# Statismo - A framework for PCA based statistical models.

*Release 0.80*

Marcel Lüthi[1], Rémi Blanc[2], Thomas Albrecht[1], Tobias Gass[3], Orcun Goksel[3], Philippe Büchler[4], Michael Kistler[4], Habib Bousleiman[4], Mauricio Reyes[4], Philippe C. Cattin[5], Thomas Vetter[1]

July 24, 2012

[1]Department of Mathematics and Computer Science, University of Basel, Switzerland
[2]IMS Laboratory, University of Bordeaux, France
[3]Computer Vision Lab, ETH Zurich, Switzerland
[4]Institute for Surgical Technology and Biomechanics, University of Bern, Switzerland
[5]Medical Image Analysis Center, University of Basel, Switzerland

**Abstract**

This paper describes the *Statismo* framework, which is a framework for PCA based statistical models. Statistical models are used to describe the variability of an object within a population, learned from a set of training samples. Originally developed to model shapes, statistical models are now increasingly used to model the variation in different kind of data, such as for example images, volumetric meshes or deformation fields.

*Statismo* has been developed with the following main goals in mind: 1) To provide generic tools for learning different kinds of PCA based statistical models, such as shape, appearance or deformations models. 2) To make the exchange of such models easier among different research groups and to improve the reproducibility of the models. 3) To allow for easy integration of new methods for model building into the framework. To achieve the first goal, we have abstracted all the aspects that are specific to a given model and data representation, into a user defined class. This does not only make it possible to use *Statismo* to create different kinds of PCA models, but also allows *Statismo* to be used with any toolkit and data format. To facilitate data exchange, *Statismo* defines a storage format based on HDF5, which includes all the information necessary to use the model, as well as meta-data about the model creation, which helps to make model building reproducible. The last goal is achieved by providing a clear separation between data management, model building and model representation. In addition to the standard method for building PCA models, *Statismo* already includes two recently proposed algorithms for building conditional models, as well as convenience tools for facilitating cross-validation studies.

Although *Statismo* has been designed to be independent of a particular toolkit, special efforts have been made to make it directly useful for VTK and ITK. Besides supporting model building for most data representations used by VTK and ITK, it also provides an ITK transform class, which allows for the integration of *Statismo* with the ITK registration framework. This leverages the efforts from the ITK project to readily access powerful methods for model fitting.

## Contents



## 1 Introduction

Statistical models have become a firmly established tool in computer vision and medical image analysis. They model the variability of a class of objects by means of a normal distribution, learned from example data. In particular in the form of shape models, they have been employed in a wide range of applications such as object recognition [10, 20], image manipulation [8], surgery planning [24, 12] or implant design [9, 15]. Furthermore, they have been used as a prior for different algorithms, such as segmentation [13] or registration [2]. In spite of their success, there is still no established toolkit for the creation and application

of such models. This requires the same method to be implemented over and over again. Moreover, it makes the exchange of models difficult, and thus hinders the collaboration between different researchers as well as the reproducibility of experiments. In this paper we introduce the *Statismo* framework, which is a C++ framework for the creation and use of different kinds of statistical models. In contrast to other libraries (such as the statistical models currently implemented in ITK), *Statismo* provides a high-level interface: Statistical models are interpreted as a probability distribution over the modeled object, from which samples (i.e. shapes, images, etc.) can be drawn and probabilities of object instances can be computed. The current implementation focuses on the most commonly used type of statistical models, i.e. PCA-based shape models, and its associated statistical interpretation as a multivariate normal distribution.

The standard example for a statistical PCA model is the statistical shape model. It represents a class of shapes in terms of deformations of a triangulated reference surface. Nevertheless, the concept can be applied to various other types of data, such as images, displacements fields, volumetric meshes, etc. Independent of the concrete type of model, the statistical analysis requires each dataset to be converted in to a vectorial representation. *Statismo* provides a unified treatment of these models by letting the user provide special template classes, so called *Representer* classes, which define this conversion. Depending on the type of data, this may include preprocessing steps, such as establishing correspondence or alignment of the datasets. A representer also provides the reverse functionality, to convert the vectorial representation back to its original representation. The abstraction introduced by the Representer does not only allow for a unified treatment of different kinds of PCA based models, but also to make *Statismo* independent of the library used to represent the data. Thus, the exact same code that is used for creating a shape model from shapes represented in VTK can also be used for creating deformation models from ITK displacement fields.

One of the main design goals of *Statismo* is to facilitate the exchange of statistical models among different research groups and end users. We therefore defined a platform independent storage based on HDF5 [1]. *Statismo* stores all the information needed to capture the complete probabilistic interpretation of the model. This requires in particular that all the pre-processing steps, such as alignment or normalization can be reproduced. *Statismo* enforces this by requiring the same Representer to be used in the application of the model, as was used for model building. Moreover, *Statismo* stores all the parameters and additional metadata necessary to make the models more easily reproducible. Another goal of *Statismo* is to allow for the easy integration of different model building algorithms. This is achieved by the modular design of the data management, model building and model representation components. Thus, new algorithms for model building can be easily integrated into *Statismo* and make use of all the functionality already implemented. In addition to standard PCA models, *Statismo* currently supports the building of conditional models based on surrogate variables [4, 7], as well as geometric constraints [16].

*Statismo* includes standard Representers for the most commonly used data representations in VTK and ITK, making it possible to build and exchange PCA based shape models, image models, and deformation models, using both ITK and VTK. *Statismo* also provides wrapper classes that allow for the seamless integration of *Statismo* into ITK. Furthermore, a special itkTransform class is provided, which makes it possible to exploit the power of the ITK Registration Framework for statistical model fitting.

