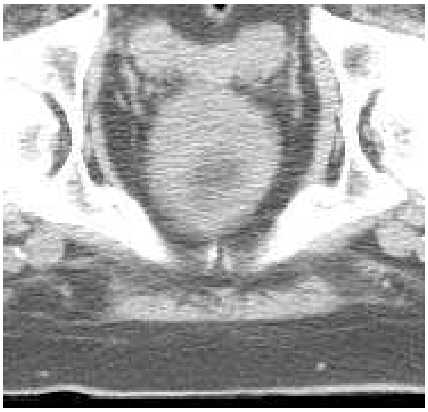


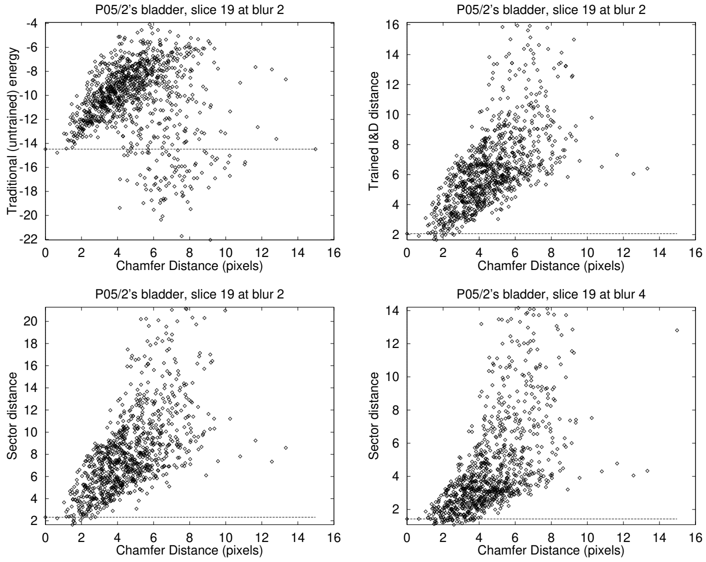

P04/4's bladder, slice 23 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P04/4's bladder, slice 23 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P04/4's bladder, slice 23 at blur 2
Sector distance
Chamfer Distance (pixels)
P04/4's bladder, slice 23 at blur 4
Sector distance
Chamfer Distance (pixels)



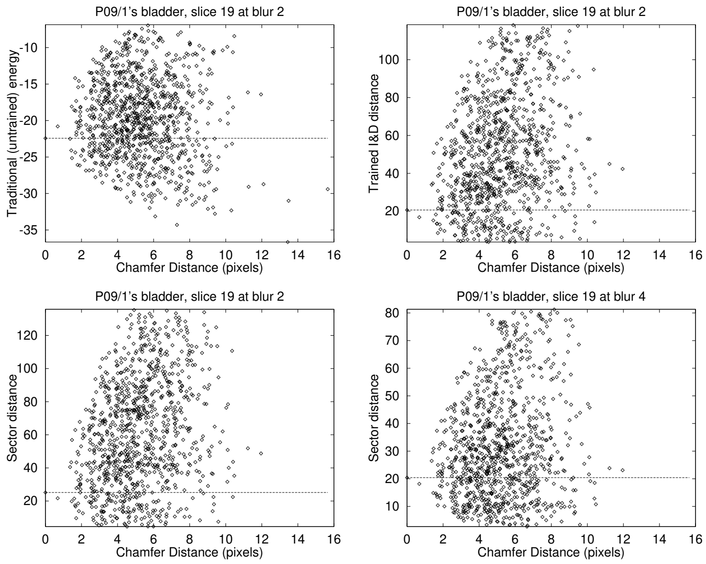


P05/1's bladder, slice 20 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P05/1's bladder, slice 20 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P05/1's bladder, slice 20 at blur 2
Sector distance
Chamfer Distance (pixels)
P05/1's bladder, slice 20 at blur 4
Sector distance
Chamfer Distance (pixels)

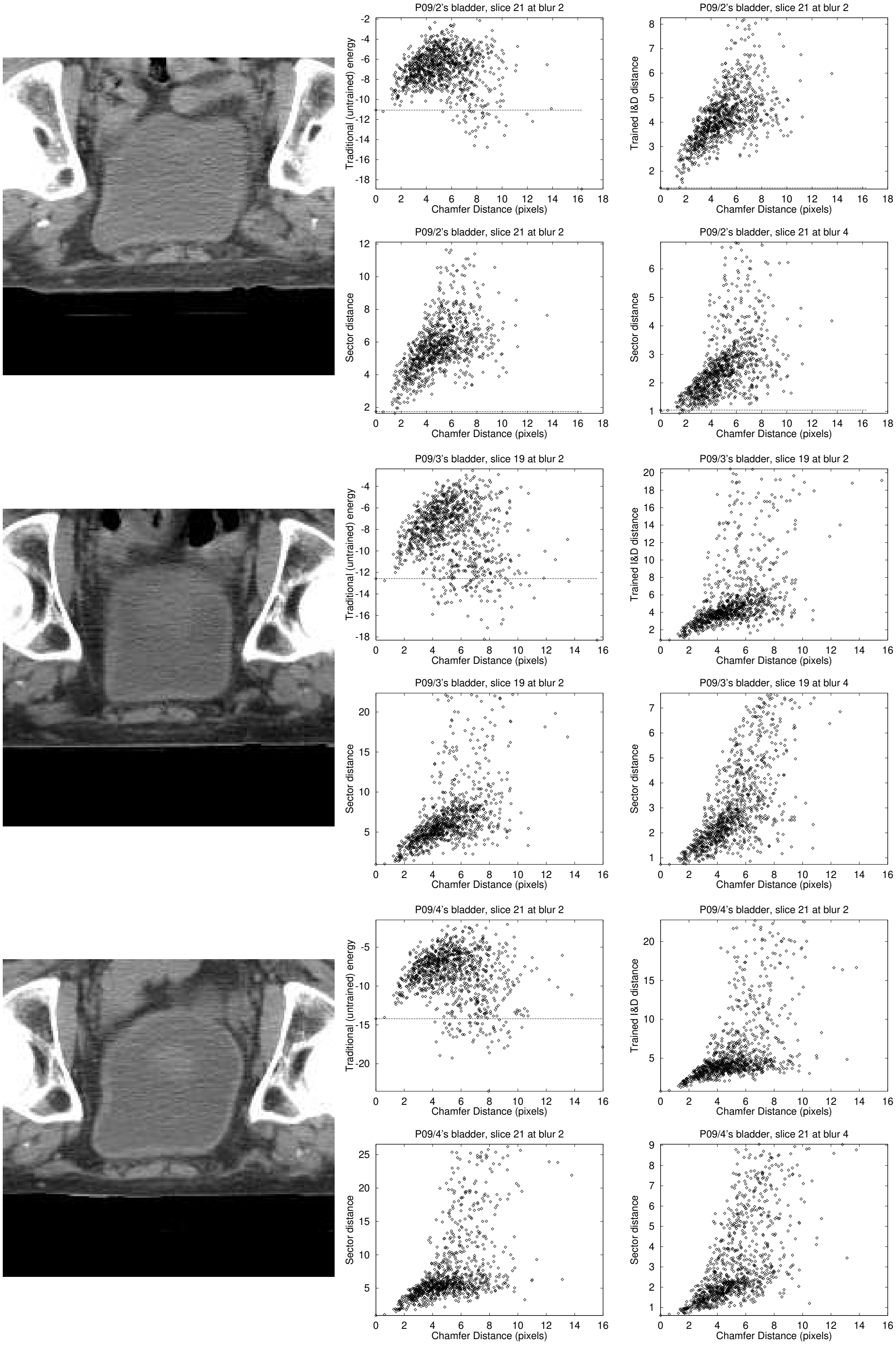


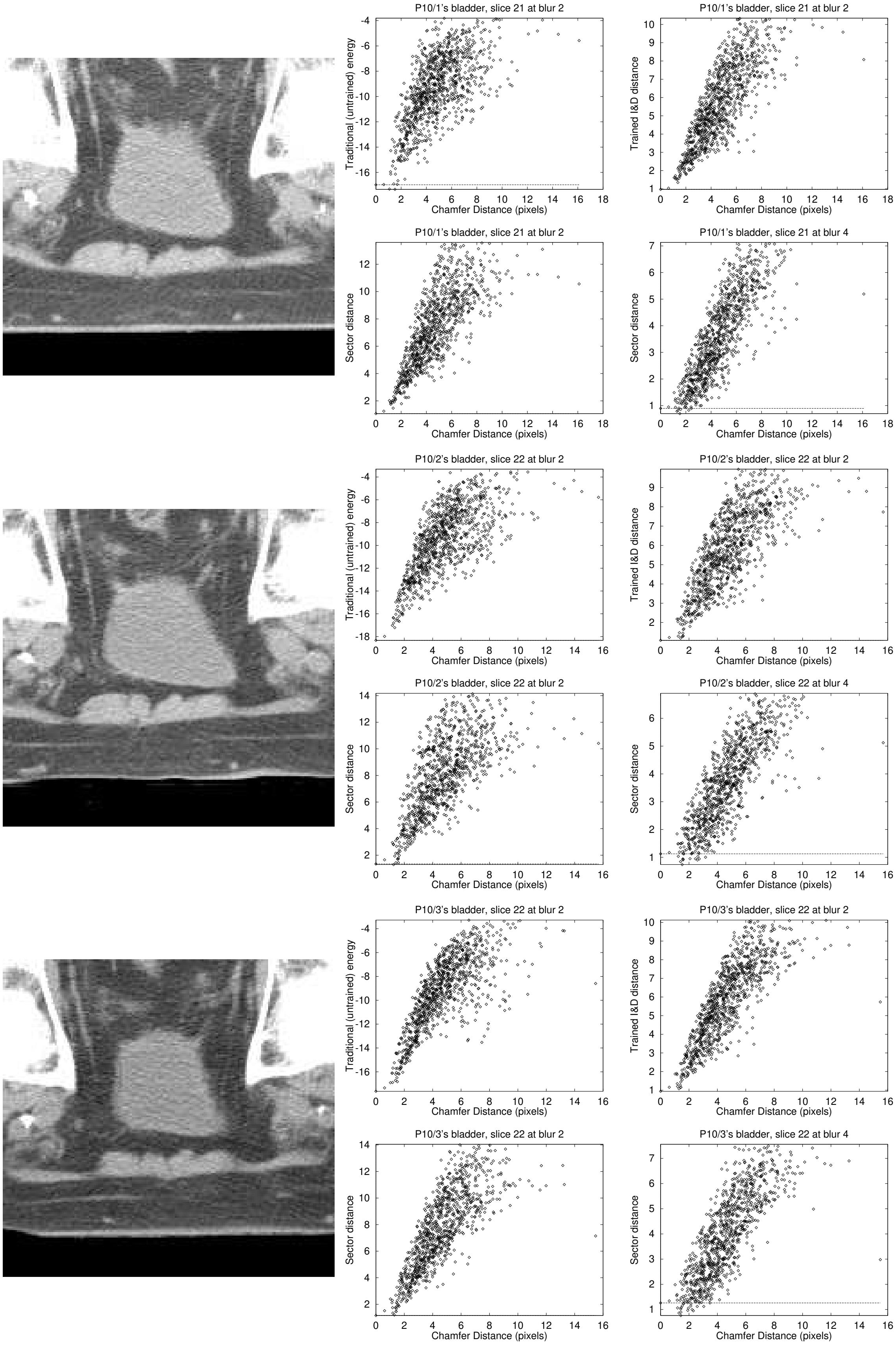


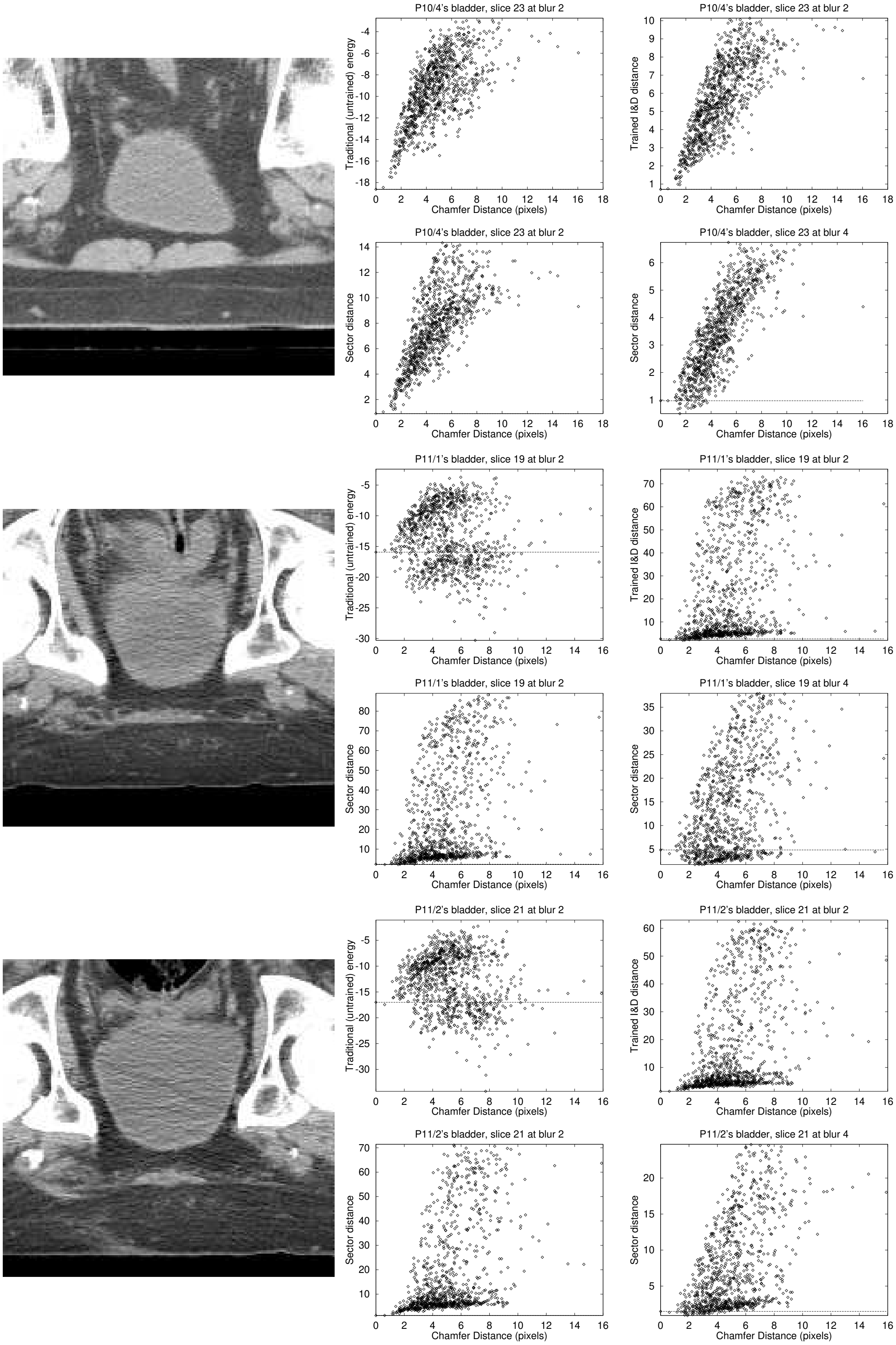
P05/2's bladder, slice 19 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P05/2's bladder, slice 19 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P05/2's bladder, slice 19 at blur 2
Sector distance
Chamfer Distance (pixels)
P05/2's bladder, slice 19 at blur 4
Sector distance
Chamfer Distance (pixels)



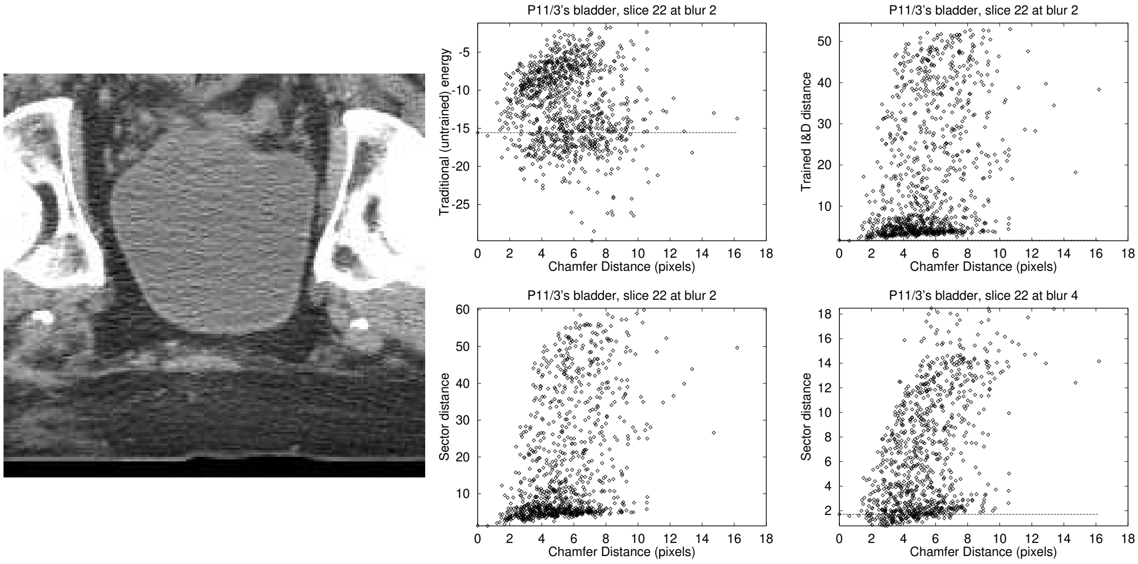
P09/1's bladder, slice 19 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P09/1's bladder, slice 19 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P09/1's bladder, slice 19 at blur 2
Sector distance
Chamfer Distance (pixels)
P09/1's bladder, slice 19 at blur 4
Sector distance
Chamfer Distance (pixels)

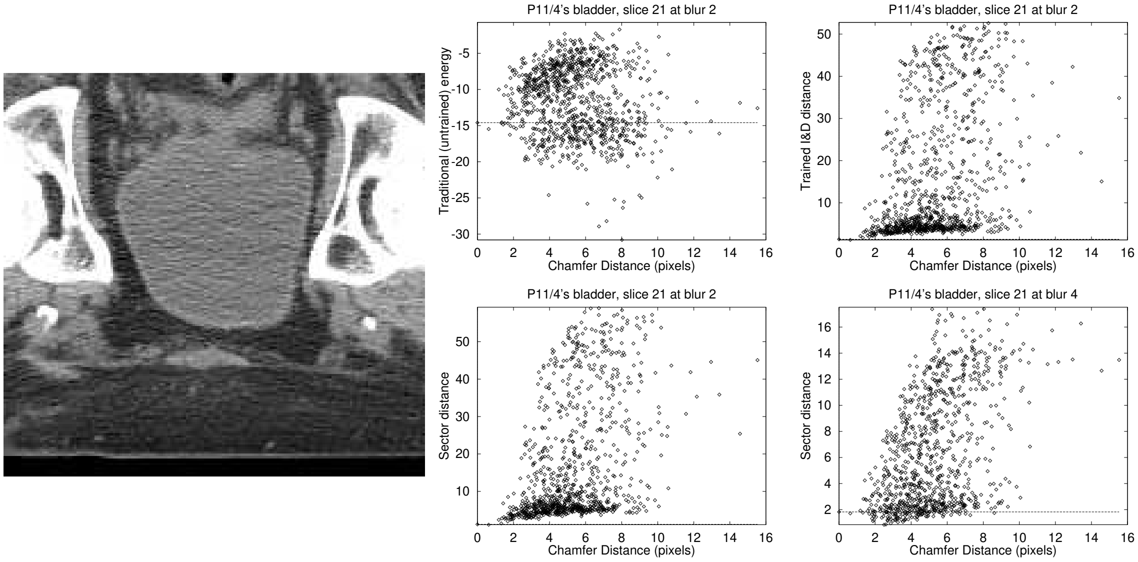

P09/2's bladder, slice 21 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P09/2's bladder, slice 21 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P09/2's bladder, slice 21 at blur 2
Sector distance
Chamfer Distance (pixels)
P09/2's bladder, slice 21 at blur 4
Sector distance
Chamfer Distance (pixels)
P09/3's bladder, slice 19 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P09/3's bladder, slice 19 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P09/3's bladder, slice 19 at blur 2
Sector distance
Chamfer Distance (pixels)
P09/3's bladder, slice 19 at blur 4
Sector distance
Chamfer Distance (pixels)
P09/4's bladder, slice 21 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P09/4's bladder, slice 21 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P09/4's bladder, slice 21 at blur 2
Sector distance
Chamfer Distance (pixels)
P09/4's bladder, slice 21 at blur 4
Sector distance
Chamfer Distance (pixels)

