4-2012

Automatic Atlas Based Analysis of Radiotracer Uptake in Bones from Fused Nuclear Imaging/CT Data Sets of Mice

Jeffrey Lee VanOss
Grand Valley State University

Automatic atlas based analysis of radiotracer uptake in bones from fused nuclear imaging/CT data sets of mice

Jeffrey Lee VanOss

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Masters of Science in Engineering

Padnos College of Engineering & Computing

April 2012

Acknowledgments

Dr. Anthony Chang:	For supporting me giving me the wonderful opportunity to work in his small animal imaging lab
Dr. Samhita Rhodes	For her invaluable advice
Anderson Peck	For numerous helpful discussion sessions
My family and friends	For sticking with me regardless of how many times I told them "I can't hang out, I have to work on my thesis instead"

This project is funded under the National Science Foundation American Recovery and Reinvestment Act of 2009 (ARRA) (Public Law 111-5).

Abstract

Preclinical in vivo imaging is a powerful tool used for a wide variety of biomedical research applications including oncology, cardiac disease, and neurological disease. Disease physiology can be imaged in vivo with molecular imaging such as PET and SPECT. Quality analysis of molecular in vivo images currently requires an expert technician. The feasibility of large preclinical molecular imaging studies is limited by the man hours required to process the overwhelming amount of data created from preclinical scans.

Our proposed solution to the bottle neck of manual image analysis is to implement automation of preclinical molecular image analysis. The method described in this study automatically registers different bone regions of interest in fused molecular imaging/CT scans. Automated analysis can run without supervision from a user, allowing for an increase in image processing throughput compared to manual analysis.

The results of this novel image analysis show that atlas based registration of CT data is possible with a moderate degree of accuracy. Using this registration method to generate radiotracer uptake values for different bone groups resulted in mixed success. Bones that are registered first; skull, spine, pelvis, had automated radiotracer uptake measurements that correlated highly with the manual radiotracer uptake measurements. Bones that were last to be registered; tibia, hindpaws, were susceptible to large amounts of variation from the manual radiotracer uptake measurements. Large improvements to the accuracy of the results could be made by ensuring the accuracy of the joint registration of the atlas to the CT dataset.

Table of Contents

1. Introduction

Traditional small animal cancer model experiments require a large starting cohort of animals. At several points during the study, portions of the cohort are euthanized and dissected in order to generate data for the time point. A large number of animals was required to compensate for the inter subject variability. Preclinical small animal imaging allows anatomical structures and physiological functions to be imaged in vivo. This allows for different time points to be captured from the same animal throughout a longitudinal study. Acquiring multiple time points from the same animal and effectively using each animal as its own control allows for a dramatic decrease in the amount of animal models needed in the starting cohort. For this reason *in vivo* small animal imaging has become increasingly prevalent in preclinical research.

Datasets generated from imaging systems require expert analysis in order to generate quantifiable, relevant data. There are many software packages available to aid researchers with image analysis. These programs range from general purpose to specific applications and from manual segmentation to fully automated segmentation. General purpose applications tend towards manual segmentation while applications that target a specific situation are more automated. The processing of images is the bottleneck in obtaining useful data from preclinical investigations. In order to improve our ability to draw conclusive inferences from imaging studies, we need to develop algorithms that will allow the automated analysis of images.

1.1. Imaging Modalities

Several different imaging modalities are in commonly used in preclinical imaging. These include x-ray computed tomography (CT), which excels at imaging hard tissues like bone; magnetic resonance imaging (MRI), which is capable of imaging soft tissue; single photon emission computed tomography (SPECT) and positron emission tomography (PET), which use radioisotopes to image physiologic functions, and optical imaging which can image fluorescence and bioluminescence. This thesis focuses on images obtained via CT, SPECT, and PET modalities although the applications of the methods detailed within could be extended to other modalities.

1.2. Computed Tomography

X-ray computed tomography, sometimes referred to as computed tomography or computed axial tomography (CAT), obtains multiple 2D x-ray projection images at different angles around the subject. These projections can then be input into a reconstruction algorithm, commonly the filtered back projection algorithm, to reconstruct 3D volumetric data of the subject. CT images are grayscale and are a measurement of x-ray attenuation (Hounsfield units), which is analogous to the density of the material the x-ray passed through.. Improved CT images are typically acquired by adjusting the x-ray energy, increasing the exposure time, and increasing total number of projections. Because CT scans expose the subject to x-rays, the dose to the subject must be considered when trying to maximize image quality. Since CTs use x-rays to generate the final image, they have similar image quality limitations as standard x-ray projections, namely that there is large contrast between air, soft tissue, and bone, but boundaries between soft tissues

are often impossible to distinguish. This has led to CT being used to detect bone or lung lesions, but not for soft tissue lesions.

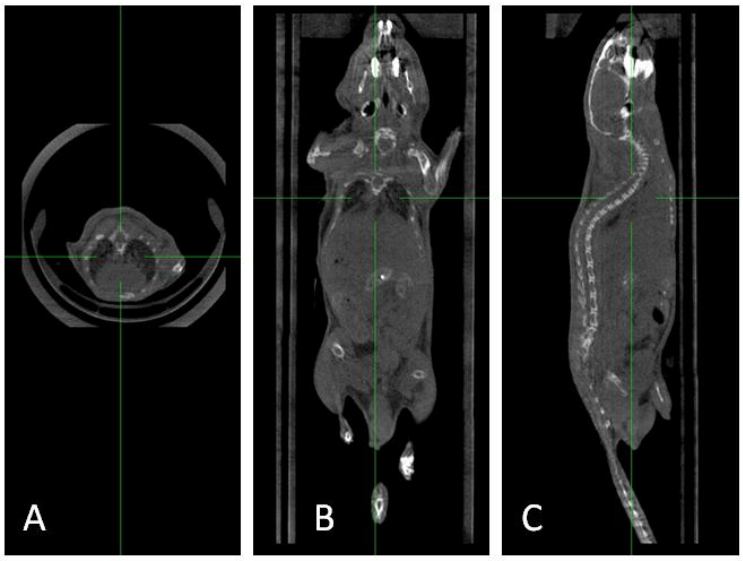

Figure 1: Axial (A), Coronal (B), and Sagittal (C) views of a CT scan. Notice the high bone contrast, but poor contrast between soft tissues

1.3. Molecular Imaging

Molecular imaging (MI) is a subset of medical imaging. The goal of molecular imaging is to view physiological events by tracking specific molecules. MI usually consists of two parts. The first part is a tracer, sometimes referred to as probe, which is the molecule to be tracked throughout the body. A tracer could be a protein, antibody, or other biological substance. Most tracers are not inherently visible to any imaging modality, thus the second part of MI is the method by which the tracer is tracked. PET and SPECT are MI modalities but are sometimes called Nuclear Imaging (NI) or Nuclear Medicine (NM) modalities because they use radionuclide. For these modalities a radionuclide is attached to the tracer. When the radionuclide decays, its location can be detected by the scanner.

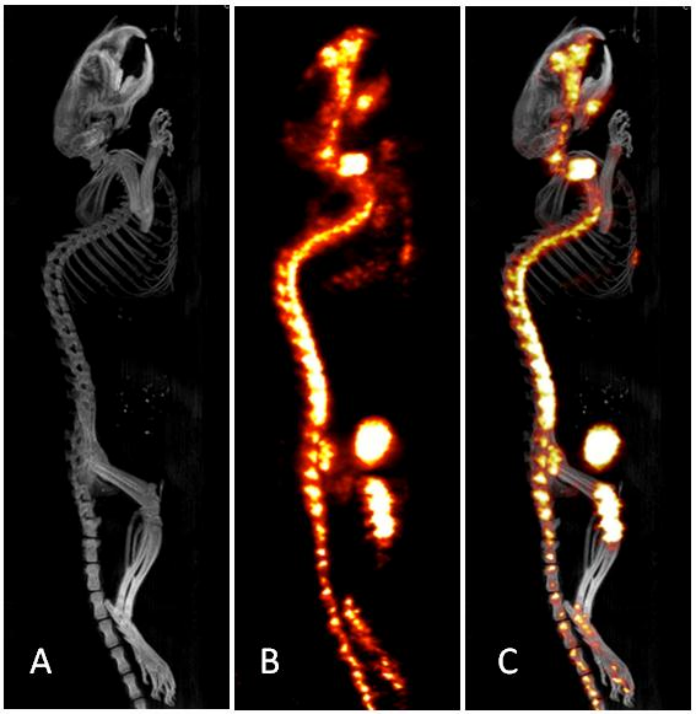

Figure 2: CT image (A) shows anatomical information. SPECT image (B) of radiotracer 99mTc-MDP shows regions of osteoblastic activity in bones. Functional images such as this one might not provide much information about the location of the signal. Often CT and MI images are captured together so that they can be overlaid, or fused, to give additional anatomical information to the MI data.

SPECT imagers measure electrons created from β^- nuclear decay. When these radionuclides attached to tracer molecules undergo β^- decay, they releasing photons at different energies. The energy level of these photons can be detected by gamma cameras on the SPECT which record the event. This allows for multiple isotopes, and therefore multiple physiologic events to be imaged at once. SPECT scanners use collimators to limit the incident angle of incoming photons. This results in a single 2D view. The SPECT heads rotate around the subject to take multiple 2D images at different angles. These images can be reconstructed to create 3D volumetric data much the same way 2D projections from a CT scan are reconstructed. SPECT scans require radiation and so subject dose is always a consideration. In order to reduce dose, only a minimum level of radioactive isotope is injected, resulting in a low resolution image.

PET creates images from positron (β^+) nuclear decay. A positron will travel a short distance in the object before it collides with an electron. This results in mutual annihilation of the two particles. As per Einstein's equation $E = mc^2$, two photons are created each with an energy level of 511keV. These photons travel in opposite directions. PET scanners take advantage of this and record when two photon detections must have come from the same event. This is called coincidence detection. Because of coincidence detection, PET scanners do not need collimators in order to determine the location of the annihilation event. This means that no signals are intentionally blocked and the radiation activity can be much smaller, typically by an order of magnitude, than SPECT. While the radiation activity is smaller, the energy levels of the PET radiotracers are higher than SPECT. Resolution suffers in PET images because, in addition to the reduced radiation activity, PET scans record the annihilation event location, not the β+ decay.

CT is conventionally used to image anatomy but not functional physiology. The benefit of MI is that diseases can often be detected earlier by looking for their molecular characteristics as opposed to waiting for anatomical or obvious physiological symptoms. However, a drawback of MI can be the lack of specificity in probes.

An example might help clarify. Fludeoxyglucose (FDG) is a PET radiotracer. Glucose is the tracer, the molecule we are interested in following. 18-Fluroide is the radionuclide that reports the location of the tracer when it decays. Cells uptake FDG as though it were normal glucose, thus areas of the body with more FDG correlate to areas that more metabolically active. FDG is often used to detect cancerous lesions because an elevated metabolic rate is a hallmark of malignant tissue. However other parts of the body have naturally elevated metabolic rates. This low specificity for lesions lowers the signal to noise ratio (SNR). A low SNR can hide lesions and make a radiologist job of reading medical images more difficult.

2. Related Work

One approach to automating medical image analysis has been the use of biological atlases. A biological atlas is an image of previously segmented biological structures. This data can be applied to new data as a reference to aid segmentation. 3D atlases can be used to segment volumetric data obtained from medical imaging modalities like CT, SPECT, and PET. These atlases can be subdivided into whole body or targeted atlases, and clinical or preclinical atlases. Whole body atlases try to be as all inclusive as possible and segment most, if not all, organs. Targeted atlases focus on a particular organ system or a small group of constituent anatomical structures. A very popular region for targeted atlases is the brain[1]. Brain atlases are used heavily in neurological research.

There are several commonly used atlases for clinical research. The Talairach brain atlas[2] is a 3D map of regions in the human brain. This atlas is used for basic medical image segmentation as well as brain mapping and neurosurgery.

The Visible Human Project[3] consists of whole body atlases of a male and female. MRI, CT, and cryosections of two bodies we used in conjunction to create the Visible Human atlases. The detail available in these atlases is extreme. Fine details such as nerves are segmented. The level of detail makes the atlases widely applicable and they are currently being used for education, industrial use, and virtual reality. Google and Zygote provide a web based tool, which allows users to navigate the visible man/woman atlases.

The 4D NCAT torso phantom[4] is another common clinical atlas. The NCAT atlas strength lies in its ability to simulate different human positions and their effects on organ movement. This model is often used to simulate human-device interactions. Reviews of atlases[5–7] and body models have

been done, but they usually focus on human atlases and on the computational modeling aspects of the atlases, not the image segmentation capabilities.

Preclinical atlases for a wide variety of animal models exist. However, because of the relative simplicity of creating preclinical animal model atlases in comparison to human atlases, many preclinical atlases are proprietary. Of the publicly available atlases, many are targeted atlases[1]. There are three commonly cited, publicly available, whole body small animal atlases, two of mice, and one of a rat. The three atlases are Digimouse[8], Moby mouse[9], and SD rat[10]. These three atlases have been segmented, developed into articulated atlases, and made available for public download[11]. The Moby mouse atlas was developed at John Hopkins University form a male C57BL/6 mouse, a common black lab mouse. The mouse was imaged using high resolution MRI and the subsequent dataset was segmented manually. Almost all of the major organs were segmented. In addition, data was captured in 4D allowing functions like breathing and respiratory rate to be simulated using the Moby mouse atlas. The Digimouse atlas was constructed from CT, PET, and cryosections of a male nude mouse model. The Digimouse atlas was also segmented by hand. Most major organs were segmented, however, suborgan regions were not segmented as they were in the Moby mouse. The SD Rat atlas was created from cryosectioning of a male Sprague-Dawley rat. Segmentation was performed manually and most major organs were segmented.