## 2 Background: PCA based statistical models

### 2.1 General Introduction

The idea behind PCA Models is to learn a (linear) model for an object class from a set of typical example of this class. Let $O = \{O^1, \ldots, O^n\}$ be a set of training examples. These objects can for example be images,

deformation fields or shapes. Model building consists of the following three steps:

1) Discretization: This aims to define a discrete domain $\Omega = \{p_1, \ldots, p_N\}$ on which we approximate all the objects, i.e. we define an object as:

$$\hat{O}^i = (\varphi^i(p_1), \ldots, \varphi^i(p_N)),\ i = 1 \ldots, n.$$

For shape models, the domain $\Omega \subset \mathbb{R}^3$ is usually defined by the points of a 3D reference mesh and $\varphi^i : \mathbb{R}^3 \to \mathbb{R}^3$ is a function that maps each point $p_j \in \Omega$ to the corresponding point of an example shape $\hat{O}^i$. For images, $\Omega$ is an image domain and $\varphi^i : \Omega \to \mathbb{R}$ that assigns to each point its image intensity. Similarly, for deformation models, $\varphi^i : \Omega \to \mathbb{R}^d$ assigns to each point a deformation that is defined on the deformation field $\hat{O}^i$.

Note that this definition implies correspondence between all the examples. The exact notion of correspondence depends on the type of both the objects and the application. The correspondence between the examples is usually established through registration.

2) Normalization: Depending on the type of the objects, further preprocessing steps may follow. Shapes, for example, are usually rigidly aligned using Procrustes registration before building the model. For images, a histogram equalization or a low pass filtering may be performed to reduce the excess variability of the samples.

3) Model building: Once the shapes are processed, each object $O^i$ can be represented as a high-dimensional vector

$$v^i = (\varphi^i(p_1), \ldots, \varphi^i(p_N)) \in \mathbb{R}^{N \cdot d}.$$

PCA estimates an orthonormal basis spanning this high-dimensional space, defined iteratively as the directions onto which the projection of the data has maximum variance. This offers the opportunity to reduce the dimensionality of the model, keeping only the principal directions covering a prescribed proportion of the total data variability. Mathematically, these are obtained from the sample mean and sample covariance matrix of the training data, using the usual formula from statistics:

$$\begin{aligned} \overline{m} &= \frac{1}{n} \sum_{i=1}^{n} v_i \\ S &= \frac{1}{n-1} \sum_{i=1}^{n} (v_i - \overline{m})(v_i - \overline{m})^T. \end{aligned} \tag{1}$$

Using singular value decomposition, the sample Covariance Matrix is decomposed as $S = UD^2U^T$. The column $u_i$ of the matrix $U \in \mathbb{R}^{Nd \times n}$ is referred to as the $i$-th principal component. The principal components form a basis for the vector space in which all the examples are defined. An entry $\lambda_i$ in the diagonal matrix $D^2 = \text{diag}(\lambda_1, \ldots, \lambda_n)$ denotes how much variance is represented by the $i$-th principal component [18]. A statistical model $v$ is then defined as the low dimensional generative model:

$$v = v(\alpha_1, \ldots, \alpha_m) = \overline{m} + \sum_{i=1}^{m} \alpha_i \lambda_i u_i,$$

where $m < n$ is the number of principal components that span the model space. Each vector $\alpha = (\alpha_1, \ldots, \alpha_m)$ introduces a unique shape. Usually it is assumed that $\alpha$ follows a standard normal distribution:

$$\alpha \sim \mathcal{N}(0, I_m) \tag{2}$$

This in turn introduces a probability distribution over the discretized vectors representing the objects, namely

$$v \sim \mathcal{N}(\overline{m}, UD^2U^T) \approx \mathcal{N}(\overline{m}, S). \tag{3}$$

### 2.2 Probabilistic PCA

In section 2.1 we have sketched the fundamentals of PCA based statistical modeling as it is commonly presented and used in the literature. There is, however, a flaw with this simple probabilistic interpretation (Cf. Equations (2) and (3)). This model assumes that all the probability mass is concentrated on the subspace spanned by the principal components. Thus the probability of an object that does not strictly lie in this subspace is zero. In practice, however, an object might be a valid and likely instance of the class and still not lie exactly on that subspace. This may on one hand happen since this subspace is only approximated from a finite number of examples, but also because the data itself is usually noisy. To alleviate this problem, Probabilistic PCA (PPCA) [23] is implemented in *Statismo*. PPCA provides a well defined probabilistic interpretation, by assuming a small amount of noise on the examples. The new generative model is of the form

$$v = v(\alpha_1, \ldots, \alpha_n) = \overline{m} + \sum_{i=1}^{n} \alpha_i \lambda_i u_i + \varepsilon$$

with $\varepsilon \sim \mathcal{N}(0, \sigma^2)$. By virtue of the added noise term, this model now defines a valid probability distribution on the whole of $\mathbb{R}^{Nd}$. The probability of an instance now depends on the distance from this subspace, i.e. the objects that are far away are less likely than those instances that are closer to the space. It can be shown that when $\sigma^2$ goes to 0 a standard PCA model is obtained [22].

## 3 Design / Architecture

*Statismo* has been designed to achieve the following goals:

**High level:** Most applications using statistical models exploit the fact that these models define a probability distribution over the objects of interest. The interface of *Statismo* is designed to reflect this fact and provides high-level methods to work with the represented distribution, rather than exposing the eigenvalues and eigenvectors of the model.