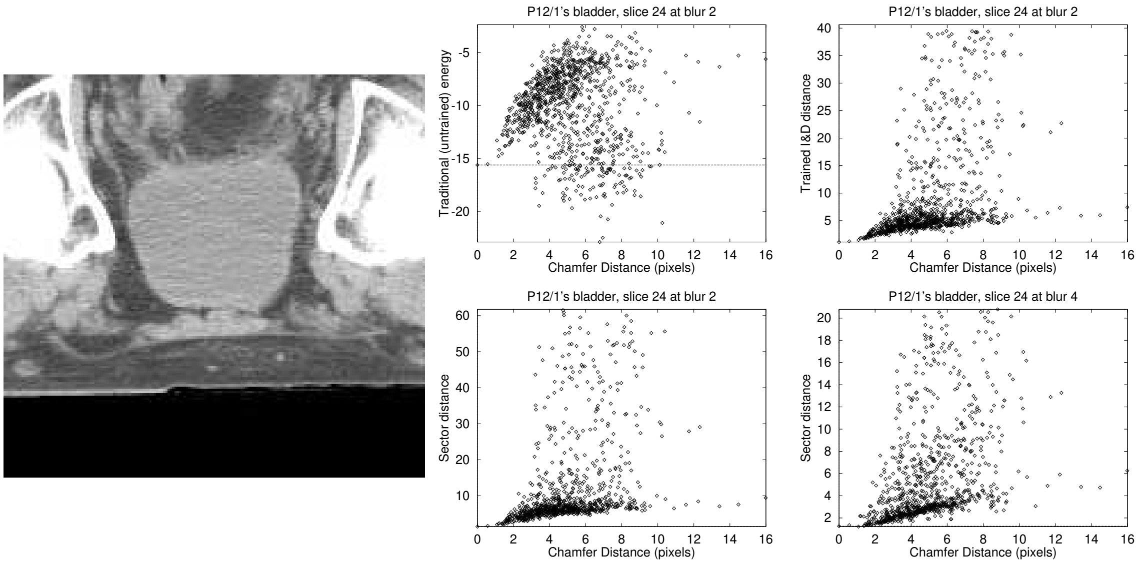

P10/1's bladder, slice 21 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P10/1's bladder, slice 21 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P10/1's bladder, slice 21 at blur 2
Sector distance
Chamfer Distance (pixels)
P10/1's bladder, slice 21 at blur 4
Sector distance
Chamfer Distance (pixels)
P10/2's bladder, slice 22 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P10/2's bladder, slice 22 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P10/2's bladder, slice 22 at blur 2
Sector distance
Chamfer Distance (pixels)
P10/2's bladder, slice 22 at blur 4
Sector distance
Chamfer Distance (pixels)
P10/3's bladder, slice 22 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P10/3's bladder, slice 22 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P10/3's bladder, slice 22 at blur 2
Sector distance
Chamfer Distance (pixels)
P10/3's bladder, slice 22 at blur 4
Sector distance
Chamfer Distance (pixels)

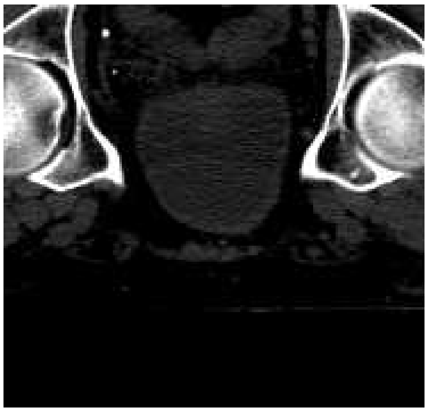


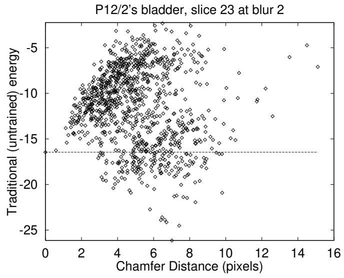
P10/4's bladder, slice 23 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P10/4's bladder, slice 23 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P10/4's bladder, slice 23 at blur 2
Sector distance
Chamfer Distance (pixels)
P10/4's bladder, slice 23 at blur 4
Sector distance
Chamfer Distance (pixels)
P11/1's bladder, slice 19 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P11/1's bladder, slice 19 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P11/1's bladder, slice 19 at blur 2
Sector distance
Chamfer Distance (pixels)
P11/1's bladder, slice 19 at blur 4
Sector distance
Chamfer Distance (pixels)
P11/2's bladder, slice 21 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P11/2's bladder, slice 21 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P11/2's bladder, slice 21 at blur 2
Sector distance
Chamfer Distance (pixels)
P11/2's bladder, slice 21 at blur 4
Sector distance
Chamfer Distance (pixels)

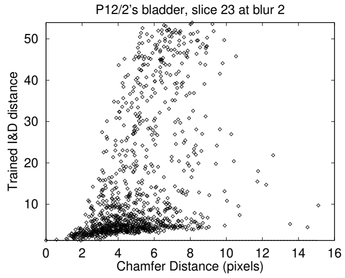

P11/3's bladder, slice 22 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P11/3's bladder, slice 22 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P11/3's bladder, slice 22 at blur 2
Sector distance
Chamfer Distance (pixels)
P11/3's bladder, slice 22 at blur 4
Sector distance
Chamfer Distance (pixels)

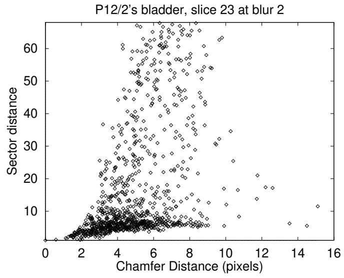

P11/4's bladder, slice 21 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P11/4's bladder, slice 21 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P11/4's bladder, slice 21 at blur 2
Sector distance
Chamfer Distance (pixels)
P11/4's bladder, slice 21 at blur 4
Sector distance
Chamfer Distance (pixels)

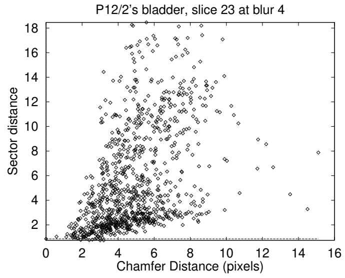

P12/1's bladder, slice 24 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
P12/1's bladder, slice 24 at blur 2
Trained I&D distance
Chamfer Distance (pixels)
P12/1's bladder, slice 24 at blur 2
Sector distance
Chamfer Distance (pixels)
P12/1's bladder, slice 24 at blur 4
Sector distance
Chamfer Distance (pixels)

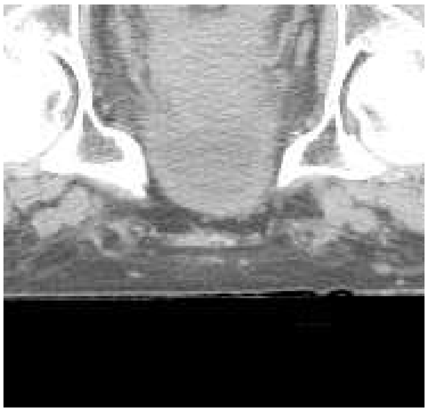


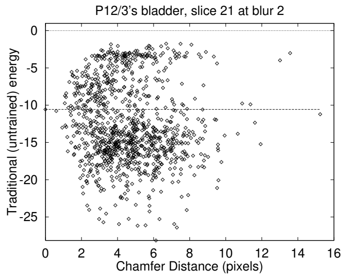

P12/2's bladder, slice 23 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)

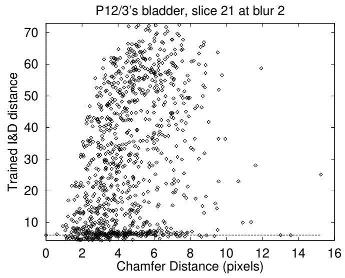

P12/2's bladder, slice 23 at blur 2
Trained I&D distance
Chamfer Distance (pixels)

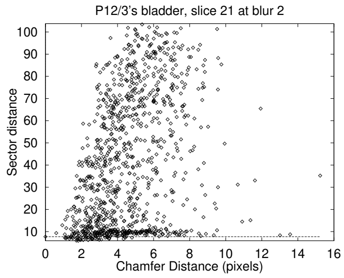

P12/2's bladder, slice 23 at blur 2
Sector distance
Chamfer Distance (pixels)

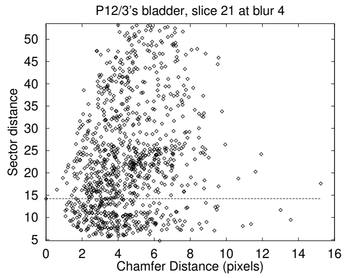

P12/2's bladder, slice 23 at blur 4
Sector distance
Chamfer Distance (pixels)

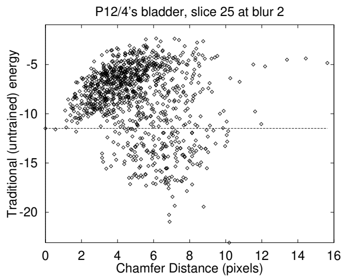


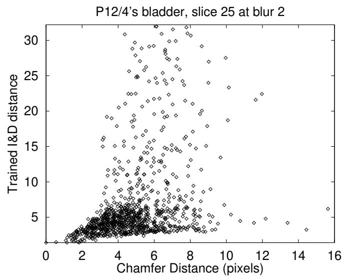

P12/3's bladder, slice 21 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)

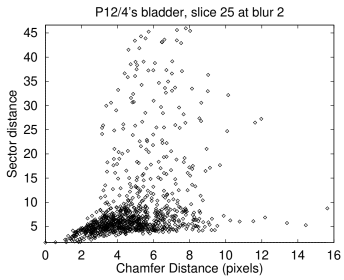

P12/3's bladder, slice 21 at blur 2
Trained I&D distance
Chamfer Distance (pixels)

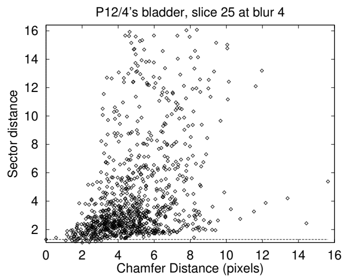

P12/3's bladder, slice 21 at blur 2
Sector distance
Chamfer Distance (pixels)

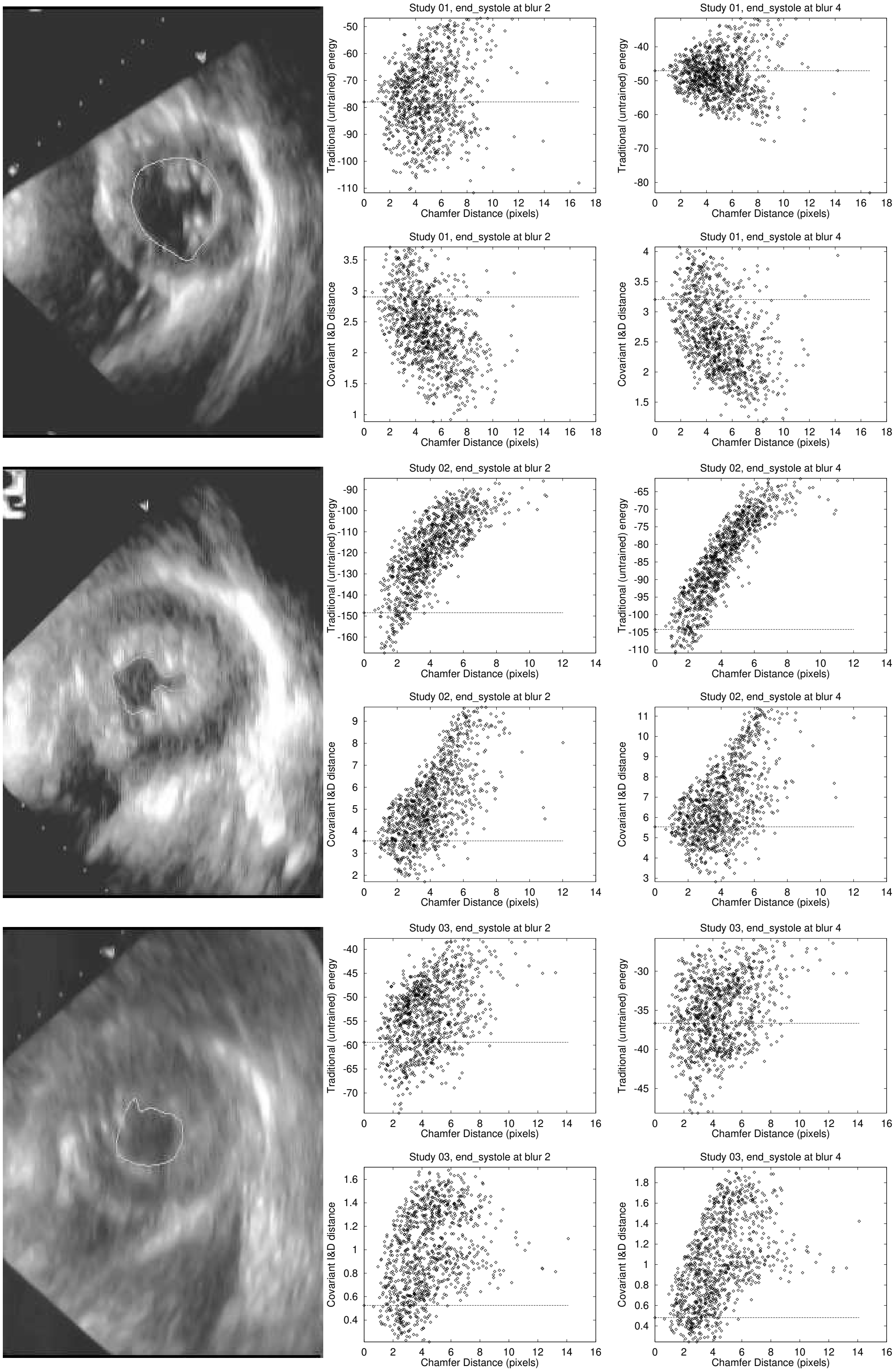

P12/3's bladder, slice 21 at blur 4
Sector distance
Chamfer Distance (pixels)

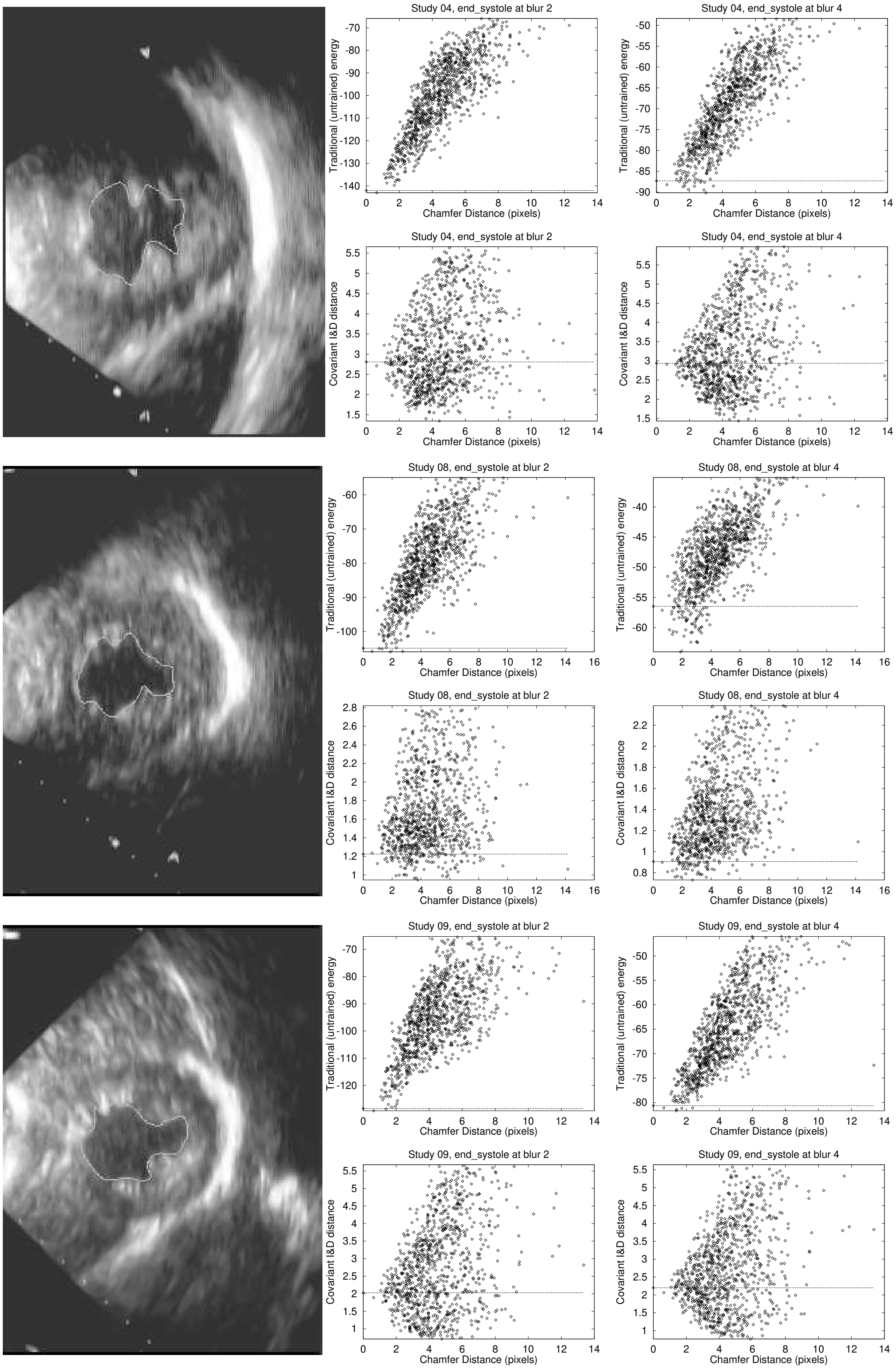

P12/4's bladder, slice 25 at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)

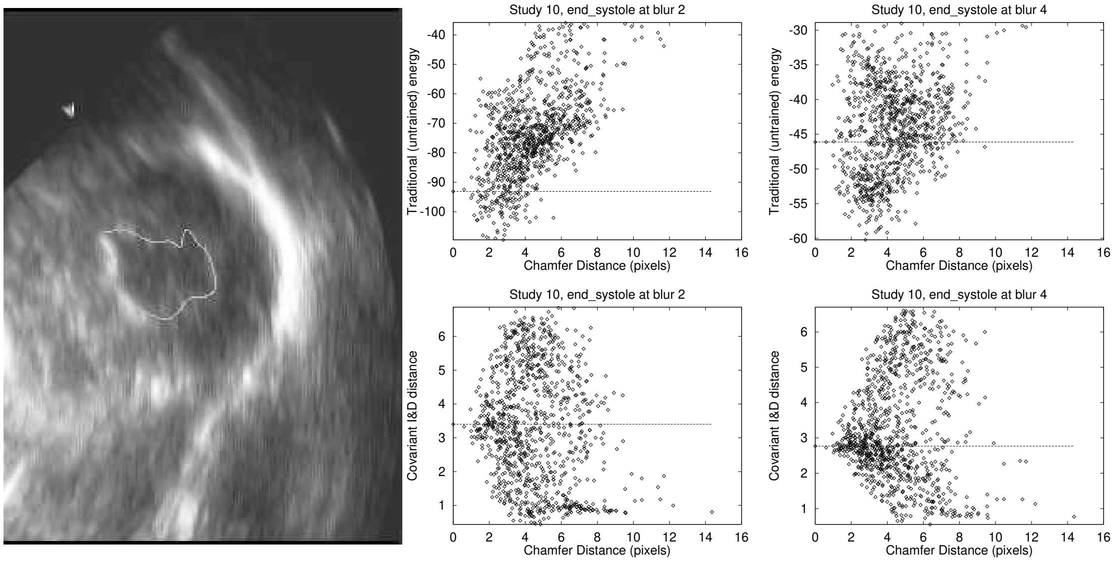

P12/4's bladder, slice 25 at blur 2
Trained I&D distance
Chamfer Distance (pixels)