Atlases by themselves are not sufficient to segment a dataset. The Moby mouse was generated from a single MRI scan; in order for this atlas to be used for segmentation directly, a new scan must be of a mouse of the exact size and proportions in the exact position as the MOBY mouse was during its scan. In order to make an atlas applicable to a new dataset a transformation must be applied to the atlas to line it up with the mouse in the new scan. The complexity of a

transform is determined by the Degrees of Freedom (DoF) that it describes. Euclidean transforms describe translation and rotation for a total of 6 DoF in three dimensional space. Allowing scaling increased the complexity to 9 DoF. Complexity can be increased to non-linear transforms that allow for N DoF.

The scope of the transformation can be described as global or local. Global transformations apply to the entire atlas in order to align it to a dataset and usually require more DoF to describe the transformation. Local transforms use several intermediate transforms, each applied to a subset of the atlas, to align the whole atlas to a dataset. Transforming an atlas with local transformation requires the determination of several intermediate transforms but each requires fewer DoF than a single global transform.

Le[12] et al describe a global transform function to align a CT dataset to an atlas using a three step process. First, the atlas and the CT image are roughly aligned using mutual information. Secondly, points on the atlas are non-rigidly aligned to points on the CT image dataset. Finally, the atlas and the CT dataset are intensity based non-rigidly registered. This method was shown to work on both mice and human datasets. It can be used to segment finer features like ribs that other registration methods have difficulty with. The negative aspect of this method is that it is non-rigid; this means that rigid objects like bones are usually deformed in a manner that is not physically possible. Additionally, this method cannot handle large variation of limb positions.

Chaudhari presents a method[13] for aligning an atlas to Optical Bioluminescence Tomography (OBT). OBT is generated from multiple standard photographs at different angles around a subject. From these pictures, a surface reconstruction can be created. Chaudhari uses harmonic mapping[14] to assign the skin of the atlas to the skin of the OBT. The same global transformation

that is used to align the skin can also be used to align other organs and bones in the atlas. This method is non-rigid so unrealistic deformation of organs and bones can take place.

Kovacevic employed a technique[15] that used a 'part-of' hierarchy method to align local regions. Affine transforms were used to align a section of the atlas to the MRI dataset. This section was then subdivided into multiple subsections which were each again aligned with affine transforms. This process was repeated until a desired level of subdivision was reached. This process yielded good results; however, because subdivisions are aligned without reference to other subdivisions, large postural changes caused problems for image registration.

A study by Martín-Fernández[16] centers on aligning an atlas to radiographs of a human hand. This paper presents the novel idea of constraining the atlas transformation by anatomical information. However, this approach was used for a clinical targeted atlas, not a preclinical whole body atlas, which is the focus of this study.

Papademetris used accurate anatomical modeling to register an articulated atlas of the hind limbs of a mouse to a CT scan[17]. This process requires pre-segmentation of articulated regions. After this preprocessing, the articulated regions can be modeled and moved around in an anatomically correct fashion and these rigid transforms can be used to affect the soft tissue registration. This approach proves the capabilities of anatomic modeling of articulated atlases in small animal imaging, but it is only applied to a subsection of the body.

Baiker and colleagues have published a series of papers documenting the development of an articulated whole body mouse atlas. The first paper presented whole body registration of an articulated skeletal atlas to CT data[18]. Further development of the software led to the registration of some soft tissue organs based on data derived from the registration of the skeleton[19,20]. This

atlas registration software has a number of features. The atlas is based on the publicly available MOBY atlas. This atlas was initially segmented into separate bones and joints which were placed to make it articulated[11]. While only one atlas is used throughout the series of papers, the authors suggest that adding a different model, mouse or otherwise, should not prevent the algorithm from working. CT data is preprocessed before registration; this preprocessing consists of smoothing and converting the volumetric data into a mesh using the marching cube algorithm[21]. The articulated atlas is roughly aligned to the CT mesh using center of gravity and pinning along various dimensions. Fine atlas alignment is done with the iterative closest point algorithm starting at the top level bone, the skull, in the skeletal bone hierarchy. Each bone undergoes local transformations that are initialized by the transformation of the parent bone in the hierarchy. High contrast organs, skin and lungs, are also registered based on intensity levels. Low contrast organs are registered based on the previously registered skeleton, skin, and lungs. Evaluation of registration is done using mean point to surface distance for each bone. Despite only transforming bones in up to 9 DoF (translation, rotation, scaling), mean point to surface distance for each bone is roughly 0.5mm. The data was originally reconstructed with a voxel size of 83µm x 83µm x83µm and resampled to a voxel size of 332µm x 332µm x332µm. This means that the registered atlas skeleton was on average 1.5 resampled voxels or 6 original voxels away from the CT bone surface. Another caveat of the algorithm is that it does not register the spine. The spine modeled as a 3D curve between the neck and the pelvis but it is not registered and cannot be segmented from the CT data.

Khmelinskii developed a method that registers the atlas system developed by Baiker to SPECT data taken using 99mTc-MDP[22], a tracer that is correlated to osteoblastic activity and therefore tends to highlight the bones, especially in regions of high bone turnover such as joints and the

spine. The difficulty in registering atlases to nuclear imaging data is that SPECT and PET data have significantly lower spatial resolution. Additionally, the method used to register the atlas relies on registering bones. This makes registering an atlas to a MI datasets generated from non-bone seeking tracers such as 99mTc-MIBI or 99mTc-MAA, difficult, if not impossible. Another difficulty in registering radionuclide bone tracer images is that bone tracers do not uptake in the entire bone, just regions of osteoblastic activity which is limited to growth plates in a healthy mouse. This could lead to inaccurate registration of bones with low levels of bone remodeling. The overarching issue with this method is that nuclear imaging studies are not typically done on healthy models. Applying this technique to MI datasets of bone disease models with could inhibit correct registration.

3. Specific Aims

The Small Animal Imaging Facility at the Van Andel Institute (VAI) is currently developing methods that will allow for increased automation of image processing. Atlas based segmentation and registration is a novel area of development and promises a decrease in image analysis time as well as more consistent results than manual image analysis. Basing design on current state-of-the-art articulated atlas registration, this thesis aims to develop software that segments and registers bones from fused CT/MI datasets of mice using an articulated atlas. Specific features of this software are:

- Robust registration despite moderate posture variations
- Robust registration despite different sized animal subjects
- Segmentation of major bones of skeleton
- Report nucleotide uptake from fused MI set for each segmented major bone

This project distinguishes itself from currently published research by applying atlas based segmentation to fused CT/MI datasets in order to gain information about radiotracer uptake.

4. Methodology

An outline of the algorithm is shown in Figure 3. For input the user is required to provide a nuclear medicine (PET or SPECT) dataset and a CT dataset that can be fused together. Datasets are fused when $nm(j) = T * ct(i)$, where T is an affine transform that maps the CT voxel location, $ct(i)$, to NM voxel location,$nm(j)$. If possible, this information is retrieved from the dataset, if not, the user can provide a transform or the identity transform is assumed.

The third input required by the algorithm is an articulated atlas. Different animal models have different anatomy; while a generic atlas might work well enough, a custom atlas designed on a specific model might work better. For the purposes of this project one atlas was used. This atlas was based on the publicly available MOBY mouse atlas[9] and the modifications described in Baiker[20] et al 2010. The MOBY mouse atlas is a non-uniform rational b-spline (NURBS) model of a mouse in 4D. An instance of the atlas was generated at full exhale of the respiratory cycle and end diastole of the cardiac cycle. From this instance, Baiker et al segmented individual bones and defined joint locations. These modifications were made publicly available and are the basis for the atlas used in this algorithm.

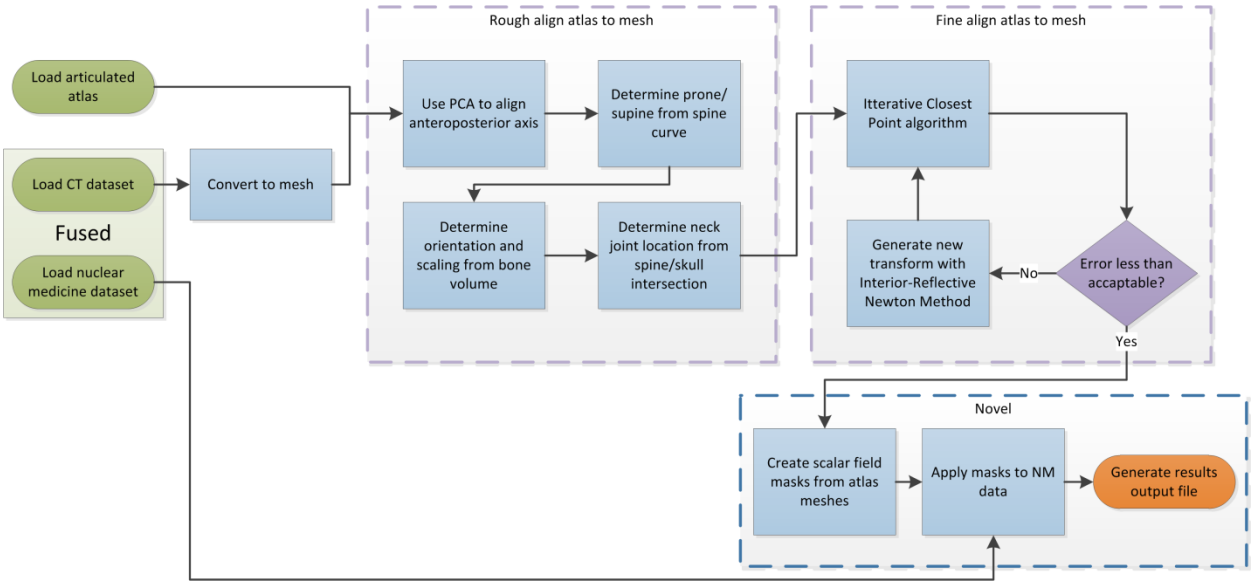

Figure 3: Generalized Software Flowchart. User must input a fused nuclear imaging/CT dataset and select an articulated atlas to use. In the rough align portion of the algorithm the CT data is preprocessing by smoothing and resampling. From this preprocessed data, orientation, scale, and initial alignment are determined. The find align section utilizes the Iterative closest point algorithm to align indivisuale bones. The aligned result is finally applied to the NM dataset as a mask to generate output that the user can view.

The automated segmentation method can be broken into three main segments. The outputs of these are illustrated in Figure 4.

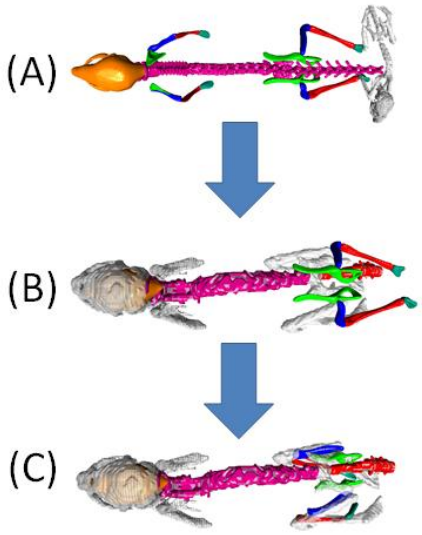

Figure 4: Steps of the alignment process. (A) Preprocessed input. Scale and orientation of atlas (colored) drastically different than CT dataset (gray). (B) Atlas rough aligned with CT dataset, notice hind limbs are not aligned. (C) Fine align complete.

Preprocessing of the input data consists of noise removal and volumetric field to surface mesh conversion in order to generate the CT skeleton mesh. To remove noise, the CT is filtered with a Gaussian filter, σ=2. Subsequently the dataset is down sampled to a voxel size of 0.6mm. This down sampling provides additional smoothing through spline interpolation as well as a reduction in the number of points in the dataset. An isosurface mesh of the skeleton was generated from the preprocessed volume using the Marching Cubes algorithm[21]. Marching cubes is a simple algorithm and developed algorithm[23] that is often used for converting volumetric fields to surface meshes. The algorithm breaks a volumetric dataset into small cubes of eight data points. Triangles are created to delineate the cube based on the isosurface value as shown in Figure 5.

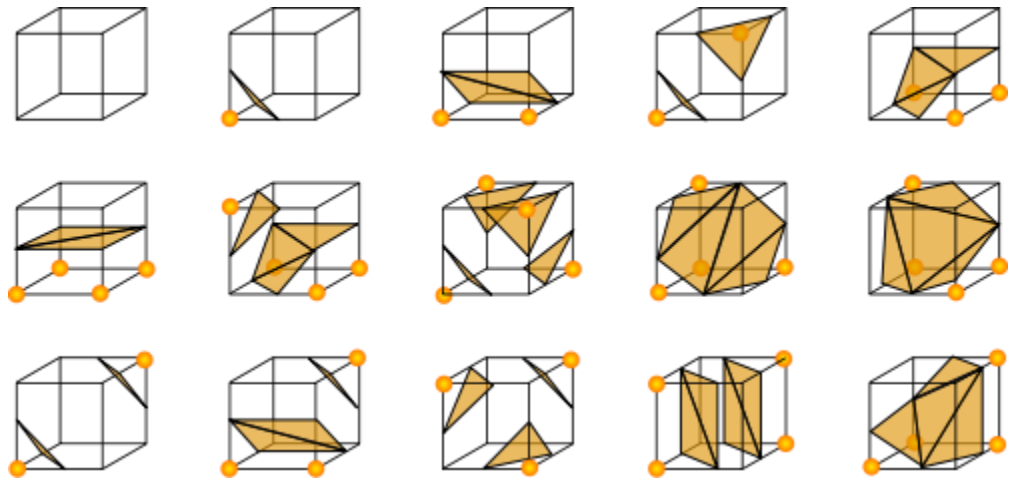

Figure 5: Fifteen possible combinations for cube vertex values. Verticies with orange dots are on one side of the isosurface value, vertices with no dots are on the other. Orange triangles segment the cube to seperate vertices that are on either side of the isosurface value. Reproduced from Wikimedia Commons.