**Independence of data representation:** Statistical models are used to model the variation in various types of objects, using a variety of different toolkits to represent the data. Independently of the concrete representation of the data, the ideas and interpretation of PCA-based statistical models is always the same. *Statismo* is designed to work for any kind of PCA models, independently of the data representation or the toolkit used.

**Reproducibility and easy data exchange:** It is often fruitful to exchange data and models with colleagues or other researchers. An important design goal is to facilitate this exchange by providing a platform independent data format, which contain all the information needed to use the model from an application. Furthermore, to help reproduce an experiment, the stored model should contain all the parameters that were used to build the model.

**Flexibility and Extensibility:** *Statismo* aims to simplify the integration of new algorithms for model building. This will not only increase the usefulness of *Statismo* itself, but also will allow such algorithms to benefit from the functionalities implemented in *Statismo*.

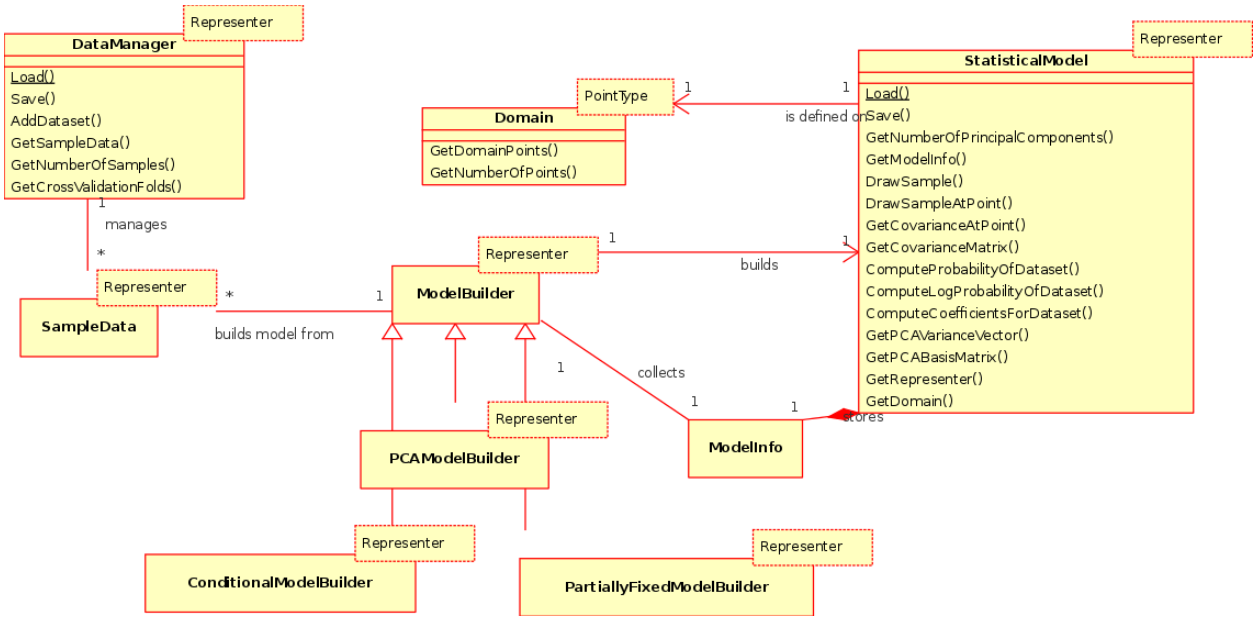


Figure 1: The core classes in *Statismo*

## 3.1 Design overview

To ensure flexibility and extensibility, *Statismo* uses a modular structure that clearly separates data management, model building and model representation. Figure 1 shows a class diagram of *Statismo*. The data used to build the model is added to the data manager `DataManager` class, via the `AddDataset` method. This method first preprocesses (e.g. aligns and normalizes) the dataset and converts it into an internal vectorial representation, which is stored together with metadata, such as the file name of the dataset. These preprocessed datasets are referred to as *samples*. The `ModelBuilder` class takes a list of samples and uses it to build a statistical model. The `ModelBuilder` class is also responsible for collecting all the metadata associated with the samples, which are then stored together with additional information set by the model builder (such as the build time, parameter settings, etc.) in the `ModelInfo` class. The `StatisticalModel` class resulting from the `ModelBuilder` can now be used independently of the other classes. It contains all the information needed in the application of a statistical model and accordingly is saved in a platform independent storage format based on HDF5.

Most terminology used in *Statismo* is directly motivated by the probabilistic interpretation, and does not require further clarification. However, while in statistics the term *sample* and *dataset* are usually used interchangeably, in *Statismo* these terms have precise meanings. A *dataset* is a representation of an object used as an input to build the model. A *sample* represents the same object but after discretization and normalization have been applied to it (Cf. Section 2.1). It is an instance of the object as it is represented by the model. In other words, the probability distribution defined by the statistical model represents the probability of the *samples* but not of the *datasets*. Finally, a *sample vector* is the representation of a *sample* as a vector.