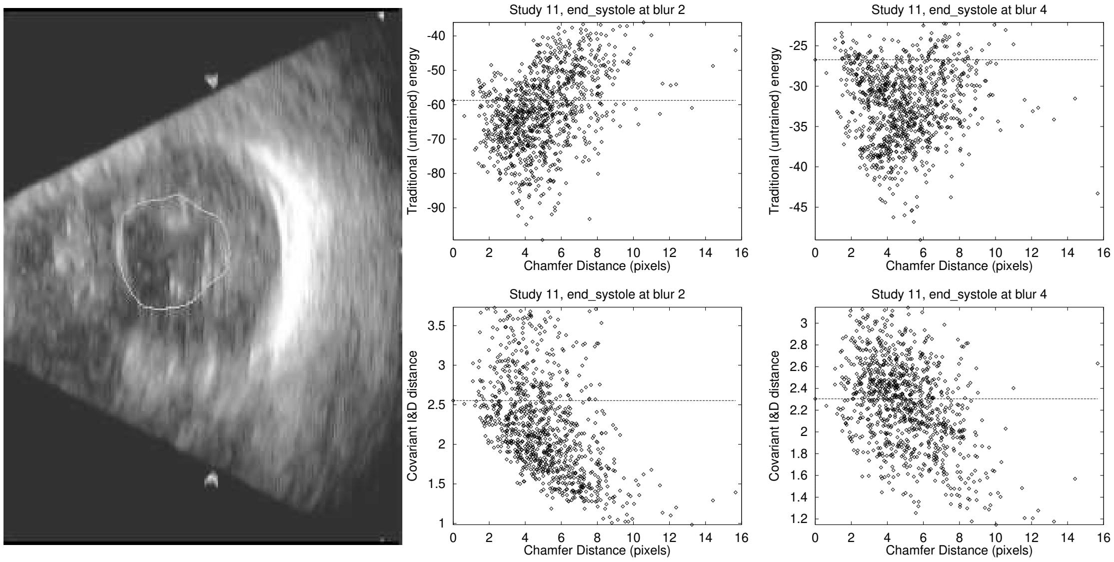

P12/4's bladder, slice 25 at blur 2
Sector distance
Chamfer Distance (pixels)

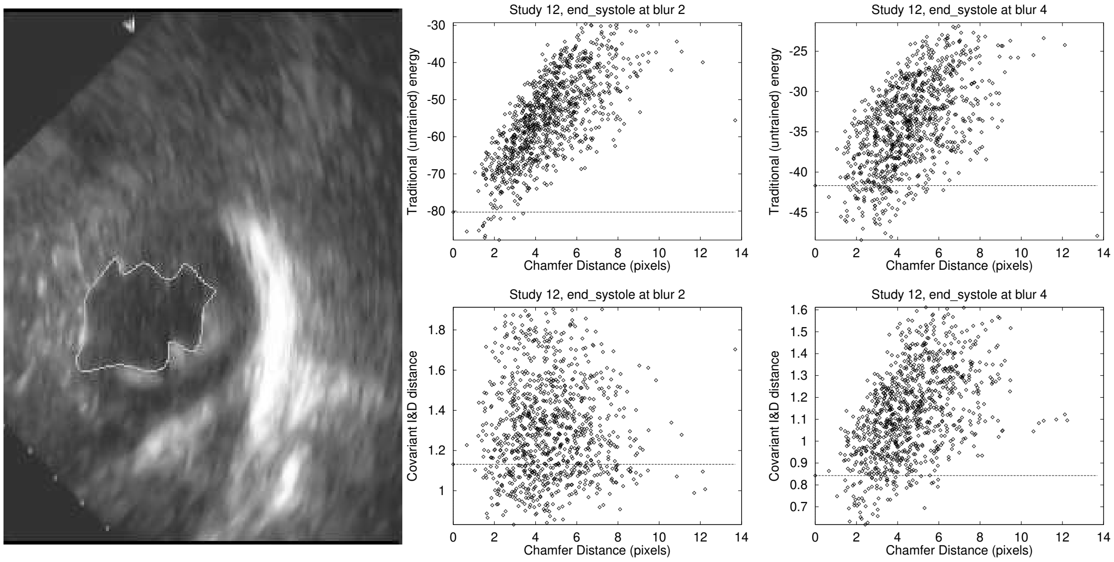

P12/4's bladder, slice 25 at blur 4
Sector distance
Chamfer Distance (pixels)

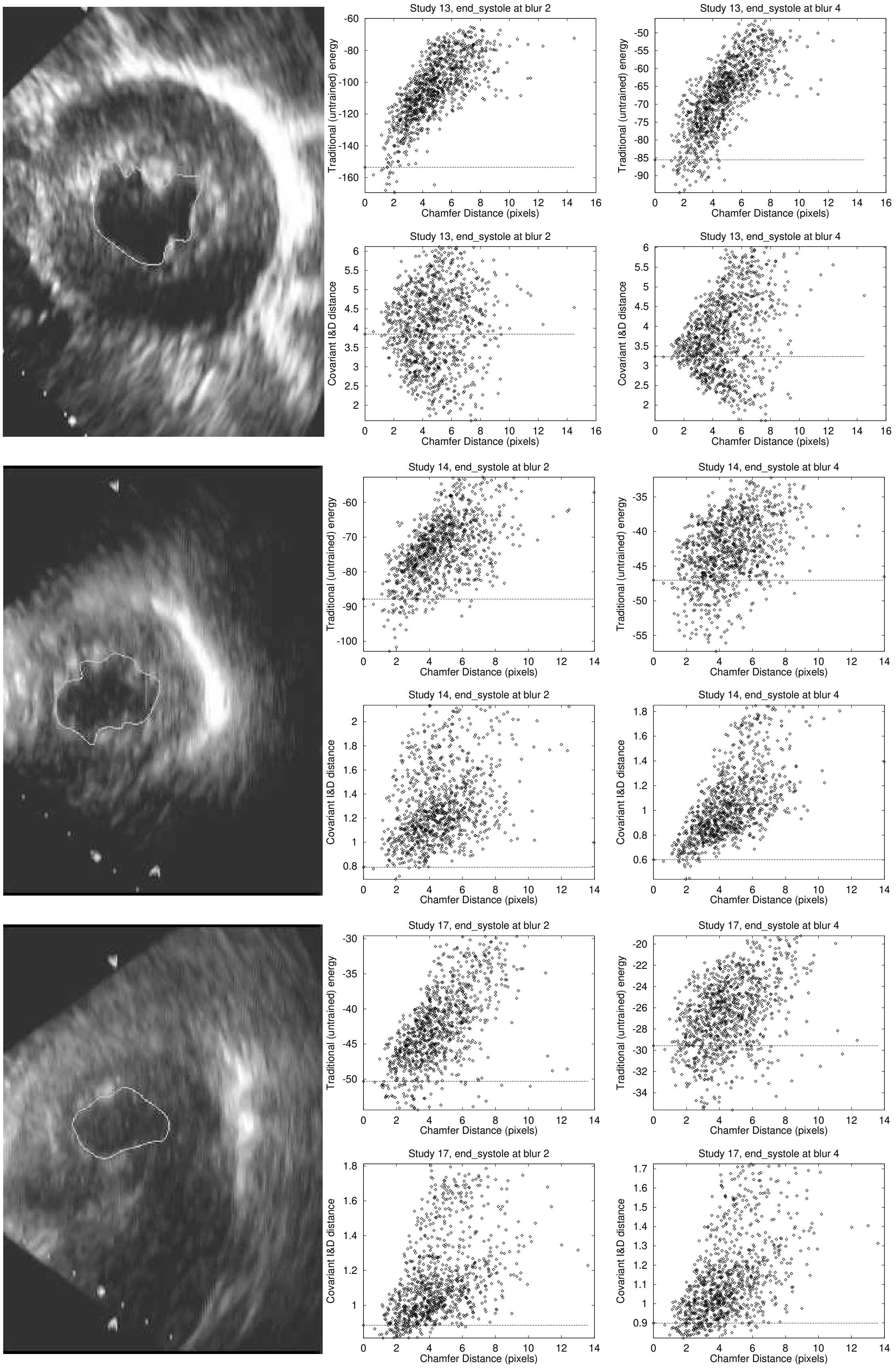

Study 01, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 01, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 01, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 01, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 02, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 02, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 02, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 02, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 03, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 03, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 03, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 03, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)

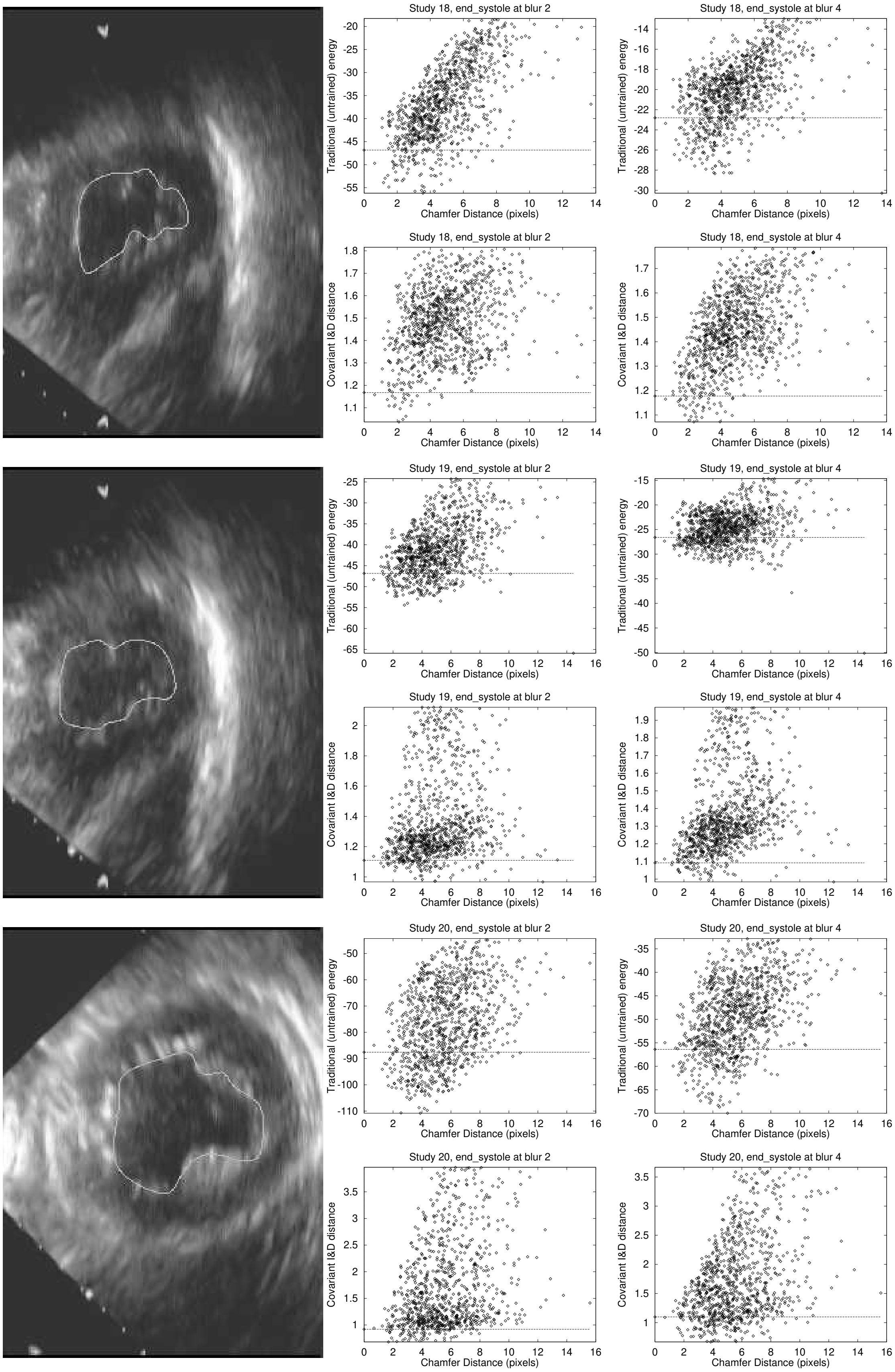

Study 04, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 04, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 04, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 04, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 08, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 08, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 08, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 08, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 09, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 09, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 09, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 09, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)


Study 10, end_systole at blur 2
Study 10, end_systole at blur 4
Traditional (untrained) energy
Covariant I&D distance
Chamfer Distance (pixels)

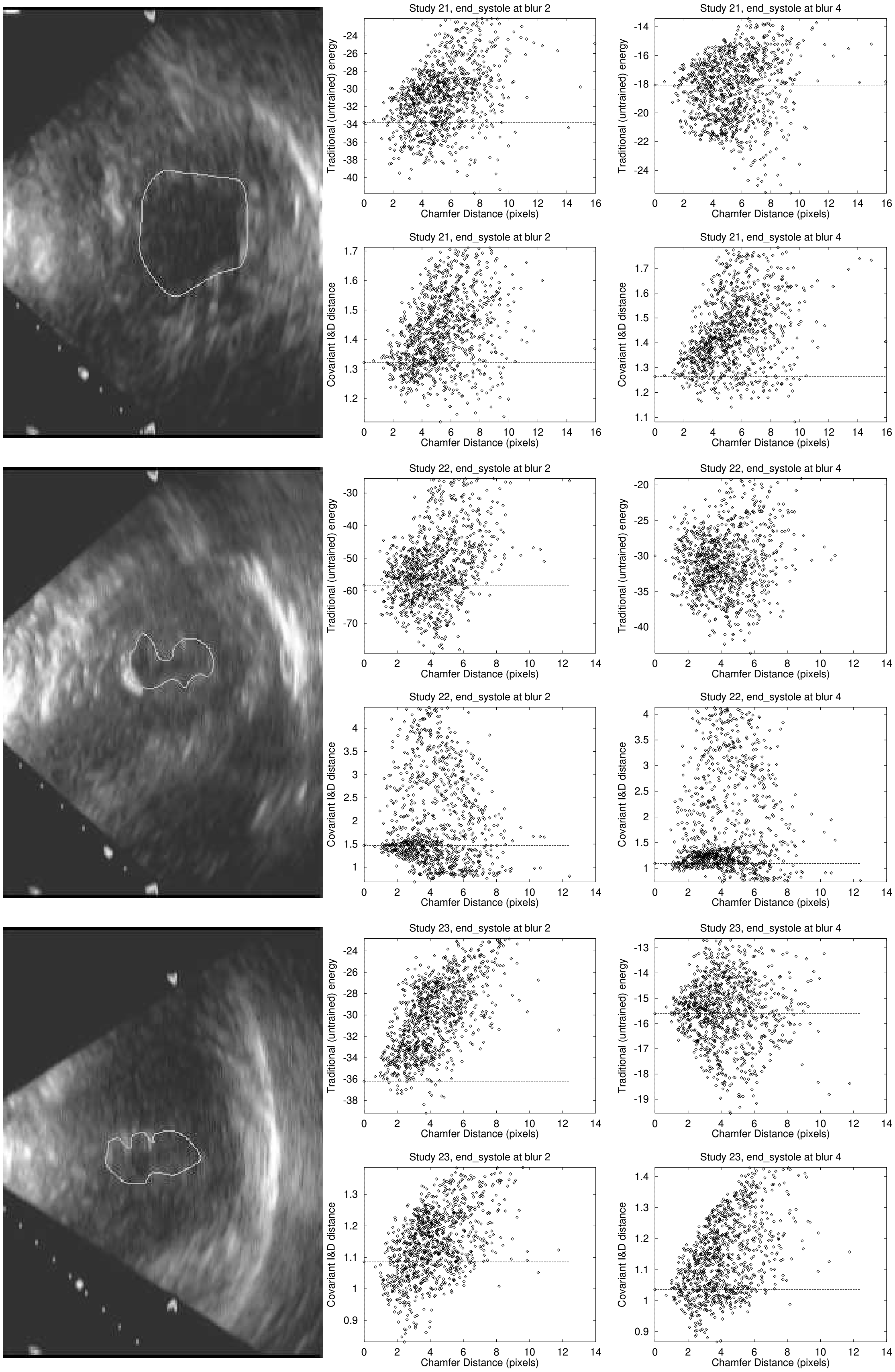

Study 11, end_systole at blur 2
Study 11, end_systole at blur 4
Traditional (untrained) energy
Covariant I&D distance
Chamfer Distance (pixels)

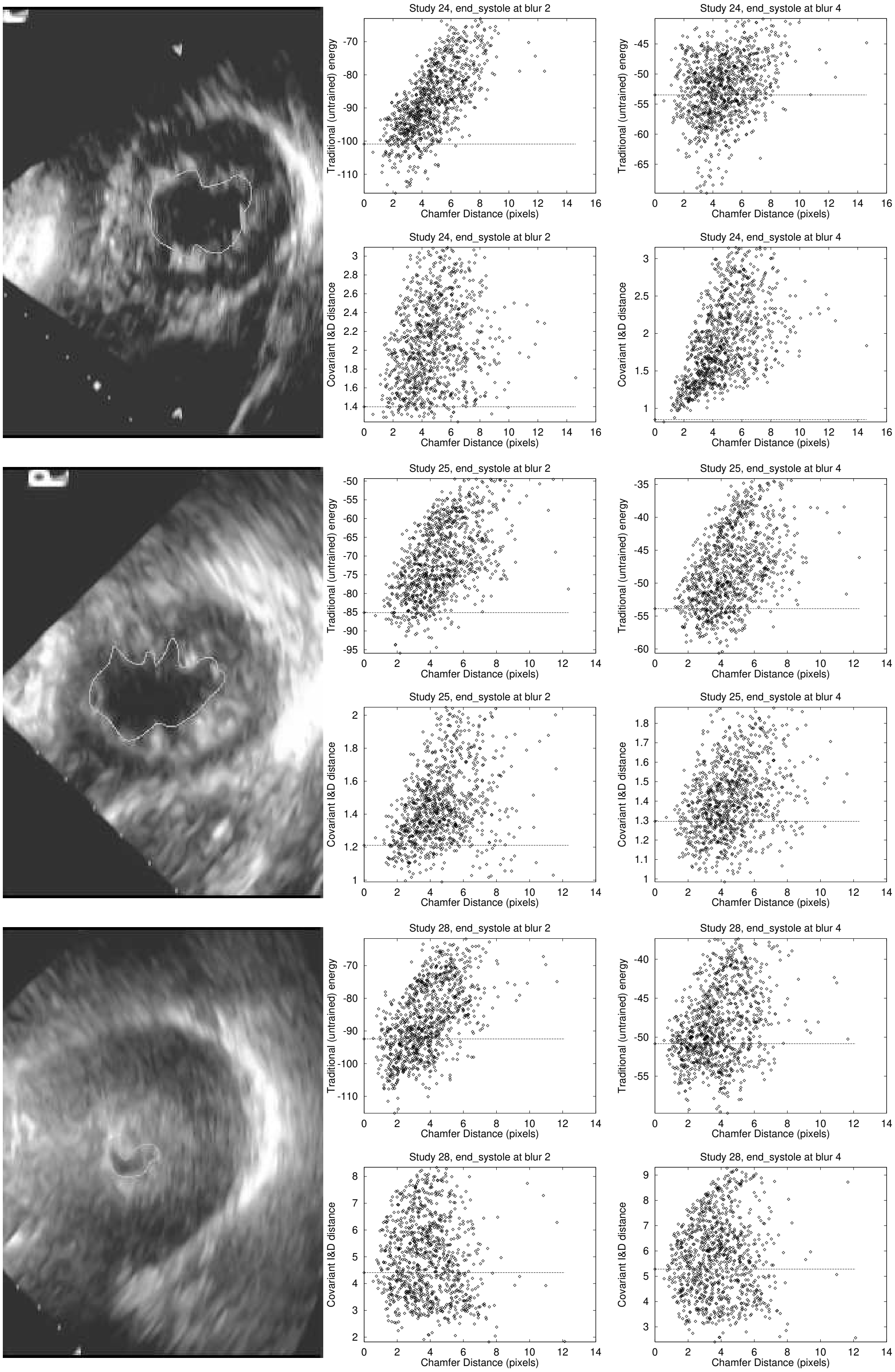

Study 12, end_systole at blur 2
Study 12, end_systole at blur 4
Traditional (untrained) energy
Covariant I&D distance
Chamfer Distance (pixels)