The Iterative Closest Point (ICP) algorithm[24] was chosen as the method of aligning the atlas and CT meshes because of its extensive history and development[25]. ICP is used to align point clouds. It is commonly used in robotics for processing of light detection and ranging (LIDAR)[26] but is also a major algorithm in image processing. ICP works by iteratively aligning the data, D, to the

model, M. This is done by minimizing the error, mean Euclidean distance as shown in equation 1.

$$\frac{1}{N_D}\sum_{j=1}^{N_D}\min_{i=1..N_M}\left(\left\|M_i - TD_j\right\|^2\right) \quad 1.$$

N_M and N_D are the number of points in M and D respectively. T is a Euclidean transform consisting of rotation and translation applied to each point in D. For each iteration of ICP, T is calculated and applied to D with the result used as the updated value of D for the next iteration. ICP has been shown to converge to a local minimum[24]. The minimum that ICP converges to is dependent on the initial positions of the point clouds. Thus, good estimation of the initial position is required for ICP to converge to a global minimum. Figure 6 shows ICP being applied to the atlas left tibia in order to align it with the CT dataset.

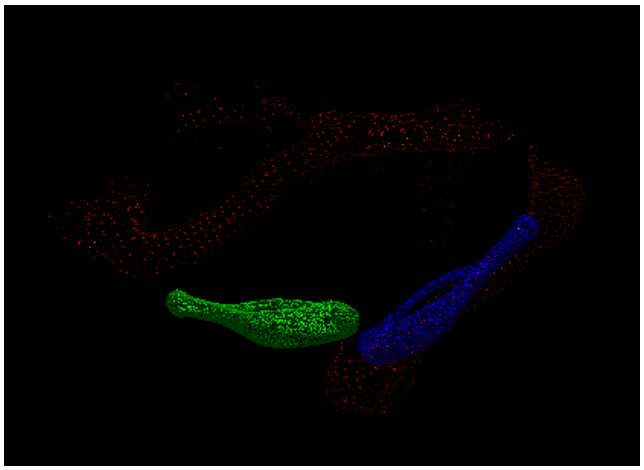

Figure 6: Alignment of Point Clouds. Red Point cloud represents left and right tibia and hindpaws of the CT dataset. The green point cloud is the atlas left tibia before ICP alignment. The blue point cloud is the atlas left tibia after alignment with the left tibia in the CT point cloud

4.1. Rough Align

The rough align serves as a method to generate an initial position for the atlas that will allow ICP to converge to the correct solution. In order to roughly align the atlas to the CT dataset in preparation for ICP orientation, location, and scale must be determined for the top level joint.

Principle Component Analysis (PCA) of the CT skeletal mesh point cloud generated three eigenvectors that represent the major axis of the mouse. The primary eigenvector is always the anteroposterior axis. The second and third eigenvectors represents either the dorsoventral or the left-right axis. The exact mapping of the second and third eigenvectors depends on the position of the mouse at scan time. With the limbs of the mouse positioned under the body, the second eigenvector represents the dorsoventral axis. When the limbs are positioned to the side of the mouse the third eigenvector represents the dorsoventral axis. The anterior vector can be determined by comparing the bone volume in the CT volume halves, where the halves are split by the plane normal to and midway along the anteroposterior axis. The half with the most bone volume contains the skull and is thus the direction of the anterior vector. Mice have a characteristic high bump in their spine in the dorsal vector. This vector can be found by first calculating the mean value, m_{dv}, of bone voxel locations along the dorsoventral axis. Then the mean value, m_{highdv}, of points whose dorsoventral axis position is greater than m_{dv} is determined as well as the mean, m_{lowdv}, of points whose dorsoventral axis position is less than m_{dv}. The greater value between $\left|m_{dv} - m_{highdv}\right|$ and $|m_{dv} - m_{lowdv}|$ determines the direction of the spine arch and therefore the dorsal vector. Determining the anterior vector and the dorsal vector is enough to constrain the 3 DoFs of orientation. Figure 7 shows the PCA axis for the atlas and CT datasets.

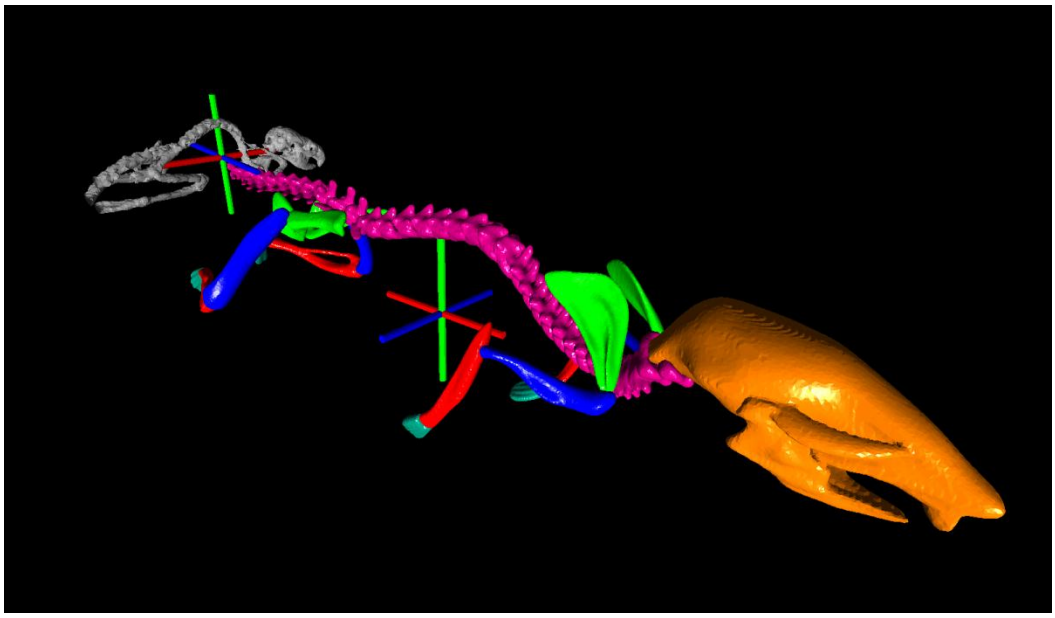

Figure 7: Atlas and Model datasets with PCA axis in the center of each dataset. Red axis is primary axis, green is secondary axis, and blue is tertiary axis. By aligning model axis and atlas axis, the two datasets will have the same orientation

Location of the spine can be determined from the coronal view. The forelimbs are connected via connective tissue to the spine. This connective tissue has a lower density than bone and careful selection of the isosurface value allows for the forelimbs to be separated from the rest of the skeleton. A 3D center of mass (CoM) curve is generated from the skeletal volume without forelimbs. The standard deviation of the bone voxels distance from this 3D curve in the transaxial plane gives an estimation of the thickness of bone along the anteroposterior axis. Moving towards posterior from the anterior, standard deviation of bone distance increases from the nose to the center of the skull, and then decreases until the spine is reached. Using this information the location of the neck can be determined as the point on the CoM line where the bone thickness has stopped decreases when moving from the anterior to the posterior.

To properly scale the atlas to the CT data, the ratio of the diagonal of two bounding boxes are used. A bounding box is the smallest rectangular solid that encompasses all points of a given set. The ratio of the diagonals between the bounding box of the atlas and the bounding box of the CT mesh is the rough isotropic scaling factor between the two datasets.

Using ICP to aligning individual vertebrae is technically possible, however the errors induced at each joint compound and registration of the lower limbs could suffer significantly. To prevent this, the spine is segmented from the volumetric data as the bone between the neck joint and the hip joint. The hip joint is located and used as the top level hierarchical joint for registration of the lower limbs. The spine is segmented as a 3D region started at the neck and grown in the posterior direction. When moving in the posterior direction from the neck the proximal end hip is first visible as two points detached from the spine. Continuing in the posterior direction these points will join the spine marking the location of the hip joint.

4.2. Fine Align

The generalized ICP algorithm consists of two parts, a matching function that determines corresponding points, and an error minimization function that minimizes error between corresponding points by determining a transform. This transform is applied to the data and the result is used as input for the next iteration of the ICP algorithm.

Corresponding points are pairs of points that should be aligned together. For each point in the data set D (atlas), a corresponding point in model set M (CT mesh), is chosen. Equation 2 shows the algorithm for choosing corresponding points based on point to point Euclidean distance. k is the iteration of ICP. T_{k-1} is the transform from the previous iteration, or the rough initial transform if $k = 0$.

$$c_k(i) = \min_{j=1,\dots,N_M} \left(\left\| M_j - T_{k-1} D_i \right\|^2 \right), i = 1, \dots, N_D \quad 2.$$

Brute force searching for closest points has a computational complexity of $O(N_M N_D)$. This represents a major portion of computation time for ICP. This algorithm uses the current state-of-the-art method for nearest neighbor searches in low dimensional space, the K-D Tree[27], which has an average complexity of $O(N_D \log N_M)$.

The second step of ICP is to minimize the error function and generate a new transform T_k, usually Euclidean distance, between corresponding points as shown in equation 3.

$$T_k = T_{k-1} * \underset{T_k}{\operatorname{argmin}} \left(\sum_{i=1}^{N_d} \left\| M_{c_k(i)} - T_k (T_{k-1} * D_i) \right\|^2 \right) \quad 3.$$

Several closed form solutions have been devised to solve the error minimization function and generate T_k. These methods use singular value decomposition[28], quaternions[29], dual-quaternions[30], or orthonormal matrices[29] and have all been previously surveyed[31]. These solutions are unbounded and unconstrained and may generate transformations that violate the inherent bounds and constraints of an anatomically realistic articulated atlas. For this reason minimization of the error function was calculated using a bounded and constrained interior point method[32–34]. The interior point method is a large scale algorithm that works for large, sparse problems as well as small, dense ones. Steps are calculated with a direct step method and a conjugated gradient method. The robustness of the interior point minimization solver allows it to work for both large datasets like the skull as well as small ones such as the paws.

Bones are rigid structures, so calculating a global transform to align the atlas to the CT data based on the low DoF rigid transforms generated by ICP would result in a poor match, especially for limbs. Allowing a higher DoF global transform would result in anatomically impossible

deformations to the bones. To maintain anatomical bone rigidity and achieve an acceptable registration of the atlas to CT data, the global transform is broken into several local transforms. Each of these local transforms is applied to a bone and all its "children". A diagram of the articulated atlas hierarchy as it applies to this project is shown in Figure 8. Using the rough align of the atlas to the neck as the initial transformation, ICP is used to align the atlas skull to the skull in the CT data. Using the rough alignment transform and the hip joint as a starting location, the left and right lower limbs are aligned. Alignment starts at the top level bone in the hierarchy and proceeds to the bottom with the rough initial transforms estimate for each bone being the cumulative transform of all parent bones in combination with eight orientation transforms defined by varying azimuth values of [0,90,180,270] degrees and elevation values of [-45, 45] degrees. ICP is used to align each bone for each of the initial rough aligns positions, the transform that results in the least error is used as the final alignment transform. A bone is considered aligned and ICP for that bone is terminated when the difference between the error in the last ICP iteration and the previous ICP iteration is less than 0.01mm.

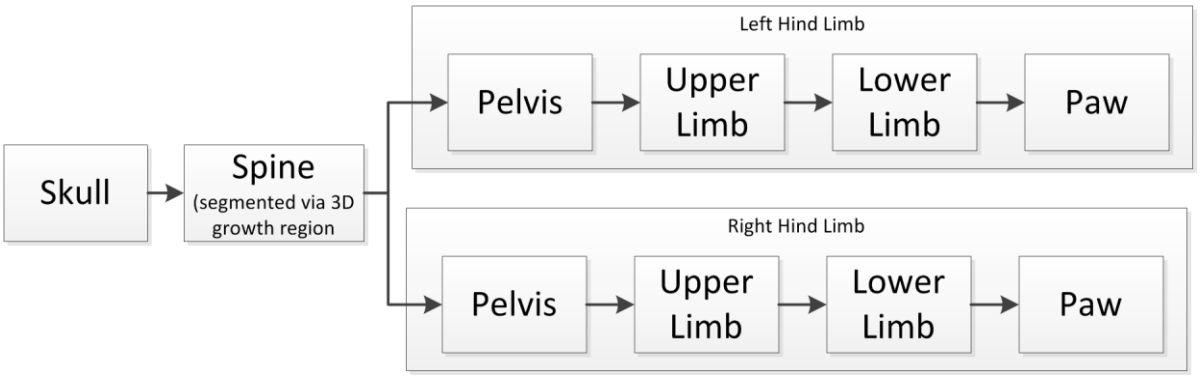

Figure 8: Bone hierarchy of MOBY articulated atlas. Transformations applied to a bone are also applied to all child bones

4.3. Segmenting Molecular Imaging Data

Once the articulated atlas is aligned to the CT data, segmentation of individual bone regions can occur. A mask volume is generated from the original CT dataset, and it is preprocessed identically to the methods discussed prior to the rough align, i.e. it is smoothed and the noise is removed. The major difference being that this volume is resampled to match the sample space of the PET dataset. Following the resampling, the skeleton is segmented by thresholding all points with a value greater than 350 Hounsfield units. To segment the skeleton into individual bone regions the nearest neighbor vertex in the aligned atlas is identified for each bone voxel. This voxel is then identified with the same bone group as the nearest neighbor atlas vertex. Applying the mask of individual bone groups to the PET dataset allows the segmentation of uptake values for each of the bone groups.

5. Experiment

5.1. Setup

Datasets were acquired from 5 mice models (NOD.CB17-*Prkdc*^{scid}/J, Jackson Laboratories, Bar Harbor, Maine) in a protocol approved by the Van Andel Institute Institutional Animal Care and Use Committee (IACUC). Approval by Van Andel Institute IACUC was deemed sufficient by Grand Valley State University IACUC.