## 3.2 Representers

The key concept that allows *Statismo* to be independent of a specific data representation, and to build all types of PCA models is the concept of a *Representer*. Every class in *Statismo* is parametrized over a

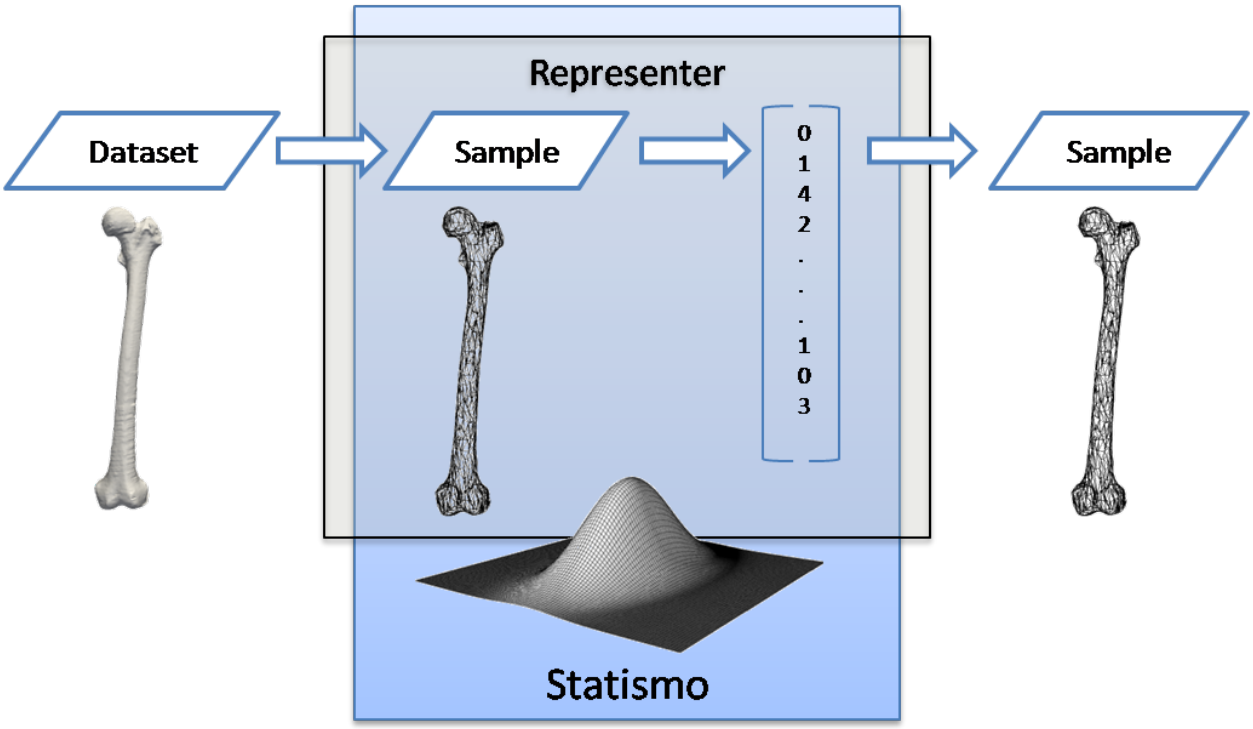


Figure 2: The dataflow in *Statismo*. Inputs to *Statismo* are datasets, in a user defined data representation. The output is always a sample from the probability distribution over the objects that are modeled.

user-defined `Representer` class. A `Representer` class has two distinct purposes: First, it is used as an adapter class to convert between the data representations used by an application and the internal representation used by *Statismo*. This allows *Statismo* to be used with a variety of different toolkits. Second, it also defines how the objects are processed before they are used for building the model. Recall from Section 2.1 that before a model is built, objects are discretized, normalized and converted into a consistent vectorial form. The discretization and normalization step are specific to the type of object that are modeled. A `Representer` class in *Statismo* needs to provide the methods `Representer::DatasetToSample`, `Representer::SampleToSampleVector` and `Representer::SampleVectorToSample`. The first method takes care of the discretization and normalization, the second method is used to convert a dataset to its vectorial representation. The last method is used to convert the vectorial representation back to an object of the modeled type. Figure 2 illustrates this concept. The input to *Statismo* is always a dataset. This is converted into a sample, and then in its vectorial representation, from which all the statistics is computed. Note that the vectorial representation is never exposed to the user, but it is always converted back to the sample representation. For shape models, for instance, the output sample is a mesh, in the specific data representation used by the `Representer`.

## 3.3 ITK Support

*Statismo* was designed to work together with different toolkits and libraries. It was therefore not possible to follow the conventions of a specific toolkit. To facilitate the integration with other libraries, we have tried to make as few assumptions as possible, and limit the interface to use standard C++. Using the generic *Statismo* interface, a code snippet to load a model looks like

```
statismo::StatisticalModel<RepresenterType>* model =
    statismo::StatisticalModel<RepresenterType>::Load("model.h5");
SampleType* sample = model->DrawSample();
delete model;
```

Nevertheless, ITK being a major focus of *Statismo*, we provide special wrappers that follow the ITK conventions and provide a seamless integration of *Statismo* in ITK. Accordingly, the same code snippet will look like

```
itk::StatisticalModel<RepresenterType>::Pointer model =
    itk::StatisticalModel<RepresenterType>::New();
model->Load("model.h5");
SampleType::Pointer sample = model->DrawSample();
```

The main difference between the two interfaces is that the former generic *Statismo* interface requires all arguments as the object is created (here with the Load function) and thus ensures that the objects themselves are well defined after construction. The memory management is done manually whereas, with the ITK Wrapper, the memory is managed using the ITK smart pointers.

Specifically for ITK, *Statismo* also provides a class `itk::StatisticalModelTransform`, which enables the use of statistical models as ITK transforms. This has great practical value since these transform can then be used in the ITK registration framework for statistical model fitting.