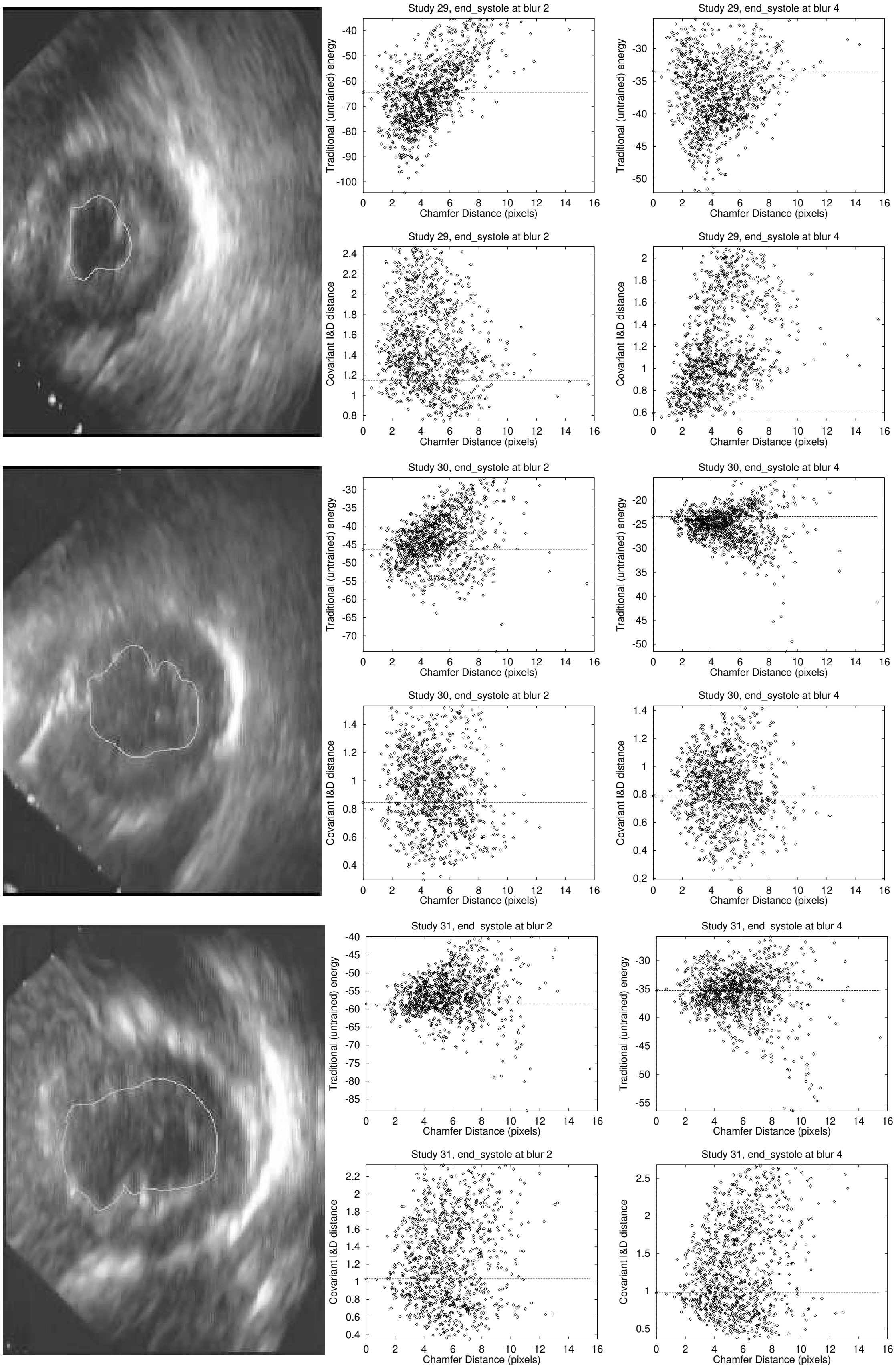

Study 13, end_systole at blur 2
Study 13, end_systole at blur 4
Traditional (untrained) energy
Covariant I&D distance
Chamfer Distance (pixels)
Study 14, end_systole at blur 2
Study 14, end_systole at blur 4
Study 17, end_systole at blur 2
Study 17, end_systole at blur 4



Study 18, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 18, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 18, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 18, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 19, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 19, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 19, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 19, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 20, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 20, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 20, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 20, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)


Study 21, end_systole at blur 2
Study 21, end_systole at blur 4
Study 22, end_systole at blur 2
Study 22, end_systole at blur 4
Study 23, end_systole at blur 2
Study 23, end_systole at blur 4
Traditional (untrained) energy
Covariant I&D distance
Chamfer Distance (pixels)


Study 24, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 24, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 24, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 24, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 25, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 25, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 25, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 25, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 28, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 28, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 28, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 28, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)


Study 29, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 29, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 29, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 29, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 30, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 30, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 30, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 30, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)
Study 31, end_systole at blur 2
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 31, end_systole at blur 4
Traditional (untrained) energy
Chamfer Distance (pixels)
Study 31, end_systole at blur 2
Covariant I&D distance
Chamfer Distance (pixels)
Study 31, end_systole at blur 4
Covariant I&D distance
Chamfer Distance (pixels)

Appendix B

Source Code

Here is the source code to the methods described in this thesis. The listings below, while still quite long, are excerpts. Elisions are indicated by ellipses ("`...`"). Routines irrelevant to the thesis, such as shape models we did not use, data storage and retrieval, memory management and caching, and user interface interaction, are omitted. It is hoped that the operation of the omitted utility data types and functions will be clear from context.

B.1 Code defining snakes

Source file `curve.h`

```
// Define cubic Bezier curve
// Sam Fenster, July 1994

#ifndef INCLUDE_CURVE_H
#define INCLUDE_CURVE_H

#include "pow.h"
#include "combinatorial.h"
#include "2D.h"

inline double Bern (unsigned short n, unsigned short i, double u)
{return combinations (n,i) * pow(u,i) * pow (1-u, (unsigned short) (n-i));}


class Cubic_Curve
{
 public:
    Vector2D p [4];

    Vector2D operator () (double u) const;
    Vector2D d_du (double u) const;

    void print () const;
```

```
    void make_flat (const Vector2D &a, const Vector2D &b);
};

#endif
```

Source file `curve.C`

```
#include "curve.h"
#include "iomanip.h"


Vector2D Cubic_Curve::operator () (double u) const
{
    int i;
    Vector2D s (0,0);
    for (i=0; i<=3; i++)
        s += p [i] * Bern (3, i, u);
    return s;
}


Vector2D Cubic_Curve::d_du (double u) const
{
    double v=1-u;
    Vector2D p10=v*p[0]+u*p[1], p11=v*p[1]+u*p[2], p12=v*p[2]+u*p[3];
    return 3*((v*p11+u*p12) - (v*p10+u*p11));
}


...


void Cubic_Curve::make_flat (const Vector2D &a, const Vector2D &b)
{
    p[0] = a;
    p[3] = b;

    p[1] = (2*p[0] + p[3]) / 3;
    p[2] = (2*p[3] + p[0]) / 3;
}
```

Source file `spline.h`

```
// Routines to solve for continuity in a cubic Bezier spline.
// Sam Fenster, 8/1/94

#ifndef INCLUDED_SPLINE_H
#define INCLUDED_SPLINE_H

#include "matrix.h"
#include "2D.h"
#include "curve.h"

class Cubic_Spline
{
    Cubic_Curve *elems;
    int n_elems, max_elems;

    void operator = (const Cubic_Spline &);   // Inaccessible
    Cubic_Spline (const Cubic_Spline &);      // Inaccessible

public:
```

```
    Cubic_Spline (): n_elems(0), elems(0), max_elems(0) {}
    ~Cubic_Spline () {delete elems;}
    void alloc (int n)
        {delete elems; elems = new Cubic_Curve [max_elems=n]; n_elems=0;}
    int n () const {return n_elems;}
    Cubic_Curve &operator [] (int i) {return elems[i];}
    Cubic_Curve const &operator [] (int i) const {return elems[i];}
    void add (const Vector2D &v) {elems [n_elems++].p[0] = v;}
    void shorten () {n_elems --;}
};

void make_spline_matrix (int n, Square_Matrix &im);
Square_Matrix &spline_matrix (int n);
void c2_spline (Cubic_Spline &);
double dpij_dpk0 (const Cubic_Spline &, int i, int j, int k);

#endif
```

Source file `snake.h`

```
#ifndef SNAKE_H
#define SNAKE_H

#include <bool.h>
#include <array_owner.h>
#include <list.h>
#include <2D.h>
#include <spline.h>
#include <draw.h>
#include <med-image.h>
#include "force.h"

class Snake {
protected:
    int debug_;
public:
    Snake (): debug_(0) {}
    Snake (Snake const &s): debug_ (s.debug_) {}
    virtual ~Snake () {}
    virtual Snake *another_Snake () const = 0;
    Snake *another () const {return (Snake *)(another_Snake());}

    int debug (int d) {int od=debug_; debug_=d; return od;}
    virtual void draw (Draw_Env, int dot_size) const = 0;
    virtual void recompute () = 0;

    virtual bool bending_energy (double &) const = 0;
    virtual bool image_energy (image_t &, double &) const = 0;
    virtual bool apply_force (double bending_weight, double image_weight,
                              double cap, image_t &) = 0;
};


// ----------------------------------------------------------------------------
class Snaxel_Snake: public Snake {
protected:
    bool done;
    List<Point2D> node_list;

public:
    Snaxel_Snake (): done(false) {}
    Snaxel_Snake *another () const {return (Snaxel_Snake *)(another_Snake());}

    virtual void start ();
    virtual void add_node (Point2D);
    virtual void finish () = 0;
    virtual bool is_complete () const {return done;}
```

```
    virtual void draw_partial (Draw_Env, int dot_size) const = 0;
    virtual void rubber_band (Draw_Env, int dot_size, Point2D) const = 0;

    virtual int n () const = 0;
    virtual Point2D points (int) const = 0;

    virtual bool read (char const *file_name, int &slice,
                       Array_Owner<char> &err_msg);
    virtual bool write (char const *file_name, int slice,
                        Array_Owner<char> &err_msg) const;
};


// -----------------------------------------------------------------------------
class Discrete_Snake: public Snaxel_Snake {
    Array_Owner<Point2D> nodes;
    int num_nodes;
    void operator = (const Discrete_Snake &);   // Inaccessible
    Discrete_Snake (const Discrete_Snake &);    // Inaccessible

public:
    Discrete_Snake () {}
    Snake *another_Snake () const {return new Discrete_Snake;}
    Discrete_Snake *another () const
        {return (Discrete_Snake *)(another_Snake());}

    void finish ();
    void recompute () {}

    void draw_partial (Draw_Env, int dot_size) const;
    void rubber_band (Draw_Env, int dot_size, Point2D) const;
    void draw (Draw_Env, int dot_size) const;

    int n () const {return num_nodes;}
    Point2D points (int i) const {return nodes[i];}
    virtual Point2D const &operator [] (int i) const {return nodes[i];}
    virtual Point2D &operator [] (int i) {return nodes[i];}

    bool bending_energy (double &) const;
    bool image_energy (image_t &, double &) const;
    bool apply_force (double bending_weight, double image_weight,
                      double cap, image_t &);
};


// -----------------------------------------------------------------------------
// Cubic spline snake with unspecified shape constraints.
// Not an instantiable class --
// missing recompute(), bending_energy(), apply_force():

class Spline_Snake: public Snaxel_Snake {
protected:
    Force *force;
public:
    Cubic_Spline spline;
public:
    Spline_Snake () {force = 0;}
    Spline_Snake *another () const {return (Spline_Snake *)(another_Snake());}

    void finish ();

    void draw_partial (Draw_Env, int dot_size) const;
    void rubber_band (Draw_Env, int dot_size, Point2D p) const;
    void draw (Draw_Env, int dot_size) const;

    int n () const {return spline.n();}
    Point2D points (int i) const {return spline[i].p[0] + Point2D(0,0);}

    Force *set (Force *f) {Force *f1=force;  force=f;  return f1;}
    bool image_energy (image_t &, double &) const;
};
```

```
// ---------------------------------------------------------------------------
class C2_Snake: public Spline_Snake {
public:
    Snake *another_Snake () const {return new C2_Snake;}
    C2_Snake *another () const {return (C2_Snake *)(another_Snake());}

    void recompute ();

    bool bending_energy (double &) const;
    bool apply_force (double bending_weight, double image_weight,
                      double cap, image_t &);
};


// ---------------------------------------------------------------------------
class Line_Snake: public Spline_Snake {
public:
    Snake *another_Snake () const {return new Line_Snake;}
    Line_Snake *another () const {return (Line_Snake *)(another_Snake());}

    void draw (Draw_Env, int dot_size) const;        // Optimized
    void recompute ();

    bool bending_energy (double &) const;
    bool image_energy (image_t &, double &) const;   // Optimized
    bool apply_force (double bending_weight, double image_weight,
                      double cap, image_t &);
};

#endif
```

Source file `spline-snake.C`

```
#include "snake.h"
#include "c2-force.h"

#include <strstream.h>
#include <string-utils.h>
#include <array_owner.h>
#include <array_io.h>

void Spline_Snake::finish ()
{
    Mutator<Point2D> p(node_list);
    int num_nodes = 0;
    for (; p; p++)  num_nodes ++;
    if (num_nodes < 3) return;

    spline.alloc (num_nodes);

    static Point2D o(0,0);
    for (p ++; p; p.remove())
        spline.add (*p - o);

    // Sometimes last point duplicates first:
    if (spline[0].p[0] == spline[spline.n()-1].p[0]) spline.shorten();
    done = true;

    recompute();
}

// ---------------------------------------------------------------------------
void Spline_Snake::draw (Draw_Env env, int dot_size) const
{
    static Point2D o (0,0), p, p1;
    const double d=1/6., e=1/100.;
    for (int i=0; i<spline.n(); i++)
    {
```

```
        p1 = o + spline[i](0);
        if (dot_size < 0) {
            Vector2D norm = rot90 (unit (spline[i].d_du(0)));
            draw_tic (env, p1, norm, -dot_size);
        }
        else if (i==0)
            draw_hollow_dot (env, p1, dot_size);
        else
            draw_dot (env, p1, dot_size);
        for (double u=d; u<1+e; u+=d) {
            p = o + spline[i](u);
            draw_line (env, p1, p);
            p1 = p;
        }
    }
}

void Spline_Snake::draw_partial (Draw_Env e, int /* dot_size */) const
{
    Point2D old_p;
    Iterator<Point2D> p(node_list);
    if (p)
    {
        old_p = *p;
        for (p++; p; p++)
        {
            draw_line (e, old_p, *p);
            old_p = *p;
        }
    }
}

void Spline_Snake::rubber_band (Draw_Env e, int /*dot_size*/, Point2D last) const
{
    draw_line (e, *node_list.last(), last);
}


// ---------------------------------------------------------------------------
bool Spline_Snake::image_energy (image_t &, double &e) const
{
    if (! force) return false;
    return calc_image_energy (spline, force->calc_snake_image_fn(), e);
}
```

Source file `line-snake.C`

```
#include "snake.h"

#include <strstream.h>
#include <string-utils.h>
#include <array_owner.h>
#include <array_io.h>
#include "line-force.h"

void Line_Snake::draw (Draw_Env e, int dot_size) const
{
    static Point2D o (0,0), p;
    for (int i=0; i<n(); i++)
    {
        p = o + spline[i].p[0];
        if (dot_size < 0)
        {
            Vector2D norm = rot90 (unit (spline[i].d_du(0)));
            draw_tic (e, p, norm, -dot_size);
        }
        else if (i==0)
```

```
        {
            Vector2D direction = spline[i].d_du(0);
            draw_arrowhead (e, p, 4*dot_size, direction);
        }
        else draw_dot (e, p, dot_size);
        draw_line (e, p, o+spline[i].p[3]);
    }
}



// ---------------------------------------------------------------------------
void Line_Snake::recompute ()
{
    for (int i=0; i<n(); i++)
    {
        Vector2D a = spline[i].p[0];
        Vector2D b = spline[(i+1)%n()].p[0];
        spline[i].make_flat(a,b);
    }
}


// ---------------------------------------------------------------------------
bool Line_Snake::bending_energy (double &e) const
{
    return line_bending_energy (spline, e);
}


// ---------------------------------------------------------------------------
bool Line_Snake::image_energy (image_t &, double &e) const
{
    if (! force) return false;
    return line_image_energy (spline, force->calc_snake_image_fn(), e);
}


// ---------------------------------------------------------------------------
static void print_force (double w, Cubic_Spline const &spline,
                         Vector2D *force)
{
    int old_precision = cout.precision (3);
    for (int i=0; i<spline.n(); i++)
        cout << spline[i].p[0] << "  +=  "
             << w << "*" << setw(5) << force[i] << endl;
    cout.precision (old_precision);
}



// ---------------------------------------------------------------------------
static Vector2D cap (Vector2D v, double max)
{
    double n = norm(v);
    if (n>max) v *= max/n;
    return v;
}

// ---------------------------------------------------------------------------
bool Line_Snake::apply_force (double bending_weight, double image_weight,
                              double max, image_t &image)
{
    if (! force) return false;
    Array_Owner<Vector2D> bending_force (new Vector2D [n()]);
    Array_Owner<Vector2D> image_force (new Vector2D [n()]);

    if (! line_bending_force (spline, bending_force))
        return false;
    if (! line_image_force (spline, force->calc_snake_image_derivs(),
                              image_force))
        return false;
```

```
    if (debug_>0) {
        cout << "Bending force:" << endl;
        print_force (bending_weight, spline, bending_force);
        cout << "Image force:" << endl;
        print_force (image_weight, spline, image_force);
    }

    for (int i=0; i<spline.n(); i++)
        spline[i].p[0] += cap (bending_weight * bending_force[i]
                              + image_weight * image_force[i],
                              max);
    recompute();
    if (! force->preprocess_snake (spline))
        cerr << "apply_force(): Couldn't postprocess snake." << endl;
    return true;
}
```

B.2 Code defining forces/energies, features and PDFs

Source file `force.h`

```
#ifndef FORCE_H
#define FORCE_H

#include <bool.h>
#include <iostream.h>
#include <array_owner.h>
#include <spline.h>
#include <med-image.h>

typedef bool (*Snake_Image_Fn) (double &e, const Cubic_Spline &snake,
                                int segment, double u);

typedef bool (*Snake_Image_Derivs_Fn) (Vector2D &dE_dp, Vector2D &dE_dn,
                        const Cubic_Spline &snake, int segment, double u);

class Training {
public:
    virtual ~Training () {}
    virtual char const *type () const = 0;
    virtual bool add (Training const &) {return true;}
    virtual bool read (const char *file_name, double &scale,
                       Array_Owner<char> &err_msg) {return true;}
    virtual bool write (const char *file_name,
                        Array_Owner<char> &err_msg) const {return true;}
    virtual void forget () {}
    virtual bool is_trained () const {return true;}
};

class Force {
public:
    virtual ~Force () {}
    virtual Training &training () = 0;
    virtual Training const &training () const = 0;

    virtual bool preprocess_image (image_t &im, double scale,
                                   char const *file_name, int slice)
        {return true;}
    virtual bool preprocess_snake (const Cubic_Spline &)
        {return true;}

    virtual bool energy (double &e,
                         const Cubic_Spline &, int segment, double u) = 0;
    virtual Snake_Image_Fn calc_snake_image_fn () const = 0;
```

```
    virtual bool force (Vector2D &dE_dp, Vector2D &dE_dn,
                        const Cubic_Spline &, int segment, double u) = 0;
    virtual Snake_Image_Derivs_Fn calc_snake_image_derivs () const = 0;

    virtual bool train (const Cubic_Spline &) {return true;}

    virtual void print_training_data (ostream &, const Cubic_Spline &) {}
};

#endif
```

Source file `line-force.h`

```
#ifndef LINE_FORCE_H
#define LINE_FORCE_H

#include <spline.h>
#include "force.h"

bool line_image_energy (const Cubic_Spline &, Snake_Image_Fn, double &);

bool line_bending_energy (const Cubic_Spline &, double &);

bool line_image_force (const Cubic_Spline &, Snake_Image_Derivs_Fn,
                       Vector2D *force);

bool line_bending_force (const Cubic_Spline &, Vector2D *force);

#endif
```