A whole-body fused SPECT/CT dataset was acquired for each mouse using a nanoSPECT/CT imager, (Bioscan, Washington, D.C.). Mice were scanned in the feet first prone position. ^{99m}Tc-Methylene Diphosphonate was used as a radiotracer to measure osteoblastic activity in osseous tissue. Scans were reconstructed with a voxel size of 0.3mm x 0.3mm x 0.3mm for SPECT and 0.2mm x 0.2mm x 0.2mm for CT.

Each dataset was analyzed with the automated software described in section 4. Manual analysis of bone uptake was also determined.

5.2. Results

Visual inspection showed that the automated algorithm successfully registered 4 of the 5 datasets. The results from these 4 successful registrations were compared to the manual analysis of those datasets. Figure 9 and Figure 10 show the Euclidean distance between atlas surface mesh and the CT dataset surface mesh as registration progresses. Figure 11 shows the distance between atlas joint locations and manually determined joint locations before and after registration of the atlas to the CT data. Figure 12 shows the differences in radiotracer uptake

measurements between automated and manual analysis. All error bars represent a single standard deviation from mean.

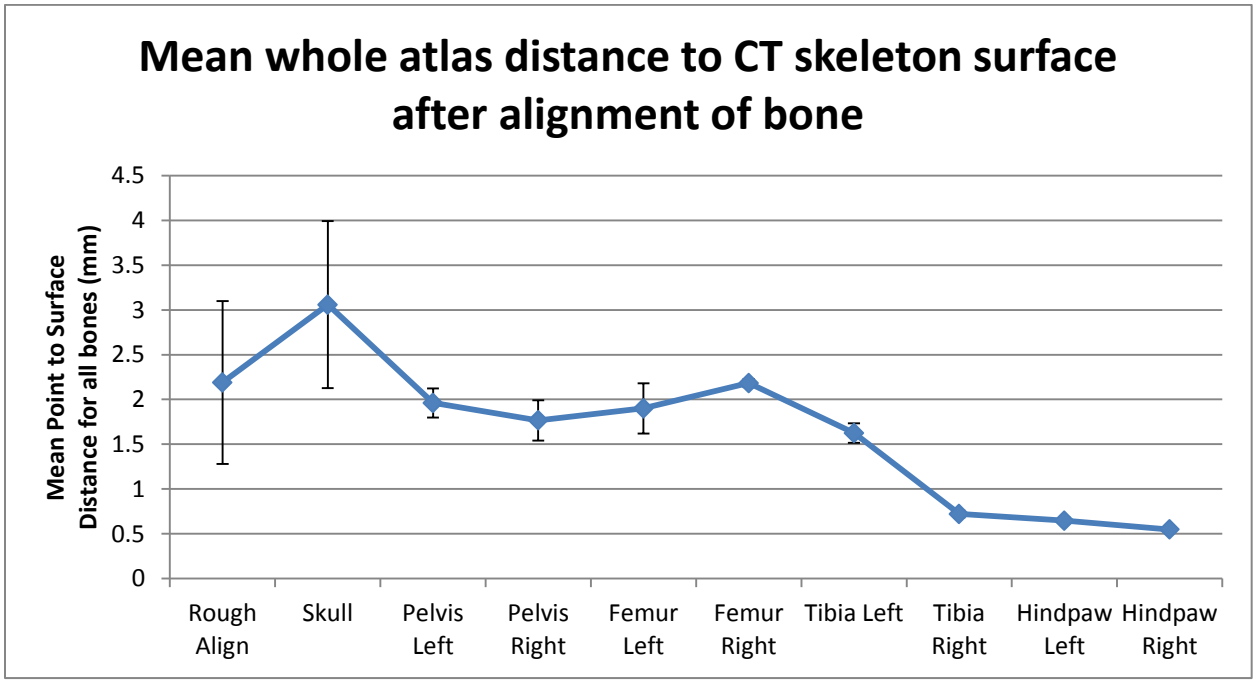

Figure 9: Mean distance between each vertex in the atlas and its nearest neighbor vertex in the CT dataset surface was determined for all atlas vertices after registration of each bone group. Bone groups are registered in the order listed. Refer to Figure 8 for bone hierarchy.

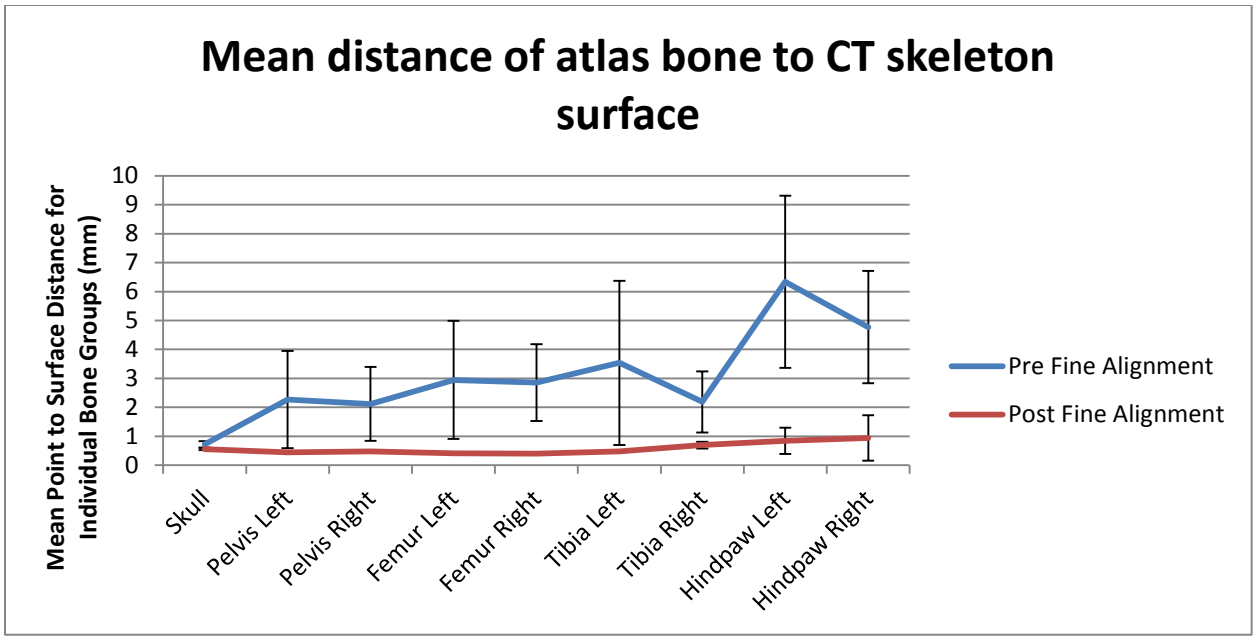

Figure 10: Mean distance between each vertex in an atlas bone group and its nearest neighbor vertex in the CT dataset surface was determined for each bone group vertices before and after registration of that bone group.

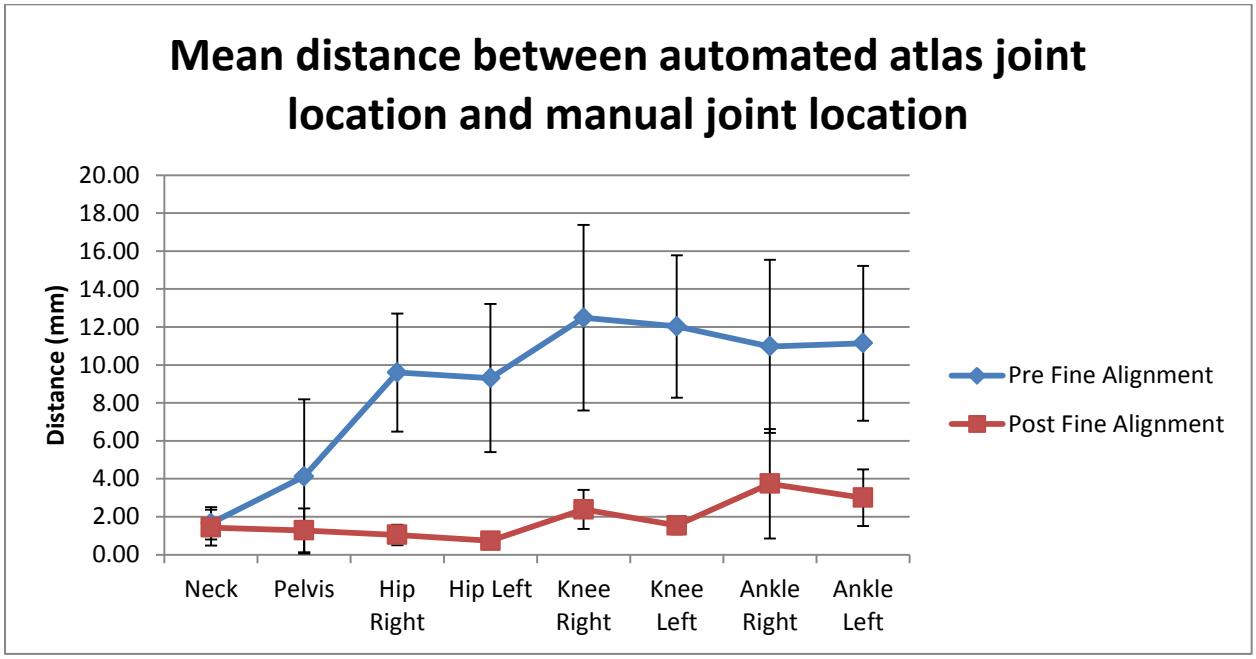

Figure 11: The 'true' joint location of each joint was determined by manual image analysis. The distance between this location and the location of the corresponding atlas joint was determined before and after registration of that joint.

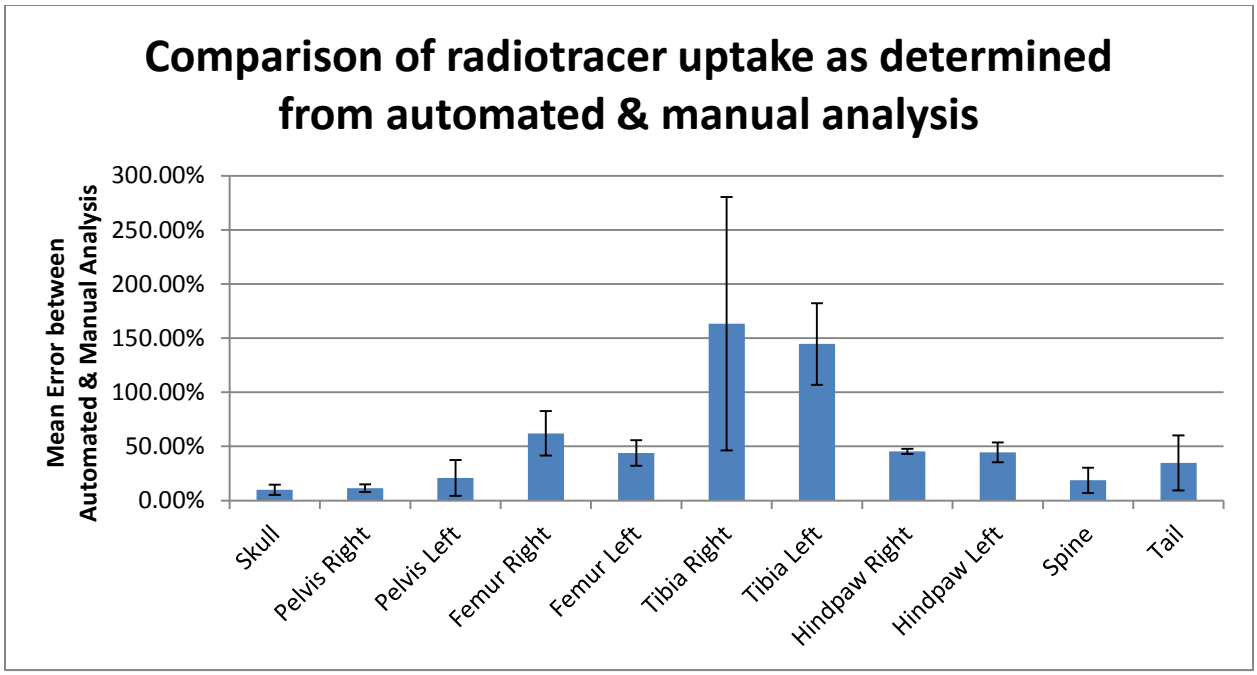

Figure 12: Radiotracer uptake was measured in 11 different bone groups using automated analysis and manual analysis. The error percentage from the automated results from the manual results is shown above. Error bars for each bone group represent standard deviation.

6. Discussion

The atlas segmentation method registered four out of the five datasets such that visual inspection ensured that no bones were placed in obviously wrong locations/orientations. In the instance where visual assessment deterred the algorithm failed, the initial rough alignment was significantly rotated from a 'good' rough alignment. A second dataset also exhibited a similar rotation but the fine align was able to compensate. Given a 'good enough' rough alignment all datasets could be aligned by ICP.

For those datasets that did successfully register automatically, the results are shown in section 5.2. Figure 9 shows the mean distance between the atlas and the CT dataset surface at each point in registration. As more bones are registered the mean distance between meshes decreases. Figure 10 shows the distance of each bone group from the CT dataset mesh before and after registration. It can be noted that bones further down the alignment hierarchy, as shown in Figure 8, exhibit greater error pre- and post- registration as well as larger variation. This is because error in registration of higher bones on the hierarchy accumulates and has a large effect of lower hierarchy bones. When comparing pre and post distances in Figure 10, the mean distance and variation between meshes is substantially reduced.

Similarly in Figure 11, the atlas joint locations are compared before and after fine alignment to the actual joint locations, as determined by manual analysis. Again, distances between automated and manual locations decreases dramatically post fine alignment. It can still be seen that joints lower in the bone hierarchy are further away from manually determined position than joints higher on the hierarchy.