## 4 Using *Statismo*

In this section we illustrate how *Statismo* can be used to perform typical tasks related with statistical models:

1. Generation and storage of a statistical model
2. Loading a model from disk
3. Visualizing the variability in a model
4. computing the likelihood of a dataset
5. performing a cross-validation experiment
6. using the ITK registration framework for model fitting.

Most examples below use the generic interface with the `vtkPolyDataRepresenter`. This representer is created by providing it with a reference dataset that defines the discretization and mesh topology. Before converting a dataset to its vectorial representation, it performs a Procrustes alignment [14] of the dataset to the reference, and thus normalizes pose variations automatically. Note that this representer assumes, that the datasets are already in correspondence. The last example illustrates the use of *Statismo* with the ITK registration framework and makes use of the ITK Wrappers for statismo. For this we use the `itkMeshRepresenter` that represents datasets of type `itk::Mesh`.

The illustrations shown in this section have been obtained from a model built from 6 femur bones, which are distributed along with this article.[1]

[1] The example data has been provided by the Virtual Skeleton Database (VSD) project. A larger collection of example data is available at http://www.virtualskeleton.ch

## 4.1 Building statistical models in *Statismo*

The first step in using *Statismo* is to define which representer is used and to parametrize all classes with that representer.

```
typedef vtkPolyDataRepresenter RepresenterType;
typedef PCAModelBuilder<RepresenterType> ModelBuilderType;
typedef StatisticalModel<RepresenterType> StatisticalModelType;
typedef DataManager<RepresenterType> DataManagerType;
```

We create a representer and provide it with a reference. As we are using the `vtkPolyDataRepresenter`, the reference is in this case a `vtkPolyData` object. Furthermore, we load all the datasets that we wish to use in our model.

```
vtkPolyData* reference = loadVTKPolyData("referenceFilename.vtk");
RepresenterType::Pointer representer = RepresenterType::create(reference);

vtkPolyData* dataset_1 = loadVTKPolyData(path_to_polydata_1);
...
vtkPolyData* dataset_n = loadVTKPolyData(path_to_polydata_n);
```

These datasets are then added to the `DataManager`. Note that we also need to provide the path of the dataset as a second argument to the `AddDataset` function. This information will be written as metadata for later reference.

```
DataManagerType* dataManager =  DataManagerType::Create(representer);
dataManager->AddDataset(dataset_1, path_to_polydata_1);
...
dataManager->AddDataset(dataset_n, path_to_polydata_n);
```

In the next step we build a model and save it as a HDF5 file.

```
ModelBuilderType* pcaModelBuilder = ModelBuilderType::Create(representer);
StatisticalModelType* model =
    pcaModelBuilder->BuildNewModel(dataManager->GetSampleData(), 0.01);
model->Save("model.h5");
```

At the end we need to delete the created *Statismo* objects (an alternative would, of course, be to use smart pointers).

```
delete model;
delete pcaModelBuilder;
delete dataManager;
```

## 4.2 Loading a model

In most applications, the first step is not to build a model, but to load an exisiting model. A model is loaded using

```
typedef vtkPolyDataRepresenter RepresenterType;
typedef itk::StatisticalModel<RepresenterType> StatisticalModelType;
StatisticalModelType* model = StatisticalModelType.Load("model.h5");
```

This restores the full model, including the representer. We stress however, that it is important that the same type of representer is used when loading the model, as the one that was used to build the model. *Statismo* will throw an exception if this is not the case.

## 4.3 Visualizing the variability

Probably the first and simplest application that is performed with a new model is to visualize the variability of the modeled objects. This can easily be achieved by drawing random samples and visualizing them using standard tools (such as e.g. Paraview).[2]

```
vtkPolyData* sample = model->DrawSample()
```

We can also explore the shape space more systematically by specifying the PCA coefficients for a given sample. In this example, we obtain three samples: The mean, a sample corresponding to 3 standard-deviation in the direction of the first principal component and one corresonding to 3 standard-deviation in the opposite direction.

```
vtkPolyData* mean = model->DrawMean()
VectorType coeffs = VectorType::Zeros(model->GetNumberOfPrincipalComponents());
coeffs[0] = 3;
vtkPolyData* sample1PC1 = model->DrawSample(coeffs);
coeffs[0] = -3;
vtkPolyData* sample2PC1 = model->DrawSample(coeffs);
```

Note that each call to DrawSample create a new sample which needs to be deleted by the user.

## 4.4 Using statistical models as a prior

In many algorithms, statistical models are used as a shape prior. A common approach is to take an instance from an algorithm, and project it back into the shape space, to get an approximation of the given shape. In *Statismo*, such a projection can be obtained by computing the PCA coefficients for a given dataset and then restoring it using the `DrawSample` method:

```
vtkPolyData* dataset = someAlgorithm();
VectorType coeffs = model->ComputeCoefficientsForDataset(dataset);
vtkPolyData* projection = model->DrawSample(coeffs);
```

Often, an algorithm is regularized by penalizing unlikely solutions. For this purpose, *Statismo* provides a method to compute a (log) probability of a given dataset.

```
float logProbability = model->ComputeLogProbabilityForDataset(aDataset);
```

[2] Note that the samples are drawn from the multivariate normal distribution that the model represents.

$-3\sigma$ mean $+3\sigma$

Figure 3: The first mode of variation ($\pm 3$ standard deviations) for a shape model of the femur. In this case the first variation mainly captures the size of the bone.