Source file `line-image-f.C`

```
#include "line-force.h"

static bool status;

static double energy (const Cubic_Spline &snake,
                      Snake_Image_Fn snake_image_fn,
                      int i, int &n)
{
    // Return image energy of snake segment i.
    // Image energy is integrated over segment with respect to arc length.
    // Add to n the length (in pixel units).

    double sum=0;
    double speed = norm (snake[i].d_du(0.5));  // Constant. Get at random u.
    for (double u=0; u<1;)
    {
        double e;
        if (! (*snake_image_fn) (e, snake, i, u))
            {status = false; return 0;}
        sum += e;
        n ++;
        u += 1/speed;  // Move by 1 pixel.
    }
    return sum;
}


static double denergy_dp (const Cubic_Spline &snake,
                          Snake_Image_Derivs_Fn snake_image_derivs_fn,
                          int i, int j, double Vector2D::*x, int &n)
{
```

```
    // Return derivative of image energy of snake line segment i with respect
    // to segment j's p[0].*x.
    // Image energy is integrated over segment with respect to arc length.
    // Add to n the length (in pixel units).

    int dist = (j-i+snake.n()) % snake.n();
    if (dist>1) return 0;

    double sum=0;
    for (double u=0; u<1;)
    {
        Vector2D dp_dpk0 (0,0);
        dp_dpk0.*x = dist? u: 1-u;          // d s[i](u) / d s[j].p[0].*x
        Vector2D dp_du = snake[i].d_du(u); //ds[i](u)/du=s[i+1].p[0]-s[i].p[0]
        double speed = norm (dp_du);
        Vector2D unit_dp_du = dp_du/speed;
        Vector2D dn_dpk0;
        dn_dpk0 = unit_dp_du * (rot90(unit_dp_du).*x) / speed;
        if (dist==1) dn_dpk0 = -dn_dpk0;

        Vector2D dE_dp, dE_dn;
        if (! (*snake_image_derivs_fn) (dE_dp, dE_dn, snake, i, u))
            {status = false; return 0;}
        sum += dot (dE_dp, dp_dpk0) + dot (dE_dn, dn_dpk0);

        n ++;
        u += 1/speed;  // Move by 1 pixel.
    }
    return sum;
}


bool line_image_energy (const Cubic_Spline &snake,
                        Snake_Image_Fn snake_image_fn,
                        double &e)
{
    status = true;
    double sum=0;
    int n=0;
    // Sum the image energy from all nodes.
    for (int i=0; i < snake.n(); i++)
    {
        sum += energy (snake, snake_image_fn, i, n);
        if (! status) return false;
    }
    e = sum/n;
    return true;
}


bool line_image_force (const Cubic_Spline &snake,
                       Snake_Image_Derivs_Fn snake_image_derivs_fn,
                       Vector2D *force)
{
    // Accept an array of snake segments and an image slice.  The snake is a
    // closed polyline.  Return an array of negative partial derivatives (with
    // respect to each endpoint coordinate) of snake's image energy.
    // Remember, the force on parameter x is -dE/dx.

    status = true;
    for (int i=0; i < snake.n(); i++)
    {
        force[i] = Vector2D(0,0);
        int n=0, dummy=0;

        for (int j=0; j < snake.n(); j++)
        {
            force[i].x -= denergy_dp (snake, snake_image_derivs_fn,
                                      j, i, &Vector2D::x, n);
            if (! status) return false;
            force[i].y -= denergy_dp (snake, snake_image_derivs_fn,
                                      j, i, &Vector2D::y, dummy);
            if (! status) return false;
```

```
        }
        force[i] /= n;
    }
    return true;
}
```

Source file `line-bending.C`

```
#include "line-force.h"

static int index (Cubic_Spline const &snake, int i)
    {return (i + snake.n()) % snake.n();}
static Vector2D const &node (Cubic_Spline const &snake, int i)
    {return snake [index(snake,i)].p[0];}


static double energy (const Cubic_Spline &snake, int i)
{
    // Return the bending energy for snake line segment i.  "Bending energy"
    // is squared magnitude of 2nd difference of node position.  (Sort of a
    // discrete version of squared 2nd derivative, which is curvature.)

    return norm2 ((node(snake,i+1) - node(snake,i))
                  - (node(snake,i) - node(snake,i-1)));
}


static double denergy_dp (const Cubic_Spline &snake, int i, int j,
                          double Vector2D::*x)
{
    // Return the change in (derivative of) bending energy for snake line
    // segment i with respect to segment j's p[0].*x.

    int dist = index (snake, j-i);
    // Only self and neighboring nodes affected:
    if (dist>1 && dist<snake.n()-1) return 0;

    double diff2 = 2 * (node(snake,i-1).*x - 2*node(snake,i).*x
                        + node(snake,i+1).*x);
    return dist==0? -2*diff2: diff2;
}


bool line_bending_energy (const Cubic_Spline &snake, double &e)
{
    e = 0;
    for (int i=0; i < snake.n(); i++)
        e += energy (snake, i);
    return true;
}


bool line_bending_force (const Cubic_Spline &snake, Vector2D *force)
{
    // Accept an array of snake segments.  Return an array of negated partial
    // derivatives (with respect to each endpoint coordinate) of sum of node
    // bending energies.  Remember, the force on parameter x is -dE/dx.

    for (int i=0; i < snake.n(); i++)
    {
        force[i].x = 0;
        force[i].y = 0;

        for (int j=0; j < snake.n(); j++)
        {
            force[i].x -= denergy_dp (snake, j, i, &Vector2D::x);
            force[i].y -= denergy_dp (snake, j, i, &Vector2D::y);
        }
```

```
    }
    return true;
}
```

Source file `energy.C`

```
#include <string-utils.h>
#include "all-forces.h"
#include "snake-utils.h"

#if defined(VMS) || defined(__VMS)
const char *gauss_fft_file = "RTP:[Fenster.Data.Gauss-FFTs]";
#else
const char *gauss_fft_file = "/u/boat/fenster/Medical/Data/Gauss-FFTs";
#endif


// ---------------------------------------------------------------------------
int main (int argc, char *argv[])
{
    Array_Owner<char> err_msg;
    if (argc != 6) {
        cout<<"Usage: "<<argv[0]
            <<" <force>"
            <<" <image_file>"
            <<" <contour_file>"
            <<" <training_file>"
            <<" <scale>"
            <<endl;
        return 1;
    }

    char *force_name = argv[1];
    char *image_filename = argv[2];
    char *contour_filename = argv[3];
    char *training_filename = argv[4];
    char *scale_s = argv[5];

    Force *force = get_force (force_name);
    if (!force) return 1;

    double scale;
    if (! string_to_double (scale_s, scale)) {
        cerr << scale_s << ": Not a valid scale" << endl;
        return 1;
    }

    // ---
    Line_Snake snake;
    int slice;
    if (! snake.read (contour_filename, slice, err_msg))
        {cerr << err_msg.p() << endl;  return 1;}
    if (slice == -1) {
        cerr<<contour_filename<<": No \"Image section #\" header."<<endl;
        return 1;
    }
    // Make snake be clockwise:
    double winding = winding_number (snake);
    if (winding < -0.5)
        reverse (snake, snake);
    if (abs (abs(winding)-1) > 0.01)
        cerr << "Warning:  Snake's winding number is " << winding << endl;

    // ---
    Image image;
    image = get_image (image_filename, slice, slice);
    if (! image.valid()) return 1;
```

```
    // ---
    double training_scale;
    if (! force->training().read (training_filename, training_scale, err_msg))
        {cerr << err_msg.p() << endl;  return 1;}

    if (! force->preprocess_image (image [slice], scale,
                                   image_filename, slice))
        {cerr << "Error: Couldn't preprocess slice." << endl;  return 1;}

    if (! force->preprocess_snake (snake.spline))
        {cerr << "Error: Couldn't preprocess snake." << endl;  return 1;}

    double e_b, e_i;
    snake.set (force);
    if (! snake.bending_energy (e_b))
        {cerr<<"Bending energy error"<<endl;  return 1;}
    if (! snake.image_energy (image[slice], e_i))
        {cerr<<"Image energy error"<<endl;  return 1;}

    cout << e_b << " " << e_i << endl;
    return 0;
}
```

Source file `train.C`

```
#include <string-utils.h>
#include "all-forces.h"
#include "snake-utils.h"

#if defined(VMS) || defined(__VMS)
const char *gauss_fft_file = "RTP:[Fenster.Data.Gauss-FFTs]";
#else
const char *gauss_fft_file = "/u/boat/fenster/Medical/Data/Gauss-FFTs";
#endif


// ----------------------------------------------------------------------------
int main (int argc, char *argv[])
{
    Array_Owner<char> err_msg;
    if (argc != 6)
    {
        cout<<"Usage: "<<argv[0]
            <<" <force>"
            <<" <image_file>"
            <<" <contour_file>"
            <<" <scale>"
            <<" <training_file>"
            <<endl;
        return 1;
    }

    char *force_name = argv[1];
    char *image_filename = argv[2];
    char *contour_filename = argv[3];
    char *scale_s = argv[4];
    char *training_filename = argv[5];

    Force *force = get_force (force_name);
    if (!force) return 1;

    double scale;
    if (! string_to_double (scale_s, scale)) {
        cerr << scale_s << ": Not a valid scale" << endl;
        return 1;
    }

    Line_Snake snake;
```

```
    int slice;
    if (! snake.read (contour_filename, slice, err_msg))
    {
        cerr << err_msg.p() << endl;
        return 1;
    }
    if (slice == -1)
    {
        cerr<<contour_filename<<": No \"Image section #\" header."<<endl;
        return 1;
    }
    // Make snake be clockwise:
    double winding = winding_number (snake);
    if (winding < -0.5)
        reverse (snake, snake);
    if (abs (abs(winding)-1) > 0.01)
        cerr << "Warning:  Snake's winding number is " << winding << endl;

    Image image;
    image = get_image (image_filename, slice, slice);
    if (! image.valid()) return 1;

    if (! force->preprocess_image (image [slice], scale,
                                   image_filename, slice)) {
        cerr << "Error: Couldn't preprocess slice." << endl;
        return 1;
    }
    if (! force->preprocess_snake (snake.spline)) {
        cerr << "Error: Couldn't preprocess snake." << endl;
        return 1;
    }

    double file_scale;
    if (! force->training().read (training_filename, file_scale, err_msg))
        cerr<<"[Creating new training file "<<training_filename<<"]"<<endl;

    if (! force->train (snake.spline)) return 1;

    if (! force->training().write (training_filename, err_msg))
        cerr << err_msg.p() << endl;
    return 0;
}
```

Source file `combine-train.C`

```
#include "all-forces.h"

#if defined(VMS) || defined(__VMS)
const char *gauss_fft_file = "RTP:[Fenster.Data.Gauss-FFTs]";
#else
const char *gauss_fft_file = "/u/boat/fenster/Medical/Data/Gauss-FFTs";
#endif


// ---------------------------------------------------------------------------
int main (int argc, char *argv[])
{
    Array_Owner<char> err_msg;
    if (argc < 4)
    {
        cout<<"Usage: "<<argv[0]
            <<" <force>"
            <<" <src_training_file> [<src_training_file>  [...]]"
            <<" <dest_training_file>"
            <<endl;
        return 0;
    }
```

```
    char *force_name = argv[1];
    Force *force = get_force (force_name);
    if (!force) return 1;

    Training *training = &force->training();
    Training *output_training = &get_force(force_name)->training();

    char *filename = argv[2];
    double scale;
    if (! output_training->read (filename, scale, err_msg)) {
        cerr << filename << ": " << err_msg.p() << endl;
        return 1;
    }

    for (int i=3; i<argc-1; i++)
    {
        filename = argv[i];
        if (! training->read (filename, scale, err_msg)) {
            cerr << filename << ": " << err_msg.p() << endl;
            return 1;
        }

        if (! output_training->add (*training)) {
            cerr << filename << ": " << "Couldn't combine" << endl;
            return 1;
        }
    }

    filename = argv[i];
    if (! output_training->write (filename, err_msg))
        cerr << err_msg.p() << endl;
    return 0;
}
```

Source file `xsnake.C`

```
// Main program for xsnake, which uses xlib, Xt and the Motif widgets
// to draw a finite element snake.
// Sam Fenster, 7/11/94

#include "snake.h"
#include "all-forces.h"
#include "snake-utils.h"

#include <X11/...>

#include <stdlib.h>
#include <mathfix.h>
#include <strstream.h>
#include <array_owner.h>
#include <array_io.h>
#include <string-utils.h>
#include <file-utils.h>

#include <xdraw.h>
#include <draw.h>
#include <med-image.h>

...

// --- Globals:
static Force *force = &sector_force;   // Opaque public interface object

C2_Snake c2_snake;
Line_Snake line_snake;
Spline_Snake *snake;  // Set to one of the above

Image image;
```

```
bool is_snake=false;


// ----------------------------------------------------------------------------

int main (int argc, char *argv[])
{
   ...
   XtAppMainLoop (app_context); // Loop forever.
   return 0;                    // Not reached.
}



...


// ----------------------------------------------------------------------------

void redraw_ac (Widget w, XEvent *, String *, Cardinal *)
{
   // Draw image:
   ...

    // Draw snake:
    if (is_snake) snake->draw (draw_env(w), app_data.dot_size);
    if (is_partial_snake)
    {
        snake->draw_partial (draw_env(w), app_data.dot_size);
        snake->rubber_band (draw_env(w), app_data.dot_size, snake_end);
    }
}



// ----------------------------------------------------------------------------
void make_snake_clockwise()
{
    // Make snake be clockwise:
    double winding = winding_number (*snake);
    if (winding < -0.5)
        reverse (*snake, *snake);
    if (abs (abs(winding)-1) > 0.01)
        cerr << "Warning:  Snake's winding number is " << winding << endl;
}


// ----------------------------------------------------------------------------

static const char * const snake_overrides_description = "\
MB2 = Click to draw next snaxel,  \
MB3 = Click to close snake.";

static const char * const snake_overrides = "\
 <Motion>: drag_snake() \n\
 <Btn2Down>: click_snake() \n\
 <Btn3Down>: close_snake()";
// <Btn2Down>(2): close_snake()

static const Point2D o (0,0);


void beg_snake_ac (Widget w, XEvent *event, String *, Cardinal *)
{
    XtOverrideTranslations (w, XtParseTranslationTable (snake_overrides));
    display_info (mouse_info_w, snake_overrides_description);
    turn_off_all ();
    turn_off_snake ();
    snake->start();
    Point2D p = image_r.scale (XPoint1 (event->xbutton), win_r);
    snake->add_node (p);
    snake_end = p;
```

```
    snake->rubber_band (draw_env(w), app_data.dot_size, snake_end);
    is_partial_snake=true;

    display_contour_info ("Contour currently being drawn", "");
    display_energy_info (false,0,0,0);
}



// ---------------------------------------------------------------------------

void drag_snake_ac (Widget w, XEvent *event, String *, Cardinal *)
{
    snake->rubber_band (draw_env(w), app_data.dot_size, snake_end);
    snake_end = image_r.scale (XPoint1 (event->xbutton), win_r);
    snake->rubber_band (draw_env(w), app_data.dot_size, snake_end);
}



// ---------------------------------------------------------------------------

void click_snake_ac (Widget w, XEvent *event, String *, Cardinal *)
{
    snake->rubber_band (draw_env(w), app_data.dot_size, snake_end);
    Point2D p = image_r.scale (XPoint1 (event->xbutton), win_r);
    snake_end = p;
    snake->rubber_band (draw_env(w), app_data.dot_size, snake_end);
    snake->add_node (p);
    snake->rubber_band (draw_env(w), app_data.dot_size, snake_end);
}



// ---------------------------------------------------------------------------

void close_snake_ac (Widget w, XEvent *, String *, Cardinal *)
{
    turn_off_partial_snake();
    snake->finish();
    if (snake->is_complete())
    {
        is_snake = true;
        snake->draw (draw_env (w), app_data.dot_size);
        display_contour_info ("Unsaved contour", "");
    }
}



// ---------------------------------------------------------------------------

void snake_cb (Widget, XtPointer client_data, XtPointer)
{
    int item = int (client_data);
    if (item == menus.snake.deform)
    {
        if (! is_snake) {
            cerr << "There is no current snake to deform!" << endl;
            return;
        }
        if (! force->preprocess_image (image [app_data.slice], app_data.scale,
                                   image_filename, app_data.slice)) {
            cerr << "Error: Couldn't preprocess slice." << endl;
            return;
        }

        int image_e, bend_e;
        XtVaGetValues (bending_energy_scale_w, XmNvalue, &bend_e, 0);
        XtVaGetValues (image_energy_scale_w, XmNvalue, &image_e, 0);
        const double e=.1;      // Arbitrary scale factor
        double w1 = e*bend_e/100.;
        double w2 = e*image_e/100.;
```

```
    // Turn off old snake:
    snake->draw (draw_env (picture_w), app_data.dot_size);

    const double cap = 2.5; // Move nodes no more than this many pixels,
                            // for stability and convergence:
    snake->set (force);
    make_snake_clockwise();
    if (! force->preprocess_snake (snake->spline))  {
        cerr << "Error: Couldn't preprocess snake." << endl;
        return;
    }
    double e_i_old=0, e_i_new=0, e_b;
    if (app_data.debug_level > 1) snake->debug(1);
    if (app_data.debug_level > 0)
        if (! snake->image_energy (image [app_data.slice], e_i_old))
            cerr << "Old energy err" << endl;
    if (snake->apply_force (w1, w2, cap, image [app_data.slice]))
    {
        if (!snake->image_energy(image[app_data.slice],e_i_new))
            cerr << "New energy err" << endl;
        if (!snake->bending_energy(e_b))
            cerr << "Bending energy err" << endl;
        if (app_data.debug_level > 0)
            cerr<<"Image energy: "<<e_i_old<<" -> "<<e_i_new<<endl;
        display_energy_info (true, e_b, e_i_new, w1*e_b + w2*e_i_new);
        display_contour_info ("Deformed (unsaved) contour", "");
    }

    // Draw new snake:
    snake->draw (draw_env (picture_w), app_data.dot_size);
}


else if (item==menus.snake.save_snake)
    pop_file_box (save_snake_w, shell_w, "saveSnake",
                  "*-*-*", save_snake_cb);

else if (item == menus.snake.load_snake)
    pop_file_box (load_snake_w, shell_w, "loadSnake",
                  "*-*-*", load_snake_cb);

else if (item == menus.snake.train)
{
    if (! is_snake)
    {
        cerr << "There is no current snake to deform!" << endl;
        return;
    }
    if (! force->preprocess_image (image [app_data.slice], app_data.scale,
                                   image_filename, app_data.slice))
    {
        cerr << "Error: Couldn't preprocess slice." << endl;
        return;
    }
    make_snake_clockwise();
    if (! force->preprocess_snake (snake->spline))
    {
        cerr << "Error: Couldn't preprocess snake." << endl;
        return;
    }

    if (force->train (snake->spline))
        display_training_info ("Unsaved training");
}


else if (item == menus.snake.write_training)
    pop_file_box (write_training_w, shell_w, "writeTraining",
                  "*", write_training_cb);

else if (item == menus.snake.read_training)
    pop_file_box (read_training_w, shell_w, "readTraining",
```

```
                      "*", read_training_cb);

    else if (item == menus.snake.forget_training)
    {
        force->training().forget();
        display_training_info ("No training");
    }

    else
        cerr << "Unknown item selected from Snake menu!" << endl;
}

...
```

B.3 Code defining training

Source file `avg-i-d-force.h`

```
#ifndef AID_FORCE_H
#define AID_FORCE_H

#include <array_owner.h>
#include "force.h"
#include "blur.h"

struct AID_Contour_Training_fields {
    double intensity_mean;
    double intensity_variance;
    double directional_gradient_mean;
    double directional_gradient_variance;
    long num_contours, num_pixels;
    double scale;
};

class Avg_I_D_Force;

class AID_Contour_Training: public Training {
    bool is_trained_d;
    AID_Contour_Training_fields data;
    Array_Owner<char> contributors;

public:
    AID_Contour_Training (): is_trained_d(false) {}
    AID_Contour_Training (AID_Contour_Training const &t) {*this = t;}
    AID_Contour_Training &operator = (AID_Contour_Training const &);

    friend class Avg_I_D_Force;

    char const *type () const {return "snake intensity/gradient training";}
    bool add (Training const &);
    bool read (const char *file_name,
               double &scale, Array_Owner<char> &err_msg);
    bool write (const char *file_name, Array_Owner<char> &err_msg) const;
    void forget () {is_trained_d = false;}
    bool is_trained () const {return is_trained_d;}
};


class Avg_I_D_Force: public Force {
    AID_Contour_Training my_training;

    bool is_cache_valid;
    unsigned short (*force_cache) [IMAGE_WIDTH];
    struct {
        Array_Owner<char> image_filename;
```

```
        int slice;
        double scale;
        image_t *image;
    } cache_info;

    double m (int x, int y) {return force_cache [y] [x];}
    bool make_blurred ();

public:
    Avg_I_D_Force (): is_cache_valid(false), force_cache(new blur_cache_t) {}
    ~Avg_I_D_Force () {delete [] force_cache;}

    Training &training () {return my_training;}
    Training const &training () const {return my_training;}

    bool preprocess_image (image_t &im, double scale,
                           char const *file_name, int slice);
    bool preprocess_snake (const Cubic_Spline &) {return true;}

    bool energy (double &e,
                 const Cubic_Spline &snake, int segment, double u);
    Snake_Image_Fn calc_snake_image_fn () const;

    bool force (Vector2D &dE_dp, Vector2D &dE_dn,
                const Cubic_Spline &snake, int segment, double u);
    Snake_Image_Derivs_Fn calc_snake_image_derivs () const;

    bool train (const Cubic_Spline &);
};

#endif
```