Figure 9 through Figure 11 measure the quality of registration of the atlas to the CT dataset. While registration is generally good, it is worse for elements lower on the registration hierarchy. This is because error in higher elements is passed along and accumulated in lower elements. The main cause of error is that the isotropic scale of the atlas is estimated once in the rough align. This rough scale estimation causes joints, and subsequently bones, to be placed further and further from their actually position for those joints lower on the hierarchy. Baiker[20] et al. successfully addressed this issue by implementing a scaling feature in the fine align section of their algorithm that dramatically reduced the error passed on the lower hierarchical elements.

Figure 12 shows the error percentage between radiotracer uptake measurements from manual and automated analysis. Here again, elements lower on the registration hierarchy exhibit more error than those higher on the hierarchy. This error seems to be extreme (250%) when compared to previously mentioned errors. This large exaggeration is caused by the characteristics of the radiotracer used. ^{99m}Tc-MDP is incorporated into the bone matrix by osteoblasts; osteoblasts are active in regions of bone growth. Since the imaged mice did not have fractures, the osteoblast activity was almost exclusively limited to the bone growth plates near the joints. Since the joints are the areas most affected by the errors, this can cause uptake amounts to be off by orders of magnitude. Figure 13 shows the labeled volume used as a mask for the NM dataset. Each gray level represents a different bone and should color only that bone. The blue arrow points to the right pelvis which is well registered. The orange dots represent the joint locations as determined by the automated algorithm while the green dots are the actual joint positions determined manually. The distance between the automated hip joint location and the manual hip joint location is minimal. However, because of inadequate bone scaling, the error in joint location is propagated and the differences between automated and manual joint location for the knee (red

arrow) is much greater. In reality, half the knee is part of the femur and half of it is part of the tibia. However, in this case, the entire knee has been mapped as the right tibia, Figure 14. Since the knee is where the vast majority of the MDP uptake occurs (as opposed to the diaphysis region), incorrect assignment of this small region of bone can cause extremely large errors in uptake measurements. Adding non-isotropic scaling to the fine alignment for each bone will decrease propagated joint location error and result in more accurate radiotracer uptake values.

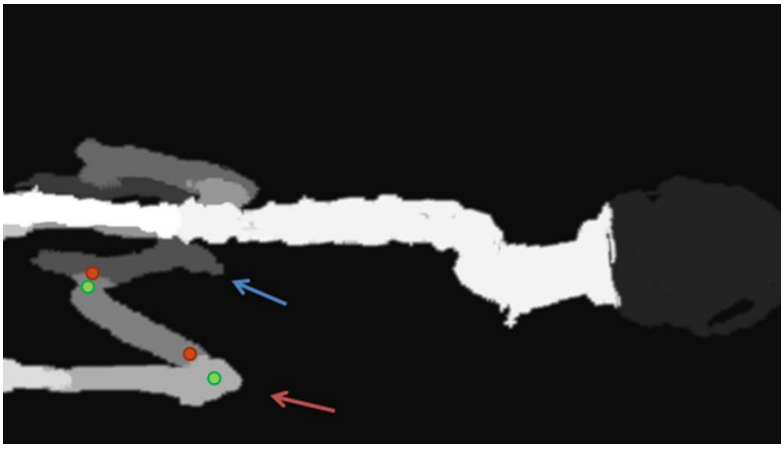

Figure 13: Map of voxels assigned to bones based on atlas registration. Different gray levels represent different bones. Blue arrow is the right pelvis, Red arrow points to the right knee. The orange dots represent joint locations as determined by the automated software. Green dots represent actual joint positions. The distance between automatic and actual joint positions is relatively small for the hip but larger for the knee. This is error is propagated down the joint hierarchy because of inadequate scaling.

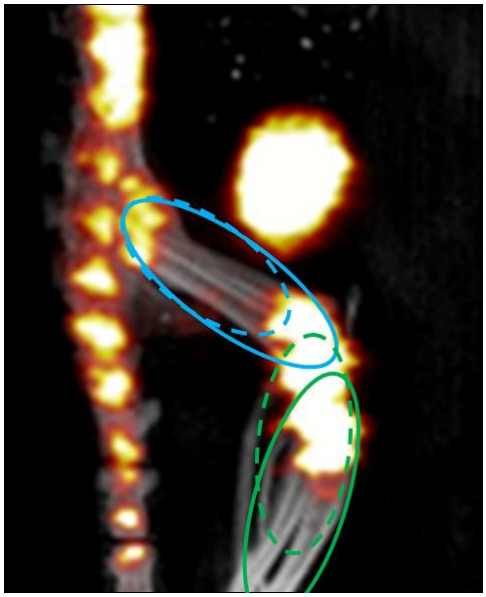

Figure 14: Fused CT/SPECT (^{99m}Tc-MDP) image. Blue lines represent ROI around the femur, green lines represent ROI around the tibia. Solid lines represent good ROIs, dashed lines represent bad ROIs caused by scaling problems. Normally activity in the knee is split between femur and tibia however, error caused by scaling results in most of the activity going to tibia.

7. Conclusion

This thesis presents a fully automated atlas-based whole body analysis for fused NM/CT dataset as a method to overcome the image processing bottleneck of preclinical *in vivo* imaging. The registration process consists of two main steps, a rough alignment based on the anatomically relevant information, and an element wise, fine alignment based on the iterative closest point method. As a fully automated method, this software is suitable for batch analysis, resulting in dramatic reduction of technician image analysis time and technician bias.

Analysis results are highly dependent on the position of the volume of interest on the registration hierarchy. Volumes registered early in the alignment process such as the spine, skull, and pelvis have much lower error levels than volumes registered at the end like the tibias and hind paws. When volume of interest error is accounted for, this method provides a robust way of analyzing nuclear medicine datasets that has not been done previously in literature.

Future improvements to this method should focus on the minimization of radiotracer uptake measurement error. Progress towards this goal can be greatly facilitated by the addition of automatic non-isotropic scaling during the fine alignment process. The nature of this method makes it applicable for the use of different atlases, potentially including atlases of different species. In addition to changing the atlas, a method of determining the rough alignment transform would need to be devised for each additional species.

This method was designed to analyze NM datasets and thus does not account for the use of CT contrast. Adding CT contrast, however, can highlight various soft tissue organs. Using this information, in addition to the anatomical information provided by a registered articulated atlas, segmentation of soft tissue organs, such as the liver, might be possible to automate.

8. Bibliography

1. Laboratory of Neuro Imaging, UCLA; http://www.loni.ucla.edu/Atlases/. at <http://www.loni.ucla.edu/Atlases/>

2. Talairach, J. & Tournoux, P. *Co-planar stereotaxic atlas of the human brain: 3-dimensional proportional system: an approach to cerebral imaging*. (Thieme: 1988).

3. The Visible Human Project; http://www.nlm.nih.gov/research/visible/visible_human.html. at <http://www.nlm.nih.gov/research/visible/visible_human.html>

4. Segars, W.P., Lalush, D.S. & Tsui, B.M.W. Modeling respiratory mechanics in the MCAT and spline-based MCAT phantoms. *Nuclear Science, IEEE Transactions on* **48**, 89-97 (2001).

5. Zaidi, H. & Tsui, B.M.W. Review of computational anthropomorphic anatomical and physiological models. *Proceedings of the IEEE* **97**, 1938-1953 (2009).

6. Caon, M. Voxel-based computational models of real human anatomy: a review. *Radiation and environmental biophysics* **42**, 229-235 (2004).

7. Zankl, M. *et al.* Realistic computerized human phantoms. *Advances in Space Research* **14**, 423-431 (1994).

8. Dogdas, B., Stout, D., Chatziioannou, A.F. & Leahy, R.M. Digimouse: a 3D whole body mouse atlas from CT and cryosection data. *Physics in medicine and biology* **52**, 577 (2007).

9. Segars, W.P. *et al.* Development of a 4-D digital mouse phantom for molecular imaging research. *Molecular Imaging and Biology* **6**, 149-159 (2004).

10. Bai, X. *et al.* A high-resolution anatomical rat atlas. *Journal of anatomy* **209**, 707-708 (2006).

11. Khmelinskii, A. *et al.* Articulated Whole-Body Atlases for Small Animal Image Analysis: Construction and Applications. *Molecular Imaging and Biology* 1-13 (2010).

12. Li, X., Yankeelov, T.E., Peterson, T.E., Gore, J.C. & Dawant, B.M. Automatic nonrigid registration of whole body CT mice images. *Medical physics* **35**, 1507 (2008).

13. Chaudhari, A.J., Joshi, A.A., Darvas, F. & Leahy, R.M. A method for atlas-based volumetric registration with surface constraints for optical bioluminescence tomography in small animal imaging. *Proc. SPIE Medical Imaging* **6510**, (2007).

14. Wang, Y., Gu, X. & Yau, S.T. Volumetric harmonic map. *Communications in Information & Systems* **3**, 191-202 (2003).

15. Kovacevic, N., Hamarneh, G. & Henkelman, M. Anatomically guided registration of whole body mouse MR images. *Medical Image Computing and Computer-Assisted Intervention-MICCAI 2003* 870-877 (2003).

16. Martín-Fernández, M.Á. *et al.* Automatic articulated registration of hand radiographs. *Image Vision Comput.* **27**, 1207-1222 (2009).

17. Papademetris, X., Dione, D., Dobrucki, L., Staib, L. & Sinusas, A. Articulated rigid registration for serial lower-limb mouse imaging. *Medical Image Computing and Computer-Assisted Intervention-MICCAI 2005* 919-926 (2005).

18. Baiker, M. *et al.* Fully automated whole-body registration in mice using an articulated skeleton atlas. *Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on* 728-731 (2007).

19. Baiker, M. *et al.* Organ approximation in μCT data with low soft tissue contrast using an articulated whole-body atlas. *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on* 1267-1270 (2008).

20. Baiker, M. *et al.* Atlas-based whole-body segmentation of mice from low-contrast Micro-CT data. *Medical Image Analysis* **14**, 723-737 (2010).

21. Lorensen, W.E. & Cline, H.E. Marching cubes: A high resolution 3D surface construction algorithm. *ACM Siggraph Computer Graphics* **21**, 163-169 (1987).

22. Khmelinskii, A. *et al.* Atlas-based articulated skeleton segmentation of μSPECT mouse data. *Biomedical Imaging: From Nano to Macro, 2011 IEEE International Symposium on* 437-440 (2011).

23. Newman, T.S. & Yi, H. A survey of the marching cubes algorithm. *Computers & Graphics* **30**, 854-879 (2006).

24. Besl, P.J. & McKay, N.D. A Method for Registration of 3-D Shapes. *IEEE Trans. Pattern Anal. Mach. Intell.* **14**, 239-256 (1992).

25. Rusinkiewicz, S. & Levoy, M. Efficient variants of the ICP algorithm. *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on* 145-152 (2001).

26. Ridene, T. & Goulette, F. Registration of fixed-and-mobile-based terrestrial laser data sets with DSM. *Computational Intelligence in Robotics and Automation (CIRA), 2009 IEEE International Symposium on* 375-380 (2009).

27. Bentley, J.L. Multidimensional binary search trees used for associative searching. *Communications of the ACM* **18**, 509-517 (1975).

28. Arun, K.S., Huang, T.S. & Blostein, S.D. Least-squares fitting of two 3-D point sets. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 698-700 (1987).

29. Horn, B.K.P. Closed-form solution of absolute orientation using unit quaternions. *JOSA A* **4**, 629-642 (1987).

30. Walker, M.W., Shao, L. & Volz, R.A. Estimating 3-D location parameters using dual number quaternions. *CVGIP: image understanding* **54**, 358-367 (1991).

31. Eggert, D.W., Lorusso, A. & Fisher, R.B. Estimating 3-D rigid body transformations: a comparison of four major algorithms. *Machine Vision and Applications* **9**, 272-290 (1997).

32. Byrd, R.H., Hribar, M.E. & Nocedal, J. An interior point algorithm for large-scale nonlinear programming. *SIAM Journal on Optimization* **9**, 877-900 (1999).

33. Waltz, R., Morales, J., Nocedal, J. & Orban, D. An interior algorithm for nonlinear optimization that combines line search and trust region steps. *Mathematical Programming* **107**, 391-408 (2006).

34. Byrd, R.H., Gilbert, J.C. & Nocedal, J. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming* **89**, 149-185 (2000).