## 4.5 Cross validation

An often neglected, but important task is the evaluation of statistical models. One particular focus of interest is the generalization ability of a model: the accuracy with which it is able to represent new instances of the object. This ability may be estimated through cross-validation studies. Cross-validation has also recently been employed to derive confidence bounds quantifying the reliability of statistical model based prediction from partial observations [5, 6]. *Statismo* provides convenience functionalities in order to facilitate such cross-validation studies.

Cross-validation is done using the `DataManager`. After we have set up the `DataManager`, and populated it with the training data, we can obtain a list with cross validation folds.

```
typedef DataManagerType::CrossValidationFoldListType CVFoldListType;
CVFoldListType cvFoldList = dataManager->GetCrossValidationFolds(4, true);
```

Each entry of the `cvFoldList` contains a list of training samples and a list with the test samples. We iterate over all the folds and build for each fold a new model. A typical validation of the model would consist of computing an approximation error between the test sample, and its projection in the model.

```
for (CVFoldListType::const_iterator it = cvFoldList.begin();
     it != cvFoldList.end();
     ++it)
{
```

```
    ModelBuilderType* builder = ModelBuilderType::Create();
    StatisticalModelType* model(builder->BuildNewModel(it->GetTrainingData(), 0.01));

    const SampleDataListType testSamplesList = it->GetTestingData();
    for (SampleDataListType::const_iterator it = testSamplesList.begin();
         it != testSamplesList.end()
         ++it)
    {
        vtkPolyData* testSample = (*it)->GetAsNewSample();
        vtkPolyData* projection = model->DrawSample(
                                        model->ComputeCoefficientsForDataset(dataset));
        /// here we could for example compute the approximation error between the
        /// projection and the test sample - omitted

        testSample->Delete();
    }
    delete modelBuilder;
}
```

## 4.6 Model Fitting using the itk registration framework

One of the most common applications of statistical models is to fit them to a patient dataset. In *Statismo* this is done using the ITK registration framework. In the example we show how a shape model is fitted to an image using `itkPointSetToImageMetric`. In this example, we use the ITK wrapper of *Statismo*, which allows us to use *Statismo* objects as normal ITK objects.

We start with the usual type definitions. We use for this example a shape model, which has been built using the `itkMeshRepresenter`.

```
typedef itk::Mesh<float, 3> MeshType;
typedef itk::ImageType<float, 3> ImageType
typedef itk::MeshRepresenter<float, 3> RepresenterType;
typedef itk::StatisticalShapeModelTransform<RepresenterType, double, 3> TransformType;
typedef itk::MeanSquaresPointSetToImageMetric<MeshType, ImageType> MetricType;
```

The StatisticalShapeModelTransform is a class defined by *Statismo*. It is a subclass of the `itk::Transform` and can thus be directly used wherever an `itk::Transform` is expected.

In the next step we load the model and obtain the reference mesh that was used to build the model via the representer. This mesh is used by the `PointSetToImageMetric`, and determines the points on which the metric is evaluated.

```
StatisticalModelType::Pointer model = StatisticalModelType::New();
model->Load(modelname);
MeshType::Pointer fixedPointSet  = model->GetRepresenter()->GetReference();
```

The transform is set up by setting the statistical model, and initializing it to the identity transform.

```
TransformType::Pointer transform = TransformType::New();
transform->SetStatisticalModel(model);
transform->SetIdentity();
```

The registration is then set up in the usual way. We omit here the details.

```
MetricType::Pointer metric = MetricType::New()
...

RegistrationFilterType::Pointer registration = RegistrationFilterType::New();
registration->SetInitialTransformParameters(transform->GetParameters());
registration->SetTransform(   transform );
registration->SetFixedPointSet( fixedPointSet );
...
registration->Update()
```

After a successful registration, the transform parameters correspond to the PCA coefficients in our model. To obtain the final mesh, we can obtain these coefficients from the transform using the method `GetCoefficients` and draw the corresponding sample from the model.

```
MeshType::Pointer finalMesh = model->DrawSample(transform->GetCoefficients());
```

## 5 Adding more prior information: Constrained PCA models

One goal of *Statismo* is to allow the integration of different algorithms for building statistical models. Currently, *Statismo* comes with three different methods to build statistical models (Cf. Figure 1). Besides the standard `PCAModelBuidler` that was used in all the examples presented so far, there is a `PartiallyFixedModelBuilder` [16] and a `ConditionalModelBuilder` [4]. Both model builders incorporate additional prior information into the model building and thus reduce the variability of the resulting model. The resulting models are standard PCA models themselves, and can be used and interpreted in exactly the same way as the models discussed so far.

### 5.1 Partially fixed models

Partially fixed models allow for introducing geometric constraints on the model. This allows us to model the variability in a model, when we know the shape for a part of the model. A typical application of this model is the reconstruction of a fractured or traumatized bone. Another scenario where these models are useful is in model fitting. Consider the shape model fitting example from Section 4.6. Assume that we know for some points of the model the corresponding points on the target image. Using the `PartiallyFixedModelBuilder`, we can incorporate these constraints into the model building, to obtain a model that represents only shapes that match the target points. The following (incomplete) code illustrates how such models can be built.

```
// ... usual type definitions - omitted
typedef PartiallyFixedModelBuilder<RepresenterType> PartiallyFixedModelBuilderType;
```