Source file **avg-i-d-force.C**

```
#include "avg-i-d-force.h"

#include <iostream.h>
#include <strstream.h>
#include <bool.h>
#include <mathfix.h>
#include <array_io.h>
#include <string-utils.h>


static Avg_I_D_Force *global_force;


// ---------------------------------------------------------------------------
static bool different (double a, double b)
    {return abs (a - b) / b  >  0.01;}


// ---------------------------------------------------------------------------
AID_Contour_Training &AID_Contour_Training::operator =
    (AID_Contour_Training const &t)
{
    is_trained_d = t.is_trained_d;
    if (is_trained()) {
        data = t.data;
        contributors = copy_string (t.contributors.p());
    }
    return *this;
}


// ---------------------------------------------------------------------------
bool Avg_I_D_Force::preprocess_image (image_t &image, double scale,
                                      char const *file_name, int slice)
```

```
{
    global_force = this;
    if (! streq (file_name, cache_info.image_filename) ||
        slice != cache_info.slice ||
        different (scale, cache_info.scale))
        is_cache_valid = false;

    cache_info.image = &image;
    if (is_cache_valid) return true;

    cache_info.image_filename = copy_string (file_name);
    cache_info.slice = slice;
    cache_info.scale = scale;
    return true;
}


// ---------------------------------------------------------------------------
bool Avg_I_D_Force::make_blurred ()
{
    if (! is_cache_valid)
        is_cache_valid = blur (*cache_info.image, cache_info.scale,
                               cache_info.image_filename, cache_info.slice,
                               force_cache);
    return is_cache_valid;
}


// ---------------------------------------------------------------------------
static inline Vector2D normal (const Vector2D &v)
    {return unit (rot90 (v));}

// ---------------------------------------------------------------------------
// Image energy function at a point:

bool Avg_I_D_Force::energy (double &e,
                            const Cubic_Spline &snake, int segment, double u)
{
    if (! make_blurred())
        return false;
    if (! training().is_trained()) {
        cerr << "No training yet!" << endl;
        return false;
    }
    if (different (my_training.data.scale, cache_info.scale)) {
        cerr<<"Training is not at current scale"<<endl;
        return false;
    }

    // xxx Bilinearly interpolate?
    Vector2D p = snake[segment](u);
    Vector2D n = normal (snake[segment].d_du(u));
    int y=nint(p.y), x=nint(p.x);

    double I = m(x,y);
    // xxx Illegal array access if at edge of image!
    Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2, (m(x,y+1) - m(x,y-1)) / 2);

    e = sqr (I - my_training.data.intensity_mean)
        / my_training.data.intensity_variance;
    e += sqr (dot(n,dI_dp) - my_training.data.directional_gradient_mean)
        / my_training.data.directional_gradient_variance;
    return true;
}


// ---------------------------------------------------------------------------
// Derivative of image energy function with respect to position and normal
// (orientation) at a point:

bool Avg_I_D_Force::force (Vector2D &dE_dp, Vector2D &dE_dn,
                           const Cubic_Spline &snake, int segment, double u)
{
```

```
    if (! make_blurred())
        return false;
    if (! training().is_trained()) {
        cerr << "No training yet!" << endl;
        return false;
    }
    if (different (my_training.data.scale, cache_info.scale)) {
        cerr<<"Training is not at current scale"<<endl;
        return false;
    }

    // xxx Bilinearly interpolate?
    Vector2D p = snake[segment](u);
    Vector2D n = normal (snake[segment].d_du(u));
    int y=nint(p.y), x=nint(p.x);

    double I = m(x,y);
    // xxx Illegal array access if at edge of image!
    Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2, (m(x,y+1) - m(x,y-1)) / 2);
    double d2I_dxdy = (m(x+1,y+1) - m(x-1,y+1) - m(x+1,y-1) + m(x-1,y-1)) / 2;
    Vector2D d2I_dpdx (m(x+1,y) - 2*m(x,y) + m(x-1,y), d2I_dxdy);
    Vector2D d2I_dpdy (d2I_dxdy, m(x,y+1) - 2*m(x,y) + m(x,y-1));

    dE_dp = 2 * (I - my_training.data.intensity_mean)
        / my_training.data.intensity_variance
        * dI_dp;

    double k = 2 * (dot(n,dI_dp) - my_training.data.directional_gradient_mean)
        / my_training.data.directional_gradient_variance;
    dE_dp += k * Vector2D (dot(d2I_dpdx,n), dot(d2I_dpdy,n));
    dE_dn = k * dI_dp;
    return true;
}


// ---------------------------------------------------------------------------
static bool global_energy_fn (double &e,
    const Cubic_Spline &snake, int segment, double u)
    {return global_force->energy (e, snake, segment, u);}

Snake_Image_Fn Avg_I_D_Force::calc_snake_image_fn () const
    {return global_energy_fn;}


static bool global_force_fn (Vector2D &dE_dp, Vector2D &dE_dn,
    const Cubic_Spline &snake, int segment, double u)
    {return global_force->force (dE_dp, dE_dn, snake, segment, u);}

Snake_Image_Derivs_Fn Avg_I_D_Force::calc_snake_image_derivs () const
    {return global_force_fn;}


// ---------------------------------------------------------------------------
...


// ---------------------------------------------------------------------------
bool AID_Contour_Training::add (Training const &t0)
{
    if (! streq (t0.type(), type())) return false;
    AID_Contour_Training const &t = (AID_Contour_Training const &) t0;
    if (! t.is_trained()) return false;
    if (! is_trained()) {*this = t; return true;}
    if (different (data.scale, t.data.scale)) {
        cerr << "AID_aggregate:  "
             << "Training sets are at different scales." << endl;
        return false;
    }
    long num1 = data.num_pixels;
    double I_sum1 = data.intensity_mean * num1;
    double dI_sum1 = data.directional_gradient_mean * num1;
    double I_squared_sum1 = num1 * (data.intensity_variance
                                + sqr (data.intensity_mean));
```

```
    double dI_squared_sum1 = num1 * (data.directional_gradient_variance
                                     + sqr (data.directional_gradient_mean));

    long num2 = t.data.num_pixels;
    double I_sum2 = t.data.intensity_mean * num2;
    double dI_sum2 = t.data.directional_gradient_mean * num2;
    double I_squared_sum2 = num2 * (t.data.intensity_variance
                                    + sqr (t.data.intensity_mean));
    double dI_squared_sum2 = num2 * (t.data.directional_gradient_variance
                                     + sqr (t.data.directional_gradient_mean));

    AID_Contour_Training training;
    training.data.intensity_mean
        = (I_sum1 + I_sum2) / (num1 + num2);
    training.data.intensity_variance
        = (I_squared_sum1 + I_squared_sum2) / (num1 + num2)
          -  sqr (training.data.intensity_mean);
    training.data.directional_gradient_mean
        = (dI_sum1 + dI_sum2) / (num1 + num2);
    training.data.directional_gradient_variance
        = (dI_squared_sum1 + dI_squared_sum2) / (num1 + num2)
          -  sqr (training.data.directional_gradient_mean);
    training.data.num_pixels = num1 + num2;
    training.is_trained_d = true;

    training.data.num_contours = data.num_contours + t.data.num_contours;
    training.data.scale = data.scale;

    ...
    *this = training;
}


// ----------------------------------------------------------------------------
bool Avg_I_D_Force::train (const Cubic_Spline &snake)
{
    if (! make_blurred())
        return false;

    long num=0;
    double I_sum=0, dI_sum=0, I_squared_sum=0, dI_squared_sum=0;

    for (int i=0; i < snake.n(); i++)
    {
        for (double u=0; u<1;)
        {
            Vector2D p = snake[i](u);
            int y=nint(p.y), x=nint(p.x);
            double I = m(x,y);

            Vector2D n = normal (snake[i].d_du(u));
            Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2,
                            (m(x,y+1) - m(x,y-1)) / 2);
            double dI = dot(n,dI_dp);

            I_sum += I; dI_sum += dI;
            I_squared_sum += sqr(I); dI_squared_sum += sqr(dI);
            num ++;
            u += 1/norm (snake[i].d_du(u));  // Move by 1 pixel.
        }
    }
    AID_Contour_Training new_training;
    new_training.data.intensity_mean = I_sum / num;
    new_training.data.intensity_variance
        = I_squared_sum / num  -  sqr (I_sum / num);
    new_training.data.directional_gradient_mean = dI_sum / num;
    new_training.data.directional_gradient_variance
        = dI_squared_sum / num  -  sqr (dI_sum / num);
    new_training.data.num_pixels = num;
    new_training.data.num_contours = 1;
    new_training.data.scale = cache_info.scale;
    new_training.is_trained_d = true;
```

```
    return my_training.add (new_training);
}

// ---------------------------------------------------------------------------

bool AID_Contour_Training::write (const char *file_name,
                                  Array_Owner<char> &err_msg) const
{
    ...
}

// ---------------------------------------------------------------------------

bool AID_Contour_Training::read (const char *file_name, double &scale,
                                 Array_Owner<char> &err_msg)
{
    ...
}
```

Source file `cov-i-d-force.h`

```
#ifndef CID_FORCE_H
#define CID_FORCE_H

#include <array_owner.h>
#include "force.h"
#include "blur.h"

struct CID_Contour_Training_fields {
    double intensity_mean;
    double intensity_std_dev;
    double directional_gradient_mean;
    double directional_gradient_std_dev;
    double correlation;
    long num_contours, num_pixels;
    double scale;
};

class Cov_I_D_Force;

class CID_Contour_Training: public Training {
    bool is_trained_d;
    CID_Contour_Training_fields data;
    Array_Owner<char> contributors;

public:
    CID_Contour_Training (): is_trained_d(false) {}
    CID_Contour_Training (CID_Contour_Training const &t) {*this = t;}
    CID_Contour_Training &operator = (CID_Contour_Training const &);

    friend class Cov_I_D_Force;

    char const *type () const {return "Covariant intensity/gradient training";}
    bool add (Training const &);
    bool read (const char *file_name,
               double &scale, Array_Owner<char> &err_msg);
    bool write (const char *file_name, Array_Owner<char> &err_msg) const;
    void forget () {is_trained_d = false;}
    bool is_trained () const {return is_trained_d;}
};


class Cov_I_D_Force: public Force {
    CID_Contour_Training my_training;

    bool is_blur_cache_valid;
```

```
    unsigned short (*blur_cache) [IMAGE_WIDTH];
    struct {
        Array_Owner<char> image_filename;
        int slice;
        double scale;
        image_t *image;
    } blur_cache_info;

    double m (int x, int y) {return blur_cache [y] [x];}
    bool make_blurred ();

public:
    Cov_I_D_Force (): is_blur_cache_valid(false),
        blur_cache(new blur_cache_t) {}
    ~Cov_I_D_Force () {delete [] blur_cache;}

    Training &training () {return my_training;}
    Training const &training () const {return my_training;}

    bool preprocess_image (image_t &im, double scale,
                           char const *file_name, int slice);
    bool preprocess_snake (const Cubic_Spline &) {return true;}

    bool energy (double &e,
                 const Cubic_Spline &snake, int segment, double u);
    Snake_Image_Fn calc_snake_image_fn () const;

    bool force (Vector2D &dE_dp, Vector2D &dE_dn,
                const Cubic_Spline &snake, int segment, double u);
    Snake_Image_Derivs_Fn calc_snake_image_derivs () const;

    bool train (const Cubic_Spline &);
};

#endif
```

Source file `cov-i-d-force.C`

```
#include "cov-i-d-force.h"

#include <iostream.h>
#include <strstream.h>
#include <bool.h>
#include <mathfix.h>
#include <array_io.h>
#include <string-utils.h>
#include <2D.h>


static Cov_I_D_Force *global_force;


// ---------------------------------------------------------------------------
static bool different (double a, double b)
    {return abs (a - b) / b  >  0.01;}


// ---------------------------------------------------------------------------
CID_Contour_Training &CID_Contour_Training::operator =
    (CID_Contour_Training const &t)
{
    is_trained_d = t.is_trained_d;
    if (is_trained()) {
        data = t.data;
        contributors = copy_string (t.contributors.p());
    }
    return *this;
}
```

```
// ---------------------------------------------------------------------------
bool Cov_I_D_Force::preprocess_image (image_t &image, double scale,
                                      char const *file_name, int slice)
{
    global_force = this;
    if (! streq (file_name, blur_cache_info.image_filename) ||
        slice != blur_cache_info.slice ||
        different (scale, blur_cache_info.scale))
        is_blur_cache_valid = false;

    blur_cache_info.image = &image;
    if (is_blur_cache_valid) return true;

    blur_cache_info.image_filename = copy_string (file_name);
    blur_cache_info.slice = slice;
    blur_cache_info.scale = scale;
    return true;
}


// ---------------------------------------------------------------------------
bool Cov_I_D_Force::make_blurred ()
{
    if (! is_blur_cache_valid)
        is_blur_cache_valid = blur (*blur_cache_info.image,
                                    blur_cache_info.scale,
                                    blur_cache_info.image_filename,
                                    blur_cache_info.slice,
                                    blur_cache);
    return is_blur_cache_valid;
}


// ---------------------------------------------------------------------------
static inline Vector2D normal (const Vector2D &v)
    {return unit (rot90 (v));}

// ---------------------------------------------------------------------------
// Image energy function at a point:

bool Cov_I_D_Force::energy (double &e,
                            const Cubic_Spline &snake, int segment, double u)
{
    if (! make_blurred())
        return false;
    if (! training().is_trained()) {
        cerr << "No training yet!" << endl;
        return false;
    }
    if (different (my_training.data.scale, blur_cache_info.scale)) {
        cerr<<"Training is not at current scale"<<endl;
        return false;
    }

    // xxx Bilinearly interpolate?
    Vector2D p = snake[segment](u);
    Vector2D n = normal (snake[segment].d_du(u));
    int y=nint(p.y), x=nint(p.x);

    double I = m(x,y);
    // xxx Illegal array access if at edge of image!
    Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2, (m(x,y+1) - m(x,y-1)) / 2);
    double DI = dot(n,dI_dp);

    double xI = (I - my_training.data.intensity_mean)
                / my_training.data.intensity_std_dev;
    double xD = (DI - my_training.data.directional_gradient_mean)
                / my_training.data.directional_gradient_std_dev;
    e = sqr(xI) + sqr(xD)  -  2 * xI * xD * my_training.data.correlation;
    return true;
}
```

```
// --------------------------------------------------------------------------
// Derivative of image energy function with respect to position and normal
// (orientation) at a point:

bool Cov_I_D_Force::force (Vector2D &dE_dp, Vector2D &dE_dn,
                           const Cubic_Spline &snake, int segment, double u)
{
    if (! make_blurred())
        return false;
    if (! training().is_trained()) {
        cerr << "No training yet!" << endl;
        return false;
    }
    if (different (my_training.data.scale, blur_cache_info.scale)) {
        cerr<<"Training is not at current scale"<<endl;
        return false;
    }

    // xxx Bilinearly interpolate?
    Vector2D p = snake[segment](u);
    Vector2D n = normal (snake[segment].d_du(u));
    int y=nint(p.y), x=nint(p.x);

    double I = m(x,y);
    // xxx Illegal array access if at edge of image!
    Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2, (m(x,y+1) - m(x,y-1)) / 2);
    double d2I_dxdy = (m(x+1,y+1) - m(x-1,y+1) - m(x+1,y-1) + m(x-1,y-1)) / 2;
    Vector2D d2I_dpdx (m(x+1,y) - 2*m(x,y) + m(x-1,y), d2I_dxdy);
    Vector2D d2I_dpdy (d2I_dxdy, m(x,y+1) - 2*m(x,y) + m(x,y-1));

    double DI = dot(n,dI_dp);
    Vector2D dD_dp = Vector2D (dot(d2I_dpdx,n), dot(d2I_dpdy,n));

    double xI = (I - my_training.data.intensity_mean)
        / my_training.data.intensity_std_dev;
    double xD = (DI - my_training.data.directional_gradient_mean)
        / my_training.data.directional_gradient_std_dev;

    Vector2D dI_dp_sI = dI_dp / my_training.data.intensity_std_dev;
    Vector2D dD_dp_sD = dD_dp / my_training.data.directional_gradient_std_dev;

    dE_dp = 2 * xI * dI_dp_sI;
    dE_dp += 2 * xD * dD_dp_sD;
    dE_dp += -2*my_training.data.correlation
            * (xI * dD_dp_sD + xD * dI_dp_sI);

    dE_dn = 2 * dI_dp / my_training.data.directional_gradient_std_dev
            * (xD - my_training.data.correlation * xI);

    return true;
}


// --------------------------------------------------------------------------
static bool global_energy_fn (double &e,
    const Cubic_Spline &snake, int segment, double u)
    {return global_force->energy (e, snake, segment, u);}

Snake_Image_Fn Cov_I_D_Force::calc_snake_image_fn () const
    {return global_energy_fn;}


static bool global_force_fn (Vector2D &dE_dp, Vector2D &dE_dn,
    const Cubic_Spline &snake, int segment, double u)
    {return global_force->force (dE_dp, dE_dn, snake, segment, u);}

Snake_Image_Derivs_Fn Cov_I_D_Force::calc_snake_image_derivs () const
    {return global_force_fn;}


// --------------------------------------------------------------------------
```

```
...

// ---------------------------------------------------------------------------
bool CID_Contour_Training::add (Training const &t0)
{
    if (! streq (t0.type(), type())) return false;
    CID_Contour_Training const &t = (CID_Contour_Training const &) t0;
    if (! t.is_trained()) return false;
    if (! is_trained()) {*this = t; return true;}
    if (different (data.scale, t.data.scale)) {
        cerr << "CID_aggregate:  "
             << "Training sets are at different scales." << endl;
        return false;
    }
    long num1 = data.num_pixels;
    double I_sum1 = data.intensity_mean * num1;
    double D_sum1 = data.directional_gradient_mean * num1;
    double I2_sum1 = num1 * (sqr(data.intensity_std_dev)
                         + sqr (data.intensity_mean));
    double D2_sum1 = num1 * (sqr(data.directional_gradient_std_dev)
                         + sqr (data.directional_gradient_mean));
    double ID_sum1 = num1
        * (data.intensity_std_dev * data.directional_gradient_std_dev
           * data.correlation
           +  data.intensity_mean * data.directional_gradient_mean);

    long num2 = t.data.num_pixels;
    double I_sum2 = t.data.intensity_mean * num2;
    double D_sum2 = t.data.directional_gradient_mean * num2;
    double I2_sum2 = num2 * (sqr(t.data.intensity_std_dev)
                         + sqr (t.data.intensity_mean));
    double D2_sum2 = num2 * (sqr(t.data.directional_gradient_std_dev)
                         + sqr (t.data.directional_gradient_mean));
    double ID_sum2 = num2
        * (t.data.intensity_std_dev * t.data.directional_gradient_std_dev
           * t.data.correlation
           +  t.data.intensity_mean * t.data.directional_gradient_mean);

    CID_Contour_Training training;
    training.data.intensity_mean
        = (I_sum1 + I_sum2) / (num1 + num2);
    training.data.intensity_std_dev
        = sqrt ((I2_sum1 + I2_sum2) / (num1 + num2)
              -  sqr (training.data.intensity_mean));
    training.data.directional_gradient_mean
        = (D_sum1 + D_sum2) / (num1 + num2);
    training.data.directional_gradient_std_dev
        = sqrt ((D2_sum1 + D2_sum2) / (num1 + num2)
              -  sqr (training.data.directional_gradient_mean));
    training.data.correlation
        = ((ID_sum1 + ID_sum2) / (num1 + num2)
           - training.data.intensity_mean
             * training.data.directional_gradient_mean)
          / (training.data.intensity_std_dev
             * training.data.directional_gradient_std_dev);
    training.data.num_pixels = num1 + num2;
    training.is_trained_d = true;

    training.data.num_contours = data.num_contours + t.data.num_contours;
    training.data.scale = data.scale;

    ...
    *this = training;
}


// ---------------------------------------------------------------------------
bool Cov_I_D_Force::train (const Cubic_Spline &snake)
{
    if (! make_blurred())
        return false;
```

```
    long num=0;
    double I_sum=0, D_sum=0, I2_sum=0, D2_sum=0, ID_sum=0;

    for (int i=0; i < snake.n(); i++)
    {
        for (double u=0; u<1;)
        {
            Vector2D p = snake[i](u);
            int y=nint(p.y), x=nint(p.x);
            double I = m(x,y);

            Vector2D n = normal (snake[i].d_du(u));
            Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2,
                            (m(x,y+1) - m(x,y-1)) / 2);
            double DI = dot(n,dI_dp);

            I_sum += I;
            D_sum += DI;
            I2_sum += sqr(I);
            D2_sum += sqr(DI);
            ID_sum += I*DI;
            num ++;
            u += 1/norm (snake[i].d_du(u));  // Move by 1 pixel.
        }
    }
    CID_Contour_Training new_training;
    double mI,sI,mD,sD,cID;
    new_training.data.intensity_mean = mI = I_sum / num;
    new_training.data.intensity_std_dev =
        sI = sqrt (I2_sum / num  -  sqr (mI));
    new_training.data.directional_gradient_mean = mD = D_sum / num;
    new_training.data.directional_gradient_std_dev =
        sD = sqrt (D2_sum / num  -  sqr (mD));
    new_training.data.correlation = cID = (ID_sum / num  -  mI*mD) / (sI*sD);
    new_training.data.num_pixels = num;
    new_training.data.num_contours = 1;
    new_training.data.scale = blur_cache_info.scale;
    new_training.is_trained_d = true;

    return my_training.add (new_training);
}


// ---------------------------------------------------------------------------
bool CID_Contour_Training::write (const char *file_name,
                                  Array_Owner<char> &err_msg) const
{
    ...
}


// ---------------------------------------------------------------------------

bool CID_Contour_Training::read (const char *file_name,
                                 double &scale_ret,
                                 Array_Owner<char> &err_msg)
{
    ...
}
```