Appendix A: Software Code

pyAtlasSegmentation.py

```
'''
Created on Jul 27, 2011

@author: grant

Ideas for Improvement
1.      Create Ball&Socket joint class as well as Hinge joint class that inherit
        from joint class, these classes will call a specific version of
        joint.rotate(). The ball&socket joint is very similar to spherical except
        the vector that the model spins around can be arbitrary (spherical spin
        vector is [0,0,1]. Relevant methods of moving a joint should be selectable
        from combobox on GUI, i.e. a Hinge joint could only move in hinge mode, but
        a joint could move in rectangular, spherical, hinge, or ball&socket mode.

2.      Different Coordinate systems should have different axis models. Rotation
        in rectangular system should display different axis then a rotation in
        spherical system. Similarly a Hinge joint and a Ball&Socket joint should
        have unique axis models

4.      Fix determining extends box so it only has to do that once, then saves the
        values (maybe, might not be a good idea if model size could change)

5.      Set up orthographic projection view

6.      Setup Atlas so that default joint orientation can be set. Also joint types
        should be setable in the atlas xml

'''
import sys, math, time
import numpy
from scipy.ndimage.filters import gaussian_filter
import TriModel
from Joint import Joint
from PySide import QtCore, QtGui
from OpenGLGUI import Ui_MainWindow
from openGLWidget import GLWidget
from STL import readSTLfile,saveSTLFile
from lxml import etree
import os.path
from BinVoxReader import BinVox
from polygonise import Polygonise
```

```
import dicom
import AlignAtlas
from InputDataTypes import UserDicom, UserAtlas
import numpyTransform
import InputDataTypes
import scipy.stats
from scipy.io.matlab.mio import savemat
from scipy.spatial import cKDTree
from cgkit.cgtypes import quat
from OpenGL import GL

class MyMainWindow(QtGui.QMainWindow):
      def __init__(self, parent=None):
            self.lastStep = 0

            super(MyMainWindow, self).__init__(parent)
            self.ui = Ui_MainWindow()
            self.ui.setupUi(self)

            self.setTabPosition(QtCore.Qt.DockWidgetArea.AllDockWidgetAreas,
QtGui.QTabWidget.TabPosition.North)
            #Set up Docks in a tab format
            self.addDockWidget(QtCore.Qt.DockWidgetArea.RightDockWidgetArea,
self.ui.dockJoint)
            self.tabifyDockWidget(self.ui.dockJoint, self.ui.dockModel)
            self.tabifyDockWidget(self.ui.dockJoint, self.ui.dockSlice)
            self.tabifyDockWidget(self.ui.dockJoint, self.ui.dockLights)
            self.tabifyDockWidget(self.ui.dockJoint, self.ui.dockCamera)
            self.tabifyDockWidget(self.ui.dockJoint, self.ui.dockScene)
            self.tabifyDockWidget(self.ui.dockJoint, self.ui.dockTol)
            self.addDockWidget(QtCore.Qt.DockWidgetArea.LeftDockWidgetArea,
self.ui.dockItemList)

            for child in self.children():
                  if isinstance(child, QtGui.QTabBar) and child.count() == 7:
      #make sure proper number of tabs
                        child.setCurrentIndex(0)

            self.glWidget = GLWidget()
            self.setupSceneControls()

            self.ui.openGlTab.layout().addWidget(self.glWidget)
            QtCore.QObject.connect(self.ui.saveCSV,QtCore.SIGNAL("clicked()"),
self.saveCSV)
            QtCore.QObject.connect(self.ui.copyTable,QtCore.SIGNAL("clicked()"),
self.copyTable)

      QtCore.QObject.connect(self.ui.chooseSpecularColorCam,QtCore.SIGNAL("clicked()
"), self.setSpecularColorCam)

      QtCore.QObject.connect(self.ui.chooseDiffuseColorCam,QtCore.SIGNAL("clicked()"
), self.setDiffuseColorCam)

      QtCore.QObject.connect(self.ui.chooseAmbientColorCam,QtCore.SIGNAL("clicked()"
), self.setAmbientColorCam)
```

```
        QtCore.QObject.connect(self.ui.chooseEmissiveColorCam,QtCore.SIGNAL("clicked()
"), self.setEmissiveColorCam)

        QtCore.QObject.connect(self.ui.chooseSpecularColorMat,QtCore.SIGNAL("clicked()
"), self.setSpecularColorMat)

        QtCore.QObject.connect(self.ui.chooseDiffuseColorMat,QtCore.SIGNAL("clicked()"
), self.setDiffuseColorMat)

        QtCore.QObject.connect(self.ui.chooseAmbientColorMat,QtCore.SIGNAL("clicked()"
), self.setAmbientColorMat)

        QtCore.QObject.connect(self.ui.chooseEmissiveColorMat,QtCore.SIGNAL("clicked()
"), self.setEmissiveColorMat)

        QtCore.QObject.connect(self.ui.matShinySlider,QtCore.SIGNAL("valueChanged(int)
"), self.setShininessColorMat)

        QtCore.QObject.connect(self.ui.matShinySpinBox,QtCore.SIGNAL("valueChanged(dou
ble)"), self.setShininessColorMat)
                QtCore.QObject.connect(self.ui.modelVisible,
QtCore.SIGNAL("stateChanged(int)"), self.setModelVisibility)


        QtCore.QObject.connect(self.ui.setView,QtCore.SIGNAL("currentIndexChanged(QStr
ing)"), self.setView)

        QtCore.QObject.connect(self.ui.lightNum,QtCore.SIGNAL("valueChanged(int)"),
self.updateLightPropertiesTab)

        QtCore.QObject.connect(self.ui.lightXPos,QtCore.SIGNAL("valueChanged(double)")
, self.setLightXPos)

        QtCore.QObject.connect(self.ui.lightYPos,QtCore.SIGNAL("valueChanged(double)")
, self.setLightYPos)

        QtCore.QObject.connect(self.ui.lightZPos,QtCore.SIGNAL("valueChanged(double)")
, self.setLightZPos)

        QtCore.QObject.connect(self.ui.lightEnable,QtCore.SIGNAL("stateChanged(int)"),
self.setLightEnable)

        QtCore.QObject.connect(self.ui.directionalLight,QtCore.SIGNAL("stateChanged(in
t)"), self.setLightDirectional)

        QtCore.QObject.connect(self.ui.itemTreeList,QtCore.SIGNAL("itemSelectionChange
d()"), self.itemChangedCB)

        QtCore.QObject.connect(self.ui.jointXRotSlider,QtCore.SIGNAL("valueChanged(int
)"), self.rotateJointSlider)

        QtCore.QObject.connect(self.ui.jointYRotSlider,QtCore.SIGNAL("valueChanged(int
)"), self.rotateJointSlider)
```

```
        QtCore.QObject.connect(self.ui.jointZRotSlider,QtCore.SIGNAL("valueChanged(int
)"), self.rotateJointSlider)

        QtCore.QObject.connect(self.ui.jointXRotSpinBox,QtCore.SIGNAL("valueChanged(do
uble)"), self.rotateJointSpinBox)

        QtCore.QObject.connect(self.ui.jointYRotSpinBox,QtCore.SIGNAL("valueChanged(do
uble)"), self.rotateJointSpinBox)

        QtCore.QObject.connect(self.ui.jointZRotSpinBox,QtCore.SIGNAL("valueChanged(do
uble)"), self.rotateJointSpinBox)

        QtCore.QObject.connect(self.ui.rotationOrder,QtCore.SIGNAL("currentIndexChange
d(QString)"), self.rotationOrderChanged)

        QtCore.QObject.connect(self.ui.sphericalCoord,QtCore.SIGNAL("stateChanged(int)
"), self.setRotateCoord)

        QtCore.QObject.connect(self.ui.enableColorDrivenModelsCheckBox,QtCore.SIGNAL("
stateChanged(int)"), self.setColorModel)

        QtCore.QObject.connect(self.ui.pointSize,QtCore.SIGNAL("valueChanged(double)")
, self.pointSize)

                #callbacks for slice view changes

        QtCore.QObject.connect(self.ui.spinBox100AxisSlice,QtCore.SIGNAL("valueChanged
(int)"), self.setSlice100Axis)

        QtCore.QObject.connect(self.ui.slider100AxisSlice,QtCore.SIGNAL("valueChanged(
int)"), self.setSlice100Axis)

        QtCore.QObject.connect(self.ui.spinBox010AxisSlice,QtCore.SIGNAL("valueChanged
(int)"), self.setSlice010Axis)

        QtCore.QObject.connect(self.ui.slider010AxisSlice,QtCore.SIGNAL("valueChanged(
int)"), self.setSlice010Axis)

        QtCore.QObject.connect(self.ui.spinBox001AxisSlice,QtCore.SIGNAL("valueChanged
(int)"), self.setSlice001Axis)

        QtCore.QObject.connect(self.ui.slider001AxisSlice,QtCore.SIGNAL("valueChanged(
int)"), self.setSlice001Axis)

                QtCore.QObject.connect(self.ui.GrabScreen, QtCore.SIGNAL("clicked()"),
self.grabScreen)
                QtCore.QObject.connect(self.ui.automate, QtCore.SIGNAL("clicked()"),
self.automate)
                QtCore.QObject.connect(self.ui.camUpdate, QtCore.SIGNAL("clicked()"),
self.camUpdate)
                QtCore.QObject.connect(self.ui.invertNormals,
QtCore.SIGNAL("clicked()"), self.invertNormals)
                QtCore.QObject.connect(self.ui.saveModel, QtCore.SIGNAL("clicked()"),
self.saveModel)
```

```
            QtCore.QObject.connect(self.ui.loadModel, QtCore.SIGNAL("clicked()"),
self.loadModel)
            QtCore.QObject.connect(self.ui.loadAtlas, QtCore.SIGNAL("clicked()"),
self.loadAtlas)
            QtCore.QObject.connect(self.ui.defaultScene, QtCore.SIGNAL("clicked()"),
self.createDefaultScene)
            QtCore.QObject.connect(self.ui.clearScene, QtCore.SIGNAL("clicked()"),
self.clearScene)
            QtCore.QObject.connect(self.ui.projection,
QtCore.SIGNAL("currentIndexChanged(int)"), self.setProjection)
            QtCore.QObject.connect(self.ui.useCallLists,
QtCore.SIGNAL("stateChanged(int)"), self.setUseCallLists)
            QtCore.QObject.connect(self.ui.jointScaleX,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
            QtCore.QObject.connect(self.ui.jointScaleY,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
            QtCore.QObject.connect(self.ui.jointScaleZ,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
            QtCore.QObject.connect(self.ui.jointScaleXYZ,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScaleIsometric)
            QtCore.QObject.connect(self.ui.jointName,
QtCore.SIGNAL("editingFinished()"), self.setJointName)
            QtCore.QObject.connect(self.ui.modelName,
QtCore.SIGNAL("editingFinished()"), self.setModelName)
            QtCore.QObject.connect(self.ui.jointXPosStep,
QtCore.SIGNAL("clicked()"), self.moveJointX)
            QtCore.QObject.connect(self.ui.jointYPosStep,
QtCore.SIGNAL("clicked()"), self.moveJointY)
            QtCore.QObject.connect(self.ui.jointZPosStep,
QtCore.SIGNAL("clicked()"), self.moveJointZ)

      QtCore.QObject.connect(self.ui.modelAlpha,QtCore.SIGNAL("valueChanged(int)"),
self.setModelAlpha)

            #tolerence vector callbacks

      QtCore.QObject.connect(self.ui.tolXRotSlider,QtCore.SIGNAL("valueChanged(int)"
), self.rotateTolSlider)

      QtCore.QObject.connect(self.ui.tolYRotSlider,QtCore.SIGNAL("valueChanged(int)"
), self.rotateTolSlider)

      QtCore.QObject.connect(self.ui.tolZRotSlider,QtCore.SIGNAL("valueChanged(int)"
), self.rotateTolSlider)

      QtCore.QObject.connect(self.ui.tolXRotSpinBox,QtCore.SIGNAL("valueChanged(doub
le)"), self.rotateTolSpinBox)

      QtCore.QObject.connect(self.ui.tolYRotSpinBox,QtCore.SIGNAL("valueChanged(doub
le)"), self.rotateTolSpinBox)

      QtCore.QObject.connect(self.ui.tolZRotSpinBox,QtCore.SIGNAL("valueChanged(doub
le)"), self.rotateTolSpinBox)
```

```
        QtCore.QObject.connect(self.ui.tolVecAngle,QtCore.SIGNAL("valueChanged(double)
"), self.makeTolVec)

        QtCore.QObject.connect(self.ui.tolVecLength,QtCore.SIGNAL("valueChanged(double
)"), self.updateTolVec)

        QtCore.QObject.connect(self.ui.tolVecXOrigin,QtCore.SIGNAL("valueChanged(doubl
e)"), self.updateTolVec)

        QtCore.QObject.connect(self.ui.tolVecYOrigin,QtCore.SIGNAL("valueChanged(doubl
e)"), self.updateTolVec)

        QtCore.QObject.connect(self.ui.tolVecZOrigin,QtCore.SIGNAL("valueChanged(doubl
e)"), self.updateTolVec)

                #link keypress event for nuclear medicine table
                self.ui.nmTable.keyPressEvent = self.nmTableKeyPressEvent

                self.updateLightPropertiesTab(self.ui.lightNum.value())
                self.setColorModel(self.ui.enableColorDrivenModelsCheckBox.checkState())
                #configure color mode
                if self.ui.enableColorDrivenModelsCheckBox.checkState() ==
QtCore.Qt.CheckState.Checked:
                        self.glWidget.setColorDrivenMaterial(True)
                else:
                        self.glWidget.setColorDrivenMaterial(False)
                self.selectedJoint = None
                self.selectedModel = None
                self.referenceModelJoint = None
                self.atlasJoint = None
                self.tolJoint = None
                self.atlasAxes = numpy.identity(3)      #TODO: this should really be in
an Atlas class
                self.scanData = None

        def pointSize(self, size):
#               print 'Previous Point Size', GL.glGetDouble(GL.GL_POINT_SIZE)
                if self.selectedModel is not None:
                        self.selectedModel.pointSize = size
                self.glWidget.updateGL()

        def nmTableKeyPressEvent(self, event):
                key = event.key()
                modifer = event.modifiers()

                if key == 67 and modifer == QtCore.Qt.ControlModifier:      #CTRL + c
                        self.copyTable()

        def tableTostring(self, rowDelimiter='\n', columnDelimiter='\t',
wholeTable=False):
                selectedRanges = self.ui.nmTable.selectedRanges()
                if len(selectedRanges) == 0 or wholeTable is True:
                        left = 0
                        right = self.ui.nmTable.columnCount()
```

```
                        top = 0
                        bottom = self.ui.nmTable.rowCount()
                else:
                        left = selectedRanges[0].leftColumn()
                        right = selectedRanges[0].rightColumn()+1
                        top = selectedRanges[0].topRow()
                        bottom = selectedRanges[0].bottomRow()+1
                text = 'Region' + columnDelimiter
                #create column headers
                for j in xrange(left, right):