$-3\sigma$ mean $+3\sigma$

Figure 4: A shape model of the femur, with two landmark points (in red) fixed. As the landmark points determine the size of the femur the first component is not the size anymore (as in 3), but shows mainly the variations of the femoral head and the condyles.

```
PartiallyFixedModelBuilderType*  pfmb = PartiallyFixedModelBuilderType::Create();

// ... Read model and target landmarks, omitted

StatisticalModelType::PointValueListType constraints;
for (unsigned i = 0; i < numLandmarks; i++) {
   StatisticalModelType::PointValuePairType cnstr(modelLandmarks[i], targetLandmarks[i]);
   constraints.push_back(cnstr);
}
StatisticalModelType* constraintModel =
   pfmb->BuildNewModel(dataManager->GetSampleData(), constraints);
```

Figure 4 shows the first main variation for the same shape model as in Figure 3, but this time with 2 points (in red) fixed. We see that the landmark points remain fixed in the first main variation. By using this constrained model for model fitting, we can thus guarantee that the landmark points are matched, without having to change our fitting algorithm. An additional advantage is that the search space becomes smaller, and thus the optimization problem easier. For more details on the theoretical background and the applications of these models, we refer to the papers [16, 17].

### 5.2 Conditional Models

The second algorithm for model building, which we refer to as the *ConditionalModelBuilder* for constrained model is used when there are additional surrogate information about the data available. The underlying idea is to exploit such information to generate a statistical model of the object variability given prescribed constraints. *Statismo* handles both continuous and categorical types of surrogates (e.g. the height, age and gender of the patient). To achieve this, the potential useful surrogate variables have to be associated to the training samples.

A special `DataManagerWithSurrogates` is provided for this purpose. When creating such a data manager, a file describing the number and types of variables is passed as an argument. The class has a special method to add datasets together with the associated surrogate data: `AddDatasetWithSurrogates(dataset, surrogateFilename)` (the exact format of these input files is described in the internal documentation).

In order to learn a conditional model, the user then specifies which of surrogate variables are used for conditioning, and specifies the corresponding values against which the model should be conditioned by filling a structure of type `ConditionalModelBuilder<RepresenterType>::CondVariableValueVectorType`.

In this implementation, only the training samples which respect the requested categories are selected. Continuous variables are used to learn a conditional distribution of the variability, using the assumption of joint multivariate normality. Practically speaking, an unconditional statistical model is first learned using the PCAModelBuilder class using the selected training sample. The PCA coefficients of each of these training samples are computed, and pooled with the requested continuous surrogates. The conditional distribution of the model coefficient given the prescribed constraints is derived. The assumption of multivariate normality then allows to reformulate this conditional model in the reduced space back into the original high dimensional space. Therefore, the obtained conditional model is expressed in the exact same mathematical formulation as standard PCA models. The implementation is currently limited to cases where the number of retained training samples (after selection based on categories) is larger than the number of original PCA coefficients retained plus the number of continuous surrogate variables used as conditions.

More details, and possible extension to more complex distributions models can be found in [4].

## 6 Conclusion

In this paper we have presented *Statismo*, which is a framework for PCA based statistical models. *Statismo* implements a high level view on PCA models by interpreting these models as a normal distribution over the modeled objects. The underlying low level vectorial representation of an object is never exposed to the user. The conversion between the data representation of the user and the vectorial representation used by *Statismo* is done via special Representer classes, which are defined for each data representation. This allows *Statismo* to ensure that the same discretization and normalization steps are applied when using the model, as those that were used during the model building. A statistical model in *Statismo* thus always has a well defined interpretation and semantics. This, together with the fact that *Statismo* writes all its information into a single, portable HDF5 file, greatly simplifies the exchange of statistical models built with *Statismo*. To use the model, the user only needs to have access to the same Representer that was used to build the model.

*Statismo* comes with ready made Representer classes for the most important object representations used in VTK and ITK. Currently all these representers require that the data is already in correspondence. It is, however, straight-forward to write representers that also establish correspondence, and thus to cover the full model creation pipeline with *Statismo*. By writing custom representers, it becomes also possible to

use *Statismo* with other toolkits than ITK or VTK. *Statismo* also simplifies the integration of new methods for building models. As it is a powerful toolkit for building statistical models, we hope that the number of different representers and model builders for *Statismo* will grow quickly, enabling its use in several applications across various fields.

# A Appendix

## A.1 The file format

*Statismo* uses the platform independent HDF5 [1] to save the statistical models. The HDF5 file contains all the information that is needed to use the model from an application as well as information about the data and parameters used to create the model. HDF5 provides a hierarchical, file-system like data organization. For *Statismo*, we have organized the data into 3 main groups:

**Model** Contains the vectors and matrices that define the model, i.e. the `mean`, the `pcaBasis`, the `pcaVariance` as well as the `noiseVariance` assumed in the model.

**Modelinfo** Holds the metadata for the model. The metadata consists of the `buildtime` of the model, the `scores` (the PCA coefficients of the original example data), a section using `builderInfo` that contains all the parameters of the specific model builder used to build the model, as well as a section `dataInfo` that contains the URI's of the datasets that have been used to build the model.

**Representer** This group is used to store all the information that is needed by the representer.

The data stored in the HDF5 file can eihter be explored using the API provided by *Statismo* (Cf. the class `StatisticalModel`), or directly from one of the many software packages that support HDF5, such as e.g. Matlab [19], R [21] or Pytables [3].

## A.2 Currently available Representers

*Statismo* comes with several Representer classes for building various types of statistical models, using VTK and ITK. Currently, all the representers requires that the input datasets are discretized in the same way and in correspondence.

In the following we briefly describe the different Representers:

**TrivialVectorialRepresenter** This is the most basic representers. It can be used to build statistical models in the case where the input is already available in a vector representation.

**vtkPolyDataRepresenters** Used to build shape models from VTK, in case where the data is represented as VTK Polydata. This representer performs a Procrustes alignment [14] of the datasets before the model is built.