Source file `sector-force.h`

```
#ifndef SECTOR_FORCE_H
#define SECTOR_FORCE_H

#include "spline_length.h"
#include <array_owner.h>
#include "force.h"
```

```
#include "blur.h"

class Sector_Contour_Training: public Training {
public:
    int num_sectors;
    Array_Owner<double> intensity_means;
    Array_Owner<double> intensity_variances;
    Array_Owner<double> directional_gradient_means;
    Array_Owner<double> directional_gradient_variances;
    Array_Owner<long> num_pixels;
    long num_contours;
    double scale;

    bool is_trained_d;
    Array_Owner<char> contributors;

public:
    Sector_Contour_Training (): is_trained_d(false) {}
    Sector_Contour_Training (int n);
    Sector_Contour_Training &operator = (Sector_Contour_Training const &);
    Sector_Contour_Training (Sector_Contour_Training const &t) {*this = t;}

    friend class Sector_Force;

    char const *type () const
        {return "snake sectored intensity/gradient training";}
    bool add (Training const &);
    bool read (const char *file_name,
               double &scale, Array_Owner<char> &err_msg);
    bool write (const char *file_name, Array_Owner<char> &err_msg) const;
    void forget () {is_trained_d = false;}
    bool is_trained () const {return is_trained_d;}
};


class Sector_Force: public Force {
    Sector_Contour_Training my_training;

    bool was_snake_ever_preprocessed;
    struct Spline_Length snake_length_cache;

    bool is_blur_cache_valid;
    unsigned short (*blur_cache) [IMAGE_WIDTH];
    struct {
        Array_Owner<char> image_filename;
        int slice;
        double scale;
        image_t *image;
    } blur_cache_info;

    double m (int x, int y) {return blur_cache [y] [x];}
    bool make_blurred ();
    void print_snake_preprocess_debug_info ();

public:
    Sector_Force (): was_snake_ever_preprocessed(false),
        is_blur_cache_valid(false), blur_cache(new blur_cache_t) {}
    ~Sector_Force () {delete [] blur_cache;}

    Training &training () {return my_training;}
    Training const &training () const {return my_training;}

    bool preprocess_image (image_t &im, double scale,
                           char const *file_name, int slice);
    bool preprocess_snake (const Cubic_Spline &);

    bool energy (double &e,
                 const Cubic_Spline &snake, int segment, double u);
    Snake_Image_Fn calc_snake_image_fn () const;

    bool force (Vector2D &dE_dp, Vector2D &dE_dn,
                const Cubic_Spline &snake, int segment, double u);
    Snake_Image_Derivs_Fn calc_snake_image_derivs () const;
```

```
    bool train (const Cubic_Spline &);

    void print_training_data (ostream &, const Cubic_Spline &);
};

#endif
```

Source file sector-force.C

```
#include "sector-force.h"

#include <iostream.h>
#include <strstream.h>
#include <bool.h>
#include <mathfix.h>
#include <array_io.h>
#include <string-utils.h>


static Sector_Force *global_force;


// ------------------------------------------------------------------------------
static bool different (double a, double b)
    {return abs (a - b) / b  >  0.01;}


// ------------------------------------------------------------------------------
Sector_Contour_Training::Sector_Contour_Training (int n): is_trained_d(false)
{
    num_sectors = n;
    intensity_means=new double[n];
    intensity_variances=new double[n];
    directional_gradient_means=new double[n];
    directional_gradient_variances=new double[n];
    num_pixels = new long[n];
    num_contours = 0;
    scale = -1;
    contributors = 0;
}

Sector_Contour_Training &Sector_Contour_Training::operator =
    (Sector_Contour_Training const &t)
{
    is_trained_d = t.is_trained_d;
    num_sectors = t.num_sectors;
    intensity_means
        = copy (t.intensity_means.p(), num_sectors);
    intensity_variances
        = copy (t.intensity_variances.p(), num_sectors);
    directional_gradient_means
        = copy (t.directional_gradient_means.p(), num_sectors);
    directional_gradient_variances
        = copy (t.directional_gradient_variances.p(), num_sectors);
    num_pixels
        = copy (t.num_pixels.p(), num_sectors);
    num_contours = t.num_contours;
    scale = t.scale;
    contributors = copy_string (t.contributors.p());
    return *this;
}


// ------------------------------------------------------------------------------
// This module wasn't designed with preprocess_snake() in mind -- it was a
// later addition.  So other routines still take a snake as an argument,
// instead of using a cached one.  So we just assume the caller promises to
```

```
// call this first.  A good sanity check is for the routines using the cached
// results of preprocess_snake() to check whether it has ever been called.

bool Sector_Force::preprocess_snake (Cubic_Spline const &snake)
{
    global_force = this;
    was_snake_ever_preprocessed = true;
    snake_length_cache.measure (snake);
    // print_snake_preprocess_debug_info ();
    return true;
}


// ---------------------------------------------------------------------------
bool Sector_Force::preprocess_image (image_t &image, double scale,
                                     char const *file_name, int slice)
{
    global_force = this;
    if (! streq (file_name, blur_cache_info.image_filename) ||
        slice != blur_cache_info.slice ||
        different (scale, blur_cache_info.scale))
        is_blur_cache_valid = false;

    blur_cache_info.image = &image;
    if (is_blur_cache_valid) return true;

    blur_cache_info.image_filename = copy_string (file_name);
    blur_cache_info.slice = slice;
    blur_cache_info.scale = scale;
    return true;
}


// ---------------------------------------------------------------------------
bool Sector_Force::make_blurred ()
{
    if (! is_blur_cache_valid)
        is_blur_cache_valid = blur (*blur_cache_info.image,
                                    blur_cache_info.scale,
                                    blur_cache_info.image_filename,
                                    blur_cache_info.slice,
                                    blur_cache);
    return is_blur_cache_valid;
}


// ---------------------------------------------------------------------------
static inline Vector2D normal (const Vector2D &v)
    {return unit (rot90 (v));}

// ---------------------------------------------------------------------------
// Image energy function at a point:

bool Sector_Force::energy (double &e,
                           Cubic_Spline const &snake, int segment, double u)
{
    if (! was_snake_ever_preprocessed) {
        cerr << "Snake not preprocessed!" << endl;
        return false;
    }
    if (! make_blurred())
        return false;
    if (! training().is_trained()) {
        cerr << "No training yet!" << endl;
        return false;
    }
    if (different (my_training.scale, blur_cache_info.scale)) {
        cerr<<"Training is not at current scale"<<endl;
        return false;
    }

    // xxx Bilinearly interpolate?
    Vector2D p = snake[segment](u);
```

```
    Vector2D n = normal (snake[segment].d_du(u));
    int y=nint(p.y), x=nint(p.x);

    double I = m(x,y);
    // xxx Illegal array access if at edge of image!
    Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2, (m(x,y+1) - m(x,y-1)) / 2);

    int j = snake_length_cache.around_frac (segment, u)
            * my_training.num_sectors;
    e = sqr (I - my_training.intensity_means[j])
        / my_training.intensity_variances[j];
    e += sqr (dot(n,dI_dp) - my_training.directional_gradient_means[j])
        / my_training.directional_gradient_variances[j];
    return true;
}


// ------------------------------------------------------------------------
// Derivative of image energy function with respect to position and normal
// (orientation) at a point:

bool Sector_Force::force (Vector2D &dE_dp, Vector2D &dE_dn,
                          const Cubic_Spline &snake, int segment, double u)
{
    if (! was_snake_ever_preprocessed) {
        cerr << "Snake not preprocessed!" << endl;
        return false;
    }
    if (! make_blurred())
        return false;
    if (! training().is_trained()) {
        cerr << "No training yet!" << endl;
        return false;
    }
    if (different (my_training.scale, blur_cache_info.scale)) {
        cerr<<"Training is not at current scale"<<endl;
        return false;
    }

    // xxx Bilinearly interpolate?
    Vector2D p = snake[segment](u);
    Vector2D n = normal (snake[segment].d_du(u));
    int y=nint(p.y), x=nint(p.x);

    double I = m(x,y);
    // xxx Illegal array access if at edge of image!
    Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2, (m(x,y+1) - m(x,y-1)) / 2);
    double d2I_dxdy = (m(x+1,y+1) - m(x-1,y+1) - m(x+1,y-1) + m(x-1,y-1)) / 2;
    Vector2D d2I_dpdx (m(x+1,y) - 2*m(x,y) + m(x-1,y), d2I_dxdy);
    Vector2D d2I_dpdy (d2I_dxdy, m(x,y+1) - 2*m(x,y) + m(x,y-1));

    int j = snake_length_cache.around_frac (segment, u)
            * my_training.num_sectors;
    dE_dp = 2 * (I - my_training.intensity_means[j])
        / my_training.intensity_variances[j]
        * dI_dp;

    double k = 2 * (dot(n,dI_dp) - my_training.directional_gradient_means[j])
        / my_training.directional_gradient_variances[j];
    dE_dp += k * Vector2D (dot(d2I_dpdx,n), dot(d2I_dpdy,n));
    dE_dn = k * dI_dp;
    return true;
}


// ------------------------------------------------------------------------
static bool global_energy_fn (double &e,
    const Cubic_Spline &snake, int segment, double u)
    {return global_force->energy (e, snake, segment, u);}

Snake_Image_Fn Sector_Force::calc_snake_image_fn () const
    {return global_energy_fn;}
```

```
static bool global_force_fn (Vector2D &dE_dp, Vector2D &dE_dn,
    const Cubic_Spline &snake, int segment, double u)
    {return global_force->force (dE_dp, dE_dn, snake, segment, u);}

Snake_Image_Derivs_Fn Sector_Force::calc_snake_image_derivs () const
    {return global_force_fn;}


// ----------------------------------------------------------------------------
bool Sector_Contour_Training::add (Training const &t0)
{
    if (! streq (t0.type(), type())) return false;
    Sector_Contour_Training const &t = (Sector_Contour_Training const &) t0;
    if (! t.is_trained()) return false;
    if (! is_trained()) {*this = t; return true;}
    if (different (scale, t.scale)) {
        cerr << "sector_aggregate error:  "
             << "Training sets are at different scales." << endl;
        return false;
    }
    if (num_sectors != t.num_sectors) {
        cerr << "sector_aggregate error:  "
             << "Training sets are for different numbers of sectors." << endl;
        return false;
    }

    Sector_Contour_Training training = *this;
    for (int i=0; i<num_sectors; i++) {
        long num1 = num_pixels[i];
        double I_sum1 = intensity_means[i] * num1;
        double dI_sum1 = directional_gradient_means[i] * num1;
        double I_squared_sum1 = num1 * (intensity_variances[i]
                                        + sqr (intensity_means[i]));
        double dI_squared_sum1
            = num1 * (directional_gradient_variances[i]
                      + sqr (directional_gradient_means[i]));

        long num2 = t.num_pixels[i];
        double I_sum2 = t.intensity_means[i] * num2;
        double dI_sum2 = t.directional_gradient_means[i] * num2;
        double I_squared_sum2 = num2 * (t.intensity_variances[i]
                                        + sqr (t.intensity_means[i]));
        double dI_squared_sum2
            = num2 * (t.directional_gradient_variances[i]
                      + sqr (t.directional_gradient_means[i]));

        training.intensity_means[i]
            = (I_sum1 + I_sum2) / (num1 + num2);
        training.intensity_variances[i]
            = (I_squared_sum1 + I_squared_sum2) / (num1 + num2)
            -  sqr (training.intensity_means[i]);
        training.directional_gradient_means[i]
            = (dI_sum1 + dI_sum2) / (num1 + num2);
        training.directional_gradient_variances[i]
            = (dI_squared_sum1 + dI_squared_sum2) / (num1 + num2)
            -  sqr (training.directional_gradient_means[i]);
        training.num_pixels[i] = num1 + num2;
    }
    training.num_contours = num_contours + t.num_contours;

    ...
    *this = training;
}


// ----------------------------------------------------------------------------
bool Sector_Force::train (const Cubic_Spline &snake)
{
    if (! was_snake_ever_preprocessed) {
        cerr << "Snake not preprocessed!" << endl;
        return false;
    }
```

```
    if (! make_blurred())
        return false;

    int num_sectors = training().is_trained()? my_training.num_sectors: 12;

    Array_Owner<double> I_sum (new double [num_sectors]);
    Array_Owner<double> I_squared_sum (new double [num_sectors]);
    Array_Owner<double> dI_sum (new double [num_sectors]);
    Array_Owner<double> dI_squared_sum (new double [num_sectors]);
    Array_Owner<long> num (new long [num_sectors]);

    int i;
    for (i=0; i<num_sectors; i++) {
        I_sum[i] = 0;  I_squared_sum[i] = 0;
        dI_sum[i] = 0;  dI_squared_sum[i] = 0;
        num[i] = 0;
    }
    for (i=0; i < snake.n(); i++)
    {
        for (double u=0; u<1;)
        {
            Vector2D p = snake[i](u);
            int y=nint(p.y), x=nint(p.x);
            double I = m(x,y);

            Vector2D n = normal (snake[i].d_du(u));
            Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2,
                            (m(x,y+1) - m(x,y-1)) / 2);
            double dI = dot(n,dI_dp);

            int j = snake_length_cache.around_frac (i,u) * num_sectors;
            I_sum[j] += I;  dI_sum[j] += dI;
            I_squared_sum[j] += sqr(I);  dI_squared_sum[j] += sqr(dI);
            num[j] ++;
            u += 1/norm (snake[i].d_du(u));  // Move by 1 pixel.
        }
    }
    Sector_Contour_Training new_training (num_sectors);
    for (int j=0; j<num_sectors; j++) {
        new_training.intensity_means[j] = I_sum[j] / num[j];
        new_training.intensity_variances[j]
            = I_squared_sum[j] / num[j]  -  sqr (I_sum[j] / num[j]);
        new_training.directional_gradient_means[j] = dI_sum[j] / num[j];
        new_training.directional_gradient_variances[j]
            = dI_squared_sum[j] / num[j]  -  sqr (dI_sum[j] / num[j]);
        new_training.num_pixels[j] = num[j];
    }
    new_training.num_contours = 1;
    new_training.scale = blur_cache_info.scale;
    new_training.is_trained_d = true;

    return my_training.add (new_training);
}


// ----------------------------------------------------------------------------
void Sector_Force::print_training_data (ostream &o, const Cubic_Spline &snake)
{
    if (! was_snake_ever_preprocessed) {
        cerr << "Snake not preprocessed!" << endl;
        return;
    }
    if (! make_blurred())
        return;

    int num_sectors = training().is_trained()? my_training.num_sectors: 12;

    for (int i=0; i < snake.n(); i++)
    {
        for (double u=0; u<1;)
        {
            Vector2D p = snake[i](u);
            int y=nint(p.y), x=nint(p.x);
```

```
            double I = m(x,y);

            Vector2D n = normal (snake[i].d_du(u));
            Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2,
                            (m(x,y+1) - m(x,y-1)) / 2);
            double dI = dot(n,dI_dp);

            int j = snake_length_cache.around_frac (i,u) * num_sectors;
            o<<j<<"  "<<I<<"  "<<dI<<endl;
            u += 1/norm (snake[i].d_du(u));  // Move by 1 pixel.
        }
    }
}


// ---------------------------------------------------------------------------
bool Sector_Contour_Training::write (const char *file_name,
                                     Array_Owner<char> &err_msg) const
{
    ...
}


// ---------------------------------------------------------------------------

bool Sector_Contour_Training::read (const char *file_name,
                                    double &scale_ret,
                                    Array_Owner<char> &err_msg)
{
    ...
}
```

Source file `trad-force.h`

```
#ifndef TRAD_FORCE_H
#define TRAD_FORCE_H

#include <array_owner.h>
#include "force.h"
#include "blur.h"

class Trad_Contour_Training: public Training {
public:
    Trad_Contour_Training () {}
    Trad_Contour_Training (Trad_Contour_Training const &t) {}
    Trad_Contour_Training &operator = (Trad_Contour_Training const &)
        {return *this;}

    char const *type () const {return "null training";}
    bool add (Training const &) {return true;}
    bool read (const char *file_name,
               double &scale, Array_Owner<char> &err_msg) {return true;}
    bool write (const char *file_name, Array_Owner<char> &err_msg) const
        {return true;}
    void forget () {}
    bool is_trained () const {return true;}
};


class Trad_Force: public Force {
    Trad_Contour_Training my_training;

    bool is_blur_cache_valid;
    unsigned short (*blur_cache) [IMAGE_WIDTH];
    struct {
        Array_Owner<char> image_filename;
        int slice;
        double scale;
```

```
        image_t *image;
    } blur_cache_info;

    double m (int x, int y) {return blur_cache [y] [x];}
    bool make_blurred ();

public:
    Trad_Force (): is_blur_cache_valid(false), blur_cache(new blur_cache_t) {}
    ~Trad_Force () {delete [] blur_cache;}

    Training &training () {return my_training;}
    Training const &training () const {return my_training;}

    bool preprocess_image (image_t &im, double scale,
                           char const *file_name, int slice);
    bool preprocess_snake (const Cubic_Spline &) {return true;}

    bool energy (double &e,
                 const Cubic_Spline &snake, int segment, double u);
    Snake_Image_Fn calc_snake_image_fn () const;

    bool force (Vector2D &dE_dp, Vector2D &dE_dn,
                const Cubic_Spline &snake, int segment, double u);
    Snake_Image_Derivs_Fn calc_snake_image_derivs () const;

    bool train (const Cubic_Spline &) {return true;}
};

#endif
```

Source file `trad-force.C`

```
#include "trad-force.h"

#include <iostream.h>
#include <strstream.h>
#include <bool.h>
#include <mathfix.h>
#include <array_io.h>
#include <string-utils.h>


static Trad_Force *global_force;


// ---------------------------------------------------------------------------
static bool different (double a, double b)
    {return abs (a - b) / b  >  0.01;}


// ---------------------------------------------------------------------------
bool Trad_Force::preprocess_image (image_t &image, double scale,
                                    char const *file_name, int slice)
{
    global_force = this;
    if (! streq (file_name, blur_cache_info.image_filename) ||
        slice != blur_cache_info.slice ||
        different (scale, blur_cache_info.scale))
        is_blur_cache_valid = false;

    blur_cache_info.image = &image;
    if (is_blur_cache_valid) return true;

    blur_cache_info.image_filename = copy_string (file_name);
    blur_cache_info.slice = slice;
    blur_cache_info.scale = scale;
    return true;
}
```

```
// ---------------------------------------------------------------------------
bool Trad_Force::make_blurred ()
{
    if (! is_blur_cache_valid)
        is_blur_cache_valid = blur (*blur_cache_info.image,
                                    blur_cache_info.scale,
                                    blur_cache_info.image_filename,
                                    blur_cache_info.slice,
                                    blur_cache);
    return is_blur_cache_valid;
}


// ---------------------------------------------------------------------------
// Image energy function at a point:

bool Trad_Force::energy (double &e,
                         Cubic_Spline const &snake, int segment, double u)
{
    if (! make_blurred())
        return false;

    // xxx Bilinearly interpolate?
    Vector2D p = snake[segment](u);
    int y=nint(p.y), x=nint(p.x);

    // xxx Illegal array access if at edge of image!
    Vector2D dI_dp ((m(x+1,y) - m(x-1,y)) / 2, (m(x,y+1) - m(x,y-1)) / 2);

    // Energy is gradient strength intersected by snake, negated so that
    // minimizing it puts the snake on the strongest edges.
    e = -norm(dI_dp);
    return true;
}


// ---------------------------------------------------------------------------
// Derivative of image energy function with respect to position and normal
// (orientation) at a point:

bool Trad_Force::force (Vector2D &dE_dp, Vector2D &dE_dn,
                        const Cubic_Spline &snake, int segment, double u)
{
    if (! make_blurred())
        return false;

    // xxx Bilinearly interpolate?
    Vector2D p = snake[segment](u);
    int y=nint(p.y), x=nint(p.x);

    // xxx Illegal array access if at edge of image!
    Vector2D gl ((m(x,y) - m(x-2,y)) / 2, (m(x-1,y+1) - m(x-1,y-1)) / 2);
    Vector2D gr ((m(x+2,y) - m(x,y)) / 2, (m(x+1,y+1) - m(x+1,y-1)) / 2);
    Vector2D gd ((m(x+1,y-1) - m(x-1,y-1)) / 2, (m(x,y) - m(x,y-2)) / 2);
    Vector2D gu ((m(x+1,y+1) - m(x-1,y+1)) / 2, (m(x,y+2) - m(x,y)) / 2);

    // Remember, image force is -dE/dp.
    dE_dp = Vector2D(-norm(gr) - -norm(gl), -norm(gu) - -norm(gd)) / 2;
    dE_dn = Vector2D(0,0);
    return true;
}


// ---------------------------------------------------------------------------
static bool global_energy_fn (double &e,
    const Cubic_Spline &snake, int segment, double u)
    {return global_force->energy (e, snake, segment, u);}

Snake_Image_Fn Trad_Force::calc_snake_image_fn () const
    {return global_energy_fn;}
```

```
static bool global_force_fn (Vector2D &dE_dp, Vector2D &dE_dn,
    const Cubic_Spline &snake, int segment, double u)
    {return global_force->force (dE_dp, dE_dn, snake, segment, u);}

Snake_Image_Derivs_Fn Trad_Force::calc_snake_image_derivs () const
    {return global_force_fn;}
```