                        text += self.ui.nmTable.horizontalHeaderItem(j).text() +
columnDelimiter
                text += rowDelimiter

                #create data entries
                for i in xrange(top,bottom):
                        text += self.ui.nmTable.verticalHeaderItem(i).text() +
columnDelimiter
                        for j in xrange(left, right):
                                text += self.ui.nmTable.item(i,j).text() + columnDelimiter
                        text += rowDelimiter
                return text

        def saveCSV(self):
                filepath, selectedFilter =
QtGui.QFileDialog.getSaveFileName(caption="Save Nuclear Medicine Table As",
filter="CSV (*.csv);;Any File (*.*)") #@UnusedVariable
                if len(filepath) == 0:
                        return
                f = file(filepath,'w')
                f.write(self.tableTostring(columnDelimiter=',', wholeTable=True))
                f.close()

        def copyTable(self):
                QtGui.QClipboard().setText(self.tableTostring(wholeTable=True))

        def setModelAlpha(self,alpha,model=None):
                if model is None:
                        model = self.selectedModel
                if model is not None:
                        if len(model.specularColor) == 3:
                                model.specularColor.append(alpha / 255.0)
                        elif len(model.specularColor) > 3:
                                model.specularColor[3] = alpha / 255.0
                        if len(model.ambientColor) == 3:
                                model.specularColor.append(alpha / 255.0)
                        elif len(model.ambientColor) > 3:
                                model.ambientColor[3] = alpha / 255.0
                        if len(model.diffuseColor) == 3:
                                model.diffuseColor.append(alpha / 255.0)
                        elif len(model.diffuseColor) > 3:
                                model.diffuseColor[3] = alpha / 255.0
                        if len(model.emissionColor) == 3:
                                model.emissionColor.append(alpha / 255.0)
                        elif len(model.emissionColor) > 3:
```

```
                        model.emissionColor[3] = alpha / 255.0
                self.glWidget.updateGL()

        def setupSceneControls(self):
                '''function that populates/resets all scene controls'''
                self.populateItemTree(self.glWidget.worldJoint)
                self.selectedJoint = None
                self.selectedModel = None
                self.ui.jointName.setText('No Joint Selected')
                self.ui.modelName.setText('No Model Selected')

        def setUseCallLists(self, state):
                if state == QtCore.Qt.CheckState.Checked:
                        self.glWidget.useCallLists = True
                else:
                        self.glWidget.useCallLists = False

        def setJointScale(self, scale):
                if self.selectedJoint is not None:

        self.selectedJoint.scale(self.ui.jointScaleX.value(),self.ui.jointScaleY.value
(),self.ui.jointScaleZ.value())
                        self.glWidget.updateGL()

        def setJointScaleIsometric(self, scale):
                if self.selectedJoint is not None:
                        self.selectedJoint.scale(self.ui.jointScaleXYZ.value())
                        self.glWidget.updateGL()


        def setProjection(self, i):
                if i==0:
                        self.glWidget.prespective=True
                elif i==1:
                        self.glWidget.prespective=False
                self.glWidget.resizeGL(self.glWidget.width(), self.glWidget.height())

        def camUpdate(self):
                self.ui.camXPos.setValue(self.glWidget.camera.camLoc[0])
                self.ui.camYPos.setValue(self.glWidget.camera.camLoc[1])
                self.ui.camZPos.setValue(self.glWidget.camera.camLoc[2])
                self.ui.camXLookAt.setValue(self.glWidget.camera.camFocus[0])
                self.ui.camYLookAt.setValue(self.glWidget.camera.camFocus[1])
                self.ui.camZLookAt.setValue(self.glWidget.camera.camFocus[2])
                self.ui.camXUp.setValue(self.glWidget.camera.camUp[0])
                self.ui.camYUp.setValue(self.glWidget.camera.camUp[1])
                self.ui.camZUp.setValue(self.glWidget.camera.camUp[2])
                self.ui.camQuatW.setValue(self.glWidget.camera.cameraQuat.w)
                self.ui.camQuatX.setValue(self.glWidget.camera.cameraQuat.x)
                self.ui.camQuatY.setValue(self.glWidget.camera.cameraQuat.y)
                self.ui.camQuatZ.setValue(self.glWidget.camera.cameraQuat.z)

        self.ui.camFunction.setText('self.glWidget.camera.setView(quat(%0.10f,%0.10f,%
0.10f,%0.10f),[%0.10f,%0.10f,%0.10f],[%0.10f,%0.10f,%0.10f])' %
(self.glWidget.camera.cameraQuat.w,self.glWidget.camera.cameraQuat.x,self.glWidget.ca
```

```
mera.cameraQuat.y,self.glWidget.camera.cameraQuat.z,self.glWidget.camera.camLoc[0],se
lf.glWidget.camera.camLoc[1],self.glWidget.camera.camLoc[2],self.glWidget.camera.camF
ocus[0],self.glWidget.camera.camFocus[1],self.glWidget.camera.camFocus[2]))

	def setJointName(self):
		if self.selectedJoint is not None:
			self.selectedJoint.name = self.ui.jointName.text()
			self.populateItemTree(self.glWidget.worldJoint)

	def setModelName(self):
		if self.selectedModel is not None:
			self.selectedModel.name = self.ui.modelName.text()
			self.populateItemTree(self.glWidget.worldJoint)

	def invertNormals(self):
		self.selectedModel.invertNormals()
		self.glWidget.updateGL()

	def saveModel(self):
		if self.selectedModel is not None:
			filePath, filtername =
QtGui.QFileDialog.getSaveFileName(caption="Save Model as...", filter="STL ASCII file
(*.stl);;STL Binary file (*.stl)")
			if filtername == "STL Binary file (*.stl)":
				saveSTLFile(self.selectedModel, filePath)
			elif filtername == "STL ASCII file (*.stl)":
				saveSTLFile(self.selectedModel, filePath,False)

	def grabScreen(self):
#		GL.glClearColor(1.0,1.0,1.0,1.0)
#		self.glWidget.updateGL()
#		time.sleep(0.5)

		pic = self.glWidget.grabFrameBuffer()
		QtGui.QClipboard().setImage(pic)

#		pic = self.glWidget.renderPixmap()
#		QtGui.QClipboard().setPixmap(pic)

#		GL.glClearColor(0.0,0.0,0.0,1.0)
#		self.glWidget.updateGL()

	def automate(self):
		savePrefix=None
		savePrefix='1-10_'
		start = time.time()

		#load data
		t1=time.time()
		self.atlasData = UserAtlas('atlases\MOBY_package\MOBY Atlas.xml',
self.glWidget.worldJoint)
		print 'Time to Open Atlas:',time.time()-t1
		t1=time.time()
#		self.scanData = UserDicom(joint=self.glWidget.worldJoint)
```

```
#             self.scanData = UserDicom('dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-1-10_CT.dcm', 'dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-1-10_SPECT.dcm',
#                                                   joint=self.glWidget.worldJoint,
sigma=2, resampleFactor=3, isolevel=350.0)

#             self.scanData = UserDicom('dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-2-01_CT.dcm', 'dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-2-01_SPECT.dcm',
#                                                   joint=self.glWidget.worldJoint,
sigma=2, resampleFactor=3, isolevel=350.0)

#             self.scanData = UserDicom('dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-2-03_CT.dcm', 'dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-2-03_SPECT.dcm',
#                                                   joint=self.glWidget.worldJoint,
sigma=2, resampleFactor=3, isolevel=350.0)

#             self.scanData = UserDicom('dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-3-00_CT.dcm', 'dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-3-00_SPECT.dcm',
#                                                   joint=self.glWidget.worldJoint,
sigma=2, resampleFactor=3, isolevel=350.0)

#             self.scanData = UserDicom('dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-8-00_CT.dcm', 'dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-8-00_SPECT.dcm',
#                                                   joint=self.glWidget.worldJoint,
sigma=2, resampleFactor=3, isolevel=350.0)

              self.scanData = UserDicom('dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-1-10_CT.dcm', 'dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-1-10_SPECT.dcm',
                                                    joint=self.glWidget.worldJoint,
sigma=2, resampleFactor=3, isolevel=350.0,

       isosurfaceModelShortcut='dicom\Myeloma MDP
Scans\Trial\JVO_MyelomaMDPTrial_DM533-1-10_CT.stl',

       subsection1Shortcut='Isosurface1.stl',

       subsection2Shortcut='Isosurface2.stl',

       subsection3Shortcut='Isosurface3.stl')

              print 'Time to Open Dicom files:',time.time()-t1

              self.scanData.isosurfaceModel.visible = False
              self.setModelAlpha(0.7*255, self.scanData.isosurfaceModel1)
              self.setModelAlpha(0.7*255, self.scanData.isosurfaceModel2)
              self.setModelAlpha(0.7*255, self.scanData.isosurfaceModel3)

#             #draw PCA axis of atlas and model
#             axisLen = 200.0
#             bb = numpy.array(self.scanData.isosurfaceJoint.getBoundingBox())
```

```
#             mPCA1j = Joint(name='Model PCA 1', parentJoint=self.glWidget.worldJoint)
#             mPCA2j = Joint(name='Model PCA 2', parentJoint=self.glWidget.worldJoint)
#             mPCA3j = Joint(name='Model PCA 3', parentJoint=self.glWidget.worldJoint)
#             mPCA1j.translate(bb.mean(axis=0), absolute=False)
#             mPCA2j.translate(bb.mean(axis=0), absolute=False)
#             mPCA3j.translate(bb.mean(axis=0), absolute=False)
#
#             TriModel.createCylinder(axisLen,5, [0,0,-axisLen/2],
joint=mPCA1j,color=[1.0,0.0,0.0]).invertNormals()
#             axis = numpy.cross([0.0,0.0,1.0],
self.scanData.isosurfaceModel.PCA()[0])
#             angle = numpy.arccos(numpy.dot([0.0,0.0,1.0],
self.scanData.isosurfaceModel.PCA()[0]))
#             mPCA1j.rotate(angle, axis,relative=True)
#
#             TriModel.createCylinder(axisLen,5, [0,0,-axisLen/2],
joint=mPCA2j,color=[0.0,1.0,0.0]).invertNormals()
#             axis = numpy.cross([0.0,0.0,1.0],
self.scanData.isosurfaceModel.PCA()[1])
#             angle = numpy.arccos(numpy.dot([0.0,0.0,1.0],
self.scanData.isosurfaceModel.PCA()[1]))
#             mPCA2j.rotate(angle, axis,relative=True)
#
#             TriModel.createCylinder(axisLen,5, [0,0,-axisLen/2],
joint=mPCA3j,color=[0.0,0.0,1.0]).invertNormals()
#             axis = numpy.cross([0.0,0.0,1.0],
self.scanData.isosurfaceModel.PCA()[2])
#             angle = numpy.arccos(numpy.dot([0.0,0.0,1.0],
self.scanData.isosurfaceModel.PCA()[2]))
#             mPCA3j.rotate(angle, axis,relative=True)
#
#             bb = numpy.array(self.atlasData.atlasJoint.getBoundingBox())
#             aPCA1j = Joint(name='Atlas PCA 1', parentJoint=self.glWidget.worldJoint)
#             aPCA2j = Joint(name='Atlas PCA 2', parentJoint=self.glWidget.worldJoint)
#             aPCA3j = Joint(name='Atlas PCA 3', parentJoint=self.glWidget.worldJoint)
#             aPCA1j.translate(bb.mean(axis=0), absolute=False)
#             aPCA2j.translate(bb.mean(axis=0), absolute=False)
#             aPCA3j.translate(bb.mean(axis=0), absolute=False)
#
#             TriModel.createCylinder(axisLen,5, [0,0,-axisLen/2],
joint=aPCA1j,color=[1.0,0.0,0.0]).invertNormals()
#             axis = numpy.cross([0.0,0.0,1.0], self.atlasData.atlasAxes[0])
#             angle = numpy.arccos(numpy.dot([0.0,0.0,1.0],
self.atlasData.atlasAxes[0]))
##             aPCA1j.rotate(angle, axis,relative=True)
#
#             TriModel.createCylinder(axisLen,5, [0,0,-axisLen/2],
joint=aPCA2j,color=[0.0,1.0,0.0]).invertNormals()
#             axis = numpy.cross([0.0,0.0,1.0], self.atlasData.atlasAxes[1])
#             angle = numpy.arccos(numpy.dot([0.0,0.0,1.0],
self.atlasData.atlasAxes[1]))
#             aPCA2j.rotate(angle, axis,relative=True)
#
#             TriModel.createCylinder(axisLen,5, [0,0,-axisLen/2],
joint=aPCA3j,color=[0.0,0.0,1.0]).invertNormals()
```

```
#             axis = numpy.cross([0.0,0.0,1.0], self.atlasData.atlasAxes[2])
#             angle = numpy.arccos(numpy.dot([0.0,0.0,1.0],
self.atlasData.atlasAxes[2]))
#             aPCA3j.rotate(angle, axis,relative=True)

              #set camera and redraw scene
              self.glWidget.camera.setView(quat(-0.1450718126,-0.4014994641,-
0.1638632680,0.8893262500),[-681.1013854165,-
353.9837864636,1403.4630773443],[196.4125000000,183.0531656115,582.1479125977])
              self.setupSceneControls()
              self.glWidget.resizeGL(self.glWidget.width(), self.glWidget.height())
              self.glWidget.updateGL()

              #set up slice view controls
              self.setSlice100Axis(0)
              self.setSlice010Axis(0)
              self.setSlice001Axis(0)
              self.ui.spinBox001AxisSlice.setMaximum(self.scanData.ctField.shape[2])
              self.ui.slider001AxisSlice.setMaximum(self.scanData.ctField.shape[2])
              self.ui.spinBox010AxisSlice.setMaximum(self.scanData.ctField.shape[1])
              self.ui.slider010AxisSlice.setMaximum(self.scanData.ctField.shape[1])
              self.ui.spinBox100AxisSlice.setMaximum(self.scanData.ctField.shape[0])
              self.ui.slider100AxisSlice.setMaximum(self.scanData.ctField.shape[0])

              #Rough Align Atlas to skeleton