**vtkStructuredPointsRepresenter** This representer is used to build image (intensity) models and deformation models from data that is represented as vtkStructuredPoints.

**itkMeshRepresenter** This representer is used to build shape models from itk Meshes. In contrast to the vtkPolyDataRepresenter, no alignment is performed.

**itkImageRepresenter** Used to build 2D and 3D image (intensity) models from itk images.

**itkVectorImageRepresenter** Used to build 2D and 3D deformation models from itk displacement fields.

**itkVectorImageLMAlignRepresenter** This representer is similar to the itkVectorImageRepresenter. In addition, the user can provide a number of landmark points. the input displacement fields are evaluated at these landmark points, to obtain a set of corresponding points. From these points, a rigid transformation is computed, which is used to resample the displacement fields.

### A.3 Third Party libraries used in *Statismo*

*Statismo* depends on a number of open source libraries: *HDF5* [1] is used to store the statistical models. For linear algebra we use the *Eigen* template library [11]. Furthermore, *Boost.tr1* was used to make TR1 available on all platforms. All the libraries are distributed along with *Statismo*. Detailed information regarding the license terms of the different libraries can be found in the source tree.

### Acknowledgements

This work has been supported by the CO-ME/NCCR research network of the Swiss National Science Foundation. The femur data has been made available by the Virtual Skeleton Database (VSD) project[3]. We also would like to thank Arnaud Gelas for helpful feedback regarding this software package.

## References

[1] The hdf group. hierarchical data format version 5, 2000-2010. http://www.hdfgroup.org/HDF5. 1, A.1, A.3

[2] T. Albrecht, M. Lüthi, and T. Vetter. A statistical deformation prior for non-rigid image and shape registration. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, page 18, 2008. 1

[3] Francesc Alted, Ivan Vilata, et al. PyTables: Hierarchical datasets in Python, 2002–. A.1

[4] R. Blanc, M. Reyes, C. Seiler, and G. Székely. Conditional variability of statistical shape models based on surrogate variables. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2009. 1, 5, 5.2

[5] R. Blanc, E. Syrkina, and G. Székely. Estimating the confidence of statistical model based shape prediction. In *Information Processing in Medical Imaging*, page 602613, 2009. 4.5

[6] R Blanc and G Székely. Confidence regions for statistical model based shape prediction from sparse observations. *IEEE Transactions on Medical Imaging*, February 2012. PMID: 22374354. 4.5

[7] Rmi Blanc, Christof Seiler, Gabor Szkely, Lutz-Peter Nolte, and Mauricio Reyes. Statistical model based shape prediction from a combination of direct observations and various surrogates: Application to orthopaedic research. *Medical Image Analysis*, (0), 2012. 1

[3] www.virtualskeleton.ch

[8] Volker Blanz and Thomas Vetter. A morphable model for the synthesis of 3D faces. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, page 187194. ACM Press, 1999. 1

[9] H. Bou-Sleiman, L. Ritacco, L.P. Nolte, and M. Reyes. Minimization of intra-operative shaping of orthopaedic fixation plates: a population-based design. *Medical Image Computing and Computer-Assisted InterventionMICCAI 2011*, page 409416, 2011. 1

[10] T. F Cootes, A. Hill, C. J Taylor, and J. Haslam. Use of active shape models for locating structures in medical images. *Image and vision computing*, 12(6):355365, 1994. 1

[11] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. A.3

[12] DJ Hawkes, D. Barratt, JM Blackall, C. Chan, PJ Edwards, K. Rhode, GP Penney, J. McClelland, and DLG Hill. Tissue deformation and shape models in image-guided interventions: a discussion paper. *Medical Image Analysis*, 9(2):163175, 2005. 1

[13] T. Heimann and H. P Meinzer. Statistical shape models for 3D medical image segmentation: A review. *Medical Image Analysis*, 2009. 1

[14] B. K.P Horn. Closed-form solution of absolute orientation using unit quaternions. *JOSA A*, 4(4):629642, 1987. 4, A.2

[15] N. Kozic. *Statistical Shape Space Analysis Based on Level Sets for Optimization of Orthopaedic Implant Design*. PhD thesis, 2009. 1

[16] M. Lüthi, T. Albrecht, and T. Vetter. Probabilistic modeling and visualization of the flexibility in morphable models. In *Proceedings of the 13th IMA International Conference on Mathematics of Surfaces XIII*, page 264, 2009. 1, 5, 5.1

[17] M. Lüthi, C. Jud, and T. Vetter. Using landmarks as a deformation prior for hybrid image registration. *Pattern Recognition*, page 196205, 2011. 5.1

[18] Kanti V. Mardia, J. T. Kent, and J. M. Bibby. *Multivariate Analysis*. Academic Press, 1 edition, February 1980. 2.1

[19] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010. A.1

[20] P. Paysan, R. Knothe, B. Amberg, S. Romdhani, and T. Vetter. A 3D face model for pose and illumination invariant face recognition. In *Advanced Video and Signal Based Surveillance, 2009. AVSS'09. Sixth IEEE International Conference on*, page 296301, 2009. 1

[21] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0. A.1

[22] S. Roweis. EM algorithms for PCA and SPCA. *NIPS*, page 626632, 1998. 2.2

[23] Michael E. Tipping and Christopher M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society*, 61:611–622, September 1999. 2.2

[24] Guoyan Zheng. *Statistical finite element modeling: application to orthopedic implant design*. Habilitation, University of Berne, 2011. 1