Source file `all-forces.h`

```
#ifndef ALL_FORCES_H
#define ALL_FORCES_H

#include "trad-force.h"
#include "avg-i-d-force.h"
#include "sector-force.h"
#include "cov-i-d-force.h"

char *const TRAD_FORCE = "trad";
char *const AVG_I_D_FORCE = "avg-i-d";
char *const SECTOR_FORCE = "sector";
char *const COV_I_D_FORCE = "covariant";

extern Trad_Force trad_force;
extern Avg_I_D_Force avg_i_d_force;
extern Sector_Force sector_force;
extern Cov_I_D_Force covariant_force;

Force *get_force (char const *force_name);

#endif
```

Source file `blur.h`

```
#ifndef BLUR_H
#define BLUR_H

#include <med-image.h>

typedef unsigned short blur_cache_t [IMAGE_HEIGHT] [IMAGE_WIDTH];

bool blur (image_t &, double scale, char const *file_name, int slice,
           blur_cache_t);

#endif
```

Source file `blur.C`

```
#include "blur.h"

#include <string.h>
#include <iostream.h>
#include <strstream.h>
#include <bool.h>
#include <mathfix.h>
#include <fft.h>
#include <array_io.h>
```

```
#include <string-utils.h>
#include <file-utils.h>

#if !defined(VMS) && !defined(__VMS)
#define CACHE_SUFFIX ".Blur/"
#else
#define CACHE_SUFFIX ""
#endif


// ---------------------------------------------------------------------------
static bool different (double a, double b)
    {return abs (a - b) / b  >  0.01;}



...


// ---------------------------------------------------------------------------
// Put blur (image) into blur_cache:

bool blur (image_t &im, double scale, char const *file_name, int slice,
           blur_cache_t blur_cache)
{
    if (search_cache (file_name, slice, scale, blur_cache))
        return true;

    typedef complex gauss_fft_t [IMAGE_HEIGHT] [IMAGE_WIDTH];
    static complex (*gauss_fft) [IMAGE_WIDTH] = new gauss_fft_t;
    static double std_dev;
    static bool is_initialized=false;
    if (! is_initialized  ||  scale != std_dev)
    {
        Array_Owner<char> gauss_file_name (make_gauss_file_name (scale));
        Array_Owner<Array_File_Header_Item> headers;
        if (! read_array (gauss_file_name, headers,
                          gauss_fft, sizeof (gauss_fft_t), 1,
                          "Reading FFT of Gaussian..."))
            cerr << "Oops, couldn't read gaussian FFT file \""
                 << gauss_file_name << "\"." << endl;
        else if (! is_gauss_fft (headers, std_dev))
            cerr<<gauss_file_name<<": Not Gauss FFT file"<<endl;
        else if (different (scale, std_dev))
            cerr<<gauss_file_name<<": Name doesn't match scale"<<endl;
        else
        {
            cerr << "done." << endl;
            is_initialized = true;
        }
        if (! is_initialized) return false;
    }

    int i, x, y;
    int log_width, log_height;
    // Convolve IM with GAUSS_FFT, in several steps:

    // Copy IM to an array of complex:
    cerr << "Copying image..." << flush;
    complex (*c_image) [IMAGE_WIDTH] =
        new complex [IMAGE_HEIGHT] [IMAGE_WIDTH];
    for (x=0; x<IMAGE_WIDTH; x++)
        for (y=0; y<IMAGE_HEIGHT; y++)
            c_image [y] [x] = im [y] [x];

    // Replace C_IMAGE with its FFT:
    cerr << "FFT..." << endl;
    i=IMAGE_WIDTH; log_width=0; while (i != 1) {i >>= 1; log_width ++;}
    i=IMAGE_HEIGHT; log_height=0; while (i != 1) {i >>= 1; log_height ++;}
    fft_2D (&c_image[0][0], log_width, log_height);

    // Find product of image FFT with the gauss FFT:
    cerr << "...times blur FFT..." << endl;
```

```
    for (x=0; x<IMAGE_WIDTH; x++)
    {
        for (y=0; y<IMAGE_HEIGHT; y++)
            c_image [y] [x] *= gauss_fft [y] [x];
        if (x%10==0) cerr<<'x'<<flush;
    }
    cerr<<endl;

    // Do inverse FFT on each to get grad blurred norm grad image
    // in x and y directions:
    cerr << "...inverse FFT..." << endl;
    fft_2D (&c_image[0][0], log_width, log_height, true);

    // Copy force field from complex arrays to blur_cache:
    cerr << "...copying..." << flush;
    for (x=0; x<IMAGE_WIDTH; x++)
        for (y=0; y<IMAGE_HEIGHT; y++)
            blur_cache[y][x] = (unsigned short) real (c_image[y][x]);
    cerr << "done." << endl;
    delete [] c_image;

    write_cache (file_name, slice, scale, blur_cache);
    return true;
}
```

B.4 Code for performance characterization

Source file **chamfer.h**

```
#ifndef CHAMFER_H
#define CHAMFER_H

#include <spline.h>

double chamfer_distance (Cubic_Spline const &c1, Cubic_Spline const &c2);

#endif
```

Source file **chamfer.C**

```
#include "chamfer.h"

#include <med-image.h>
#include <mathfix.h>
#include <limits.h>

// The maximum distance that we might need to store when we use
// Borgefors' (PAMI v10 p851) distance transform is 4*max(height,width).
// So if the image is 512x512 then 2^11, or 2048 suffices:

typedef unsigned short dist_image_t [IMAGE_HEIGHT] [IMAGE_WIDTH];


static void distance_transform (dist_image_t &result, Cubic_Spline const &c)
{
    int i, j;
    // Fill distance array with infinity:
    for (i=0; i<IMAGE_HEIGHT; i++)
        for (j=0; j<IMAGE_WIDTH; j++)
            result[i][j] = USHRT_MAX;
```

```
    // Locations intersected by curve c have a distance of zero from it:
    for (i=0; i<c.n(); i++)
        for (double u=0; u<1;)
        {
            Vector2D p = c[i](u);
            int y=nint(p.y), x=nint(p.x);
            result[y][x] = 0;
            double speed = norm (c[i].d_du(u));
            u += 1/speed;  // Move by 1 pixel.
        }

    // Propagate distances forward:
    for (i=1; i<IMAGE_HEIGHT; i++)
        for (j=1; j<IMAGE_WIDTH; j++)
            result[i][j] = min (result[i-1][j-1] + 4,
                                min (result[i-1][j] + 3,
                                     min (result[i-1][j+1] + 4,
                                          min (result[i][j-1] + 3,
                                               result[i][j]))));
    // Propagate distances backward:
    for (i=IMAGE_HEIGHT-2; i>=0; i--)
        for (j=IMAGE_WIDTH-2; j>=0; j--)
            result[i][j] = min (result[i][j],
                                min (result[i][j+1] + 3,
                                     min (result[i+1][j-1] + 4,
                                          min (result[i+1][j] + 3,
                                               result[i+1][j+1] + 4))));
}


double chamfer_distance (Cubic_Spline const &c1, Cubic_Spline const &c2)
{
    dist_image_t dist_from_c1;
    distance_transform (dist_from_c1, c1);

    int n=0;
    double sum=0;
    for (int i=0; i<c2.n(); i++)
        for (double u=0; u<1;)
        {
            Vector2D p = c2[i](u);
            int y=nint(p.y), x=nint(p.x);
            sum += sqr(dist_from_c1[y][x]);
            n ++;
            double speed = norm (c2[i].d_du(u));
            // xxx For efficiency, we should move by more, but:
            u += 1/speed;  // Move by 1 pixel.
        }
    return sqrt(sum/n)/3;
}
```

Source file `snake-utils.h`

```
#ifndef SNAKE_UTILS_H
#define SNAKE_UTILS_H

#include "snake.h"

void copy (Snaxel_Snake const &, Snaxel_Snake &);
double winding_number (Snaxel_Snake const &);
void reverse (Snaxel_Snake const &, Snaxel_Snake &);
Point2D centroid (Snaxel_Snake const &);
void limits (Snaxel_Snake const &, Point2D &top_left, Point2D &bottom_right);

void warp (Snaxel_Snake const &,
           double stretch_x, double stretch_y, double angle,
           Snaxel_Snake &);
```

```
void move (Snaxel_Snake const &,
           Vector2D v, double angle,
           Snaxel_Snake &);

#endif
```

Source file snake-utils.C

```
#include "snake-utils.h"

#include <mathfix.h>


// ---------------------------------------------------------------------------
// Winding number is 1.00 when points go clockwise in an image where X
// increases going right and Y increases going DOWN.

double winding_number (Snaxel_Snake const &snake)
{
    double winding=0;
    for (int i=0; i<snake.n(); i++)
    {
        int j = (i+1) % snake.n();
        int k = (j+1) % snake.n();
        Vector2D v1 = snake.points(j) - snake.points(i);
        Vector2D v2 = snake.points(k) - snake.points(j);
        double cos_t = dot (unit(v1), unit(v2));
        double sin_t = dot (unit(rot90(v1)), unit(v2));
        double t = atan2 (sin_t, cos_t);  // Between -pi and +pi
        winding += t;
    }
    return winding / (2*M_PI);
}


...


// ---------------------------------------------------------------------------
Point2D centroid (Snaxel_Snake const &snake)
{
    static Point2D const o(0,0);
    Vector2D v(0,0);
    for (int i=0; i<snake.n(); i++)
        v += snake.points(i)-o;
    v /= snake.n();
    return v+o;
}


// ---------------------------------------------------------------------------
// Stretch snake along 'angle' and 'angle+90' axes, keeping centroid fixed.
void warp (Snaxel_Snake const &snake,
           double stretch_x, double stretch_y, double angle,
           Snaxel_Snake &new_snake)
{
    if (&snake == &new_snake)
    { // xxx Should use auto_ptr for exception safety:
        Snaxel_Snake *snake_p = snake.another();
        warp (snake, stretch_x, stretch_y, angle, *snake_p);
        copy (*snake_p, new_snake);
        delete snake_p;
        return;
    }

    Point2D c = centroid (snake);
```

```
    Matrix_2D Y (Rotation_Matrix_2D (M_PI/2) *
                         Stretch_Matrix_2D (stretch_y) *
                         Rotation_Matrix_2D (-M_PI/2));
    Matrix_2D transform (Rotation_Matrix_2D (angle) *
                         Y * Stretch_Matrix_2D (stretch_x) *
                         Rotation_Matrix_2D (-angle));
    new_snake.start();
    for (int i=0; i<snake.n(); i++)
        new_snake.add_node (transform * (snake.points(i)-c) + c);
    new_snake.finish();
}


// ---------------------------------------------------------------------------
// Rotate snake by 'angle', keeping centroid fixed, then translate.
void move (Snaxel_Snake const &snake,
           Vector2D v, double angle,
           Snaxel_Snake &new_snake)
{
    if (&snake == &new_snake)
    { // xxx Should use auto_ptr for exception safety:
        Snaxel_Snake *snake_p = snake.another();
        move (snake, v, angle, *snake_p);
        copy (*snake_p, new_snake);
        delete snake_p;
        return;
    }

    Point2D c = centroid (snake);
    Matrix_2D transform ((Rotation_Matrix_2D (angle)));
    new_snake.start();
    for (int i=0; i<snake.n(); i++)
        new_snake.add_node (transform * (snake.points(i)-c) + v + c);
    new_snake.finish();
}
```

Source file `perturb-gen.C`

```
#include <mathfix.h>
#include <string-utils.h>
#include <random.h>


// ---------------------------------------------------------------------------
int main (int argc, char **argv)
{
    if (argc != 2) {
        cout<<"Usage: "<<argv[0]<<" <n>"<<endl;
        return 1;
    }

    int n;
    if (! string_to_int (argv[1], n)  ||  n <= 0)
        {cerr << argv[1] << ": Bad positive integer" << endl;  return 1;}

    // ---
    double sigma_displace = 5;  // pixels
    double sigma_stretch = log(1.1);   // ln(factor)

    double dx=0, dy=0, stretch_x=1, stretch_y=1, stretch_angle=0;
    for (int i=0; i<n; i++)
    {
        cout << dx << " " << dy << "  "
             << stretch_angle << " " << stretch_x << " " << stretch_y
             << endl;

        dx = random_normal (0,sigma_displace);
        dy = random_normal (0,sigma_displace);
```

```
        stretch_x = exp (random_normal (0,sigma_stretch));
        stretch_y = exp (random_normal (0,sigma_stretch));
        stretch_angle = random_uniform (0,M_PI/2);
    }
    return 0;
}
```

Source file `perturb-dist.C`

```
#include <strstream.h>
#include <mathfix.h>
#include <string-utils.h>
#include "snake-utils.h"
#include "chamfer.h"

int main (int argc, char **argv)
{
    Array_Owner<char> err_msg;
    if (argc != 2)
    {
        cerr<<"Usage: "<<argv[0]<<" <snake_file>"<<endl;
        return 1;
    }

    char *input_filename = argv[1];
    Line_Snake snake;
    int slice;
    if (! snake.read (input_filename, slice, err_msg))
        {cerr << err_msg.p() << endl;  return 1;}

    while (1)
    {
        strstream line_s;
        eat (cin, '\n', line_s);
        if (cin.eof()) break;

        double dx, dy,  stretch_angle, stretch_x, stretch_y;
        line_s >> dx >> dy  >>  stretch_angle >> stretch_x >> stretch_y;

        if (! line_s)
            {cerr << "stdin: Not a perturb-gen data line" << endl;  return 1;}
        char c;
        if (! (line_s >> c).eof())
            {cerr<<"stdin: Extra data in perturb-gen line"<<endl; return 1;}

        Line_Snake snake2;
        move (snake, Vector2D (dx,dy), 0, snake2);
        warp (snake2, stretch_x, stretch_y, stretch_angle, snake2);

        double dist1 = chamfer_distance (snake.spline, snake2.spline);
        double dist2 = chamfer_distance (snake2.spline, snake.spline);

        cout << dx << " " << dy << "  "
             << stretch_angle << " " << stretch_x << " " << stretch_y << "  "
             << dist1 << " " << dist2 << endl;
    }
    return 0;
}
```

Source file `perturb-do.C`

```
#include <strstream.h>
#include <libgen.h>
```

```
#include <mathfix.h>
#include <string-utils.h>
#include "snake-utils.h"
char const *program;

int main (int argc, char **argv)
{
    program = basename(argv[0]);
    Array_Owner<char> err_msg;
    if (argc != 3)
    {
        cout<<"Usage: "<<program<<" <snake_file>"<<" <output_file>"<<endl;
        return 1;
    }

    char *input_filename = argv[1];
    char *output_filename = argv[2];

    double dx, dy,  stretch_angle, stretch_x, stretch_y;
    strstream line_s;
    eat (cin, '\n', line_s);
    line_s >> dx >> dy  >>  stretch_angle >> stretch_x >> stretch_y;

    if (! line_s)
        {cerr<<program<<": stdin: Not a perturb data line"<<endl;  return 1;}
    char c;
    if (! (line_s >> c).eof())
        {cerr<<program<<": stdin: Extra data on perturb line"<<endl; return 1;}
    if (! cin.eof()) {
        eat (cin, '\n', line_s);
        if (! cin.eof()) {
            cerr<<program<<": stdin: More than one data line"<<endl;  return 1;
        }
    }
    Line_Snake snake;
    int slice;
    if (! snake.read (input_filename, slice, err_msg))
        {cerr << program << ": " << err_msg.p() << endl;  return 1;}

    move (snake, Vector2D (dx,dy), 0, snake);
    warp (snake, stretch_x, stretch_y, stretch_angle, snake);

    if (! snake.write (output_filename, slice, err_msg))
        {cerr << program << ": " << err_msg.p() << endl;  return 1;}
    return 0;
}
```

Source file `perturb-eval.C`

```
#include <mathfix.h>
#include <strstream.h>
#include <string-utils.h>
#include "all-forces.h"
#include "snake-utils.h"

#if defined(VMS) || defined(__VMS)
const char *gauss_fft_file = "RTP:[Fenster.Data.Gauss-FFTs]";
#else
const char *gauss_fft_file = "/u/boat/fenster/Medical/Data/Gauss-FFTs";
#endif


// ---------------------------------------------------------------------------
int main (int argc, char **argv)
{
    Array_Owner<char> err_msg;
    if (argc != 6) {
        cerr<<"Usage: "<<argv[0]
```

```
        <<" <force>"
        <<" <image_file>"
        <<" <contour_file>"
        <<" <training_file>"
        <<" <scale>"
        <<" < perturb-dist-file  > output-file"
        <<endl;
    return 1;
}

char *force_name = argv[1];
char *image_filename = argv[2];
char *contour_filename = argv[3];
char *training_filename = argv[4];
char *scale_s = argv[5];

Force *force = get_force (force_name);
if (!force) return 1;

double scale;
if (! string_to_double (scale_s, scale)) {
    cerr << scale_s << ": Not a valid scale" << endl;
    return 1;
}

// ---
Line_Snake snake;
int slice;
if (! snake.read (contour_filename, slice, err_msg))
    {cerr << err_msg.p() << endl;  return 1;}
if (slice == -1) {
    cerr<<contour_filename<<": No \"Image section #\" header."<<endl;
    return 1;
}
// Make snake be clockwise:
double winding = winding_number (snake);
if (winding < -0.5)
    reverse (snake, snake);
if (abs (abs(winding)-1) > 0.01)
    cerr << "Warning:  Snake's winding number is " << winding << endl;

// ---
Image image;
image = get_image (image_filename, slice, slice);
if (! image.valid()) return 1;

// ---
double training_scale;
if (! force->training().read (training_filename, training_scale, err_msg))
    {cerr << err_msg.p() << endl;  return 1;}

if (! force->preprocess_image (image [slice], scale,
                              image_filename, slice))
    {cerr << "Error: Couldn't preprocess slice." << endl;  return 1;}

// ---
while (1)
{
    strstream line_s;
    eat (cin, '\n', line_s);
    if (cin.eof()) break;

    double dx, dy,  stretch_angle, stretch_x, stretch_y, dist1, dist2;
    line_s >> dx >> dy  >>  stretch_angle >> stretch_x >> stretch_y
           >> dist1 >> dist2;

    if (! line_s)
        {cerr<<"stdin: Not a perturb-dist data line"<<endl; return 1;}
    char c;
    if (! (line_s >> c).eof())
        {cerr<<"stdin: Extra data in perturb-dist line"<<endl; return 1;}

    Line_Snake snake2;
```

```
        move (snake, Vector2D (dx,dy), 0, snake2);
        warp (snake2, stretch_x, stretch_y, stretch_angle, snake2);
        if (! force->preprocess_snake (snake2.spline))
            {cerr << "Error: Couldn't preprocess snake." << endl;  return 1;}

        double e_b, e_i;
        snake2.set (force);
        if (! snake2.bending_energy (e_b))
            {cerr<<"Aborting"<<endl;  return 1;}
        if (! snake2.image_energy (image[slice], e_i))
            {cerr<<"Aborting"<<endl;  return 1;}

        cout << dx << " " << dy << "  "
             << stretch_angle << " " << stretch_x << " " << stretch_y << "  "
             << e_b << " " << e_i << "  "
             << dist1 << " " << dist2 << endl;
    }
    return 0;
}
```