#             alignmentAxis, neckPos, scaleFactor, hipLoc, spineModel, tailModel,
spinePCA = AlignAtlas.roughAlign(self.scanData,self.atlasData, savePrefix=savePrefix,
verbose=True, visualize=True)
#             alignmentAxis, neckPos, scaleFactor, hipLoc, spineModel, tailModel,
spinePCA = AlignAtlas.roughAlign(self.scanData,self.atlasData, savePrefix=savePrefix,
verbose=True, visualize=True, pcaAxis=spinePCA)

              alignmentAxis, neckPos, scaleFactor, hipLoc, spineModel, tailModel,
spinePCA = AlignAtlas.roughAlign(self.scanData,self.atlasData ,verbose=True,
visualize=False,

       spineModelShortcut=savePrefix+'SpineModel.stl',
tailModelShortcut=savePrefix+'TailModel.stl')

              print 'Rough align Axis:'
              print alignmentAxis

              #add spine and tail models to scene
              #TODO: spine and tail models should be a replacement for the spine in
the atlas, find a way to add it to the atlas
              spineModel.setJoint(self.glWidget.worldJoint)
              tailModel.setJoint(self.glWidget.worldJoint)

              #roughAlign() returns rotation array of vectors [anteriorVector,
dorsalVector, rightVector]
              #generate a rotation matrix that will transform the atlas Axes to the CT
axes
              rotationMat =
numpyTransform.coordinateSystemConversionMatrix(self.atlasData.atlasAxes,
alignmentAxis, N=4)
```

```
            print 'Function alignment Axis:'
            print rotationMat

            for childJoint in self.atlasData.atlasJoint.childJoints:
                  if childJoint.name == 'Neck':
                        break
            offset = neckPos-childJoint.locationUnityScale
            offset = numpy.squeeze(numpy.array(offset))
            print 'Offset:',offset
            print 'Atlas Neck Position:',childJoint.locationUnityScale
            print 'Dicom Neck Position:',neckPos

#           #manually adjust scale factor
#           scaleFactor *= 1.35

            self.atlasData.atlasJoint.scale(scaleFactor)
            self.atlasData.atlasJoint.rotate(rotationMat)
            self.atlasData.atlasJoint.translate(offset, absolute=False)

            print 'Scale Matrix:'
            print self.atlasData.atlasJoint.scaleMat
            print 'Rotation Matrix:'
            print self.atlasData.atlasJoint.rotateMat
            print 'Offset Matrix:'
            print self.atlasData.atlasJoint.translateMat
            print 'Initial Offset Matrix'
            print self.atlasData.atlasJoint.initialLocationMat

            #hide not registered atlas bones
            self.atlasData.atlasModels['Spine'].visible = False
            self.atlasData.atlasModels['Scapula Right'].visible = False
            self.atlasData.atlasModels['Scapula Left'].visible = False
            self.atlasData.atlasModels['Upper Forelimb Right'].visible = False
            self.atlasData.atlasModels['Upper Forelimb Left'].visible = False
            self.atlasData.atlasModels['Lower Forelimb Right'].visible = False
            self.atlasData.atlasModels['Lower Forelimb Left'].visible = False
            self.atlasData.atlasModels['Forepaw Right'].visible = False
            self.atlasData.atlasModels['Forepaw Left'].visible = False

            #set camera and redraw scene
            self.glWidget.camera.setView(quat(0.3381642550,0.9222312229,-
0.0475532999,-0.1813096573),[79.9461811889,-88.6397062133,-
115.5570538501],[188.3407945769,90.6932903010,93.0659155950])
            self.setupSceneControls()
            self.glWidget.resizeGL(self.glWidget.width(), self.glWidget.height())
            self.glWidget.updateGL()

            #manually create kdtree for model
            self.scanData.isosurfaceModel1.kdtree=
cKDTree(self.scanData.isosurfaceModel1.transformedVertexList*self.scanData.resampleFa
ctor)

#           #Get data for paper
#           self.atlasData.atlasJoint.transformVertices()
#           dist = 0.0
```

```
#            numPoints = 0.0
#            for bone in self.scanData.bonesVol1:
#                  print 'Post Rough Align %s Joint Location %s' %
(self.atlasData.atlasModels[bone].joint.name,
self.atlasData.atlasModels[bone].joint.location)
#                  d, i =
self.scanData.isosurfaceModel1.kdtree.query(self.atlasData.atlasModels[bone].transfor
medVertexList)
#                  dist += numpy.sum(d)
#                  numPoints += len(i)
#                  print 'Post Rough Align Mean distance from %s to surface: %f' %
(self.atlasData.atlasModels[bone].name, numpy.sum(d)/len(i))
#            print 'Post Rough Align Mean distance all bones to surface: %f' %
(dist/numPoints)


#            #Create reference object, this is useful for checking position and
orientation of other models
#            scale=1.0
#            size=scale*numpy.array([1.0,1.0,1.0])
#            offset2 = scale*numpy.array([-0.5,-0.5,-0.5])
#            refPointJoint1=Joint(parentJoint=self.glWidget.worldJoint, name='Ref
Point 1 Joint')
#            TriModel.createRectangularSolid(size,offset2, joint=refPointJoint1,
name='Ref Point',color=[1.0,0.0,0.0,1.0])
#
#            #Create reference object
#            scale=1.0
#            size=scale*numpy.array([1.0,1.0,1.0])
#            offset2 = scale*numpy.array([-0.5,-0.5,-0.5])
#            refPointJoint2=Joint(parentJoint=self.glWidget.worldJoint, name='Ref
Point 2 Joint')
#            TriModel.createRectangularSolid(size,offset2, joint=refPointJoint2,
name='Ref Point',color=[0.0,1.0,0.0,1.0])

            #Do Fine align
#            AlignAtlas.FineAlign(self.scanData, self.atlasData, hipLocation =
hipLoc, spineVertices=spineModel.OriginalVertexList,
#
      tailVertices=tailModel.OriginalVertexList,savePrefix=savePrefix,getResults=Fal
se)

            AlignAtlas.FineAlign(self.scanData, self.atlasData, hipLocation =
hipLoc, spineVertices=spineModel.OriginalVertexList,

      tailVertices=tailModel.OriginalVertexList,
spineVertexShortcut=savePrefix+'spineIndices.mat',

      skullVertexShortcut=savePrefix+'skullvertices.mat',
hipLeftVertexShortcut=savePrefix+'hipLeftvertices.mat',

      hipRightVertexShortcut=savePrefix+'hipRightvertices.mat',
femurLeftVertexShortcut=savePrefix+'femurLeftvertices.mat',
```

```
	femurRightVertexShortcut=savePrefix+'femurRightvertices.mat',
tibiaLeftVertexShortcut=savePrefix+'tibiaLeftvertices.mat',

	tibiaRightVertexShortcut=savePrefix+'tibiaRightvertices.mat',
lowerLeftPawVertexShortcut=savePrefix+'lowerLeftPawvertices.mat',

	lowerRightPawVertexShortcut=savePrefix+'lowerRightPawvertices.mat',fineAlignTr
ansformsShortcut=savePrefix+'FineAlignTransforms.mat',
                                        getResults=False, updateSceneFunc =
self.glWidget.updateGL)

            #Create NM data sized mask of CT data
            boneVertexLists=[]
            self.atlasData.atlasJoint.transformVertices()
            skullVlist = numpy.append(self.atlasData.atlasModels['Skull
Inside'].transformedVertexList, self.atlasData.atlasModels['Skull
Outside'].transformedVertexList)
            skullVlist = skullVlist.reshape((-1,3))
            boneVertexLists.append(skullVlist)
            boneVertexLists.append(self.atlasData.atlasModels['Pelvis
Right'].transformedVertexList)
            boneVertexLists.append(self.atlasData.atlasModels['Pelvis
Left'].transformedVertexList)
            boneVertexLists.append(self.atlasData.atlasModels['Upper Hindlimb
Right'].transformedVertexList)
            boneVertexLists.append(self.atlasData.atlasModels['Upper Hindlimb
Left'].transformedVertexList)
            boneVertexLists.append(self.atlasData.atlasModels['Lower Hindlimb
Right'].transformedVertexList)
            boneVertexLists.append(self.atlasData.atlasModels['Lower Hindlimb
Left'].transformedVertexList)
            boneVertexLists.append(self.atlasData.atlasModels['HindPaw
Right'].transformedVertexList)
            boneVertexLists.append(self.atlasData.atlasModels['HindPaw
Left'].transformedVertexList)
            boneVertexLists.append(spineModel.transformedVertexList)
            boneVertexLists.append(tailModel.transformedVertexList)
            #the problem is that the scale and origins are off. atlas data must be
scaled to match mesh units, also mask[0,0,0] is not the same as the atlas origin,
this is because the mask volume had to be changed so that its size was the same as
the NM data
            for i in xrange(len(boneVertexLists)):
                  vlist = boneVertexLists[i]
                  vlist *= self.scanData.ctDCM.SliceThickness /
self.scanData.nmDCM.SliceThickness     #convert vertex points in CT units to NM
units
                  vlist -=
self.scanData.ctVolumeNMmaskOriginLocation/self.scanData.nmDCM.SliceThickness
      #subtract offset that has been scaled from mm to NM units
                  boneVertexLists[i] = vlist
            mask = self.scanData.createNM_Mask(boneVertexLists)

            if savePrefix is not None:
                  savemat(savePrefix+'Mask.mat', {'Mask':mask})
```

```
            import visvis as vv
            app = vv.use()
            vv.figure()
            vv.volshow(mask)
            app.Run()

            boneNames = ['Skull', 'Pelvis Right', 'Pelvis Left', 'Femur Right',
'Femur Left', 'Tibia Right', 'Tibia Left', 'Hindpaw Right', 'Hindpaw
Left','Spine','Tail']
            self.populateNMTable(self.scanData.nmField, mask, boneNames)

#           #create texture map
#           t1 = time.time()
#           textureVolume =
numpy.zeros((labelVolume.shape[0],labelVolume.shape[1],labelVolume.shape[2],3),
dtype=numpy.ubyte)
#           colormap = [ [0xFF, 0x00, 0x00],
#                                   [0xFF, 0x80, 0x00],
#                                   [0xFF, 0xFF, 0x00],
#                                   [0x80, 0xFF, 0x00],
#                                   [0x00, 0xFF, 0x00],
#                                   [0x00, 0xFF, 0x80],
#                                   [0x00, 0xFF, 0xFF],
#                                   [0x00, 0x80, 0xFF],
#                                   [0x00, 0x00, 0xFF],
#                                   [0x7F, 0x00, 0xFF],
#                                   [0xFF, 0x00, 0xFF],
#                                   [0xFF, 0x00, 0x7F],
#                                   [0x80, 0x80, 0x80],
#                                   [0xFF, 0x66, 0x66],
#                                   [0xFF, 0xB2, 0x66],
#                                   [0xFF, 0xFF, 0x66],
#                                   [0xB2, 0xFF, 0x66],
#                                   [0x66, 0xFF, 0x66],
#                                   [0x66, 0xFF, 0xB2],
#                                   [0x66, 0xFF, 0xFF],
#                                   [0x66, 0xB2, 0xFF],
#                                   [0x66, 0x66, 0xFF],
#                                   [0xB2, 0x66, 0xFF],
#                                   [0xFF, 0x66, 0xFF],
#                                   [0xFF, 0x66, 0xB2],
#                                   [0xC0, 0xC0, 0xC0],
#                                   [0xFF, 0xFF, 0xFF]]
#           for i in xrange(uniqueValues.shape[0]):
#                 if i < len(colormap):
#                       textureVolume[labelVolume==uniqueValues[i]]=i
#           self.scanData.createLabelVolumeTextures(textureVolume)
#           self.scanData.setTextureToLabeledVolume()
#           print 'Time to create textures from labeled volume and apply them:',
time.time()-t1

            elapsedTime = time.time()-start
            print 'Time to Automate: %02d:%02d:%06.3f' %
(int(elapsedTime/3600),(int(elapsedTime)%3600)/60,elapsedTime%60)
```

```
            #redraw gui stuff
            self.setupSceneControls()
#           self.setView()
#           self.glWidget.resizeGL(self.glWidget.width(), self.glWidget.height())
            self.glWidget.updateGL()
#           self.close()

      def populateNMTable(self, nmField, labelVolume, boneNames):
            '''
            The index of the bone name + 1 is the label number for that bone in the
labelVolume
            ex. if boneNames[2] = 'Tibia', then the label for tibia in labelVolume
is 3.
            '''
            #clear table
            self.ui.nmTable.clearContents()

            numberOfBones = len(boneNames)
            if numberOfBones <= 0:
                  return
            #get proper number of rows
            while self.ui.nmTable.rowCount() > numberOfBones:
                  self.ui.nmTable.removeRow(self.ui.nmTable.rowCount())
            while self.ui.nmTable.rowCount() < numberOfBones:
                  self.ui.nmTable.insertRow(self.ui.nmTable.rowCount())

            columnLabels = ['Region Size (mm^3)','Sum (MBq)', 'Average (MBq/mm^3)',
'Minimum (MBq)', '1st Quartile (MBq)', 'Median (MBq)', '3rd Quartile (MBq)', 'Maximum
(MBq)','Standard Deviation (MBq)','Variance (MBq)']
            #set up columns
            while self.ui.nmTable.columnCount() > len(columnLabels):
                  self.ui.nmTable.removeColumn(self.ui.nmTable.columnCount())
            while self.ui.nmTable.columnCount() < len(columnLabels):
                  self.ui.nmTable.insertColumn(self.ui.nmTable.columnCount())
            self.ui.nmTable.setHorizontalHeaderLabels(columnLabels)
            self.ui.nmTable.setVerticalHeaderLabels(boneNames)

            #set contents of table
            for i in xrange(len(boneNames)):
                  uptakeData =
nmField[labelVolume==i+1]/self.scanData.nmDCM[0x6001,0x10a1].value/1000.0      #get
scan data in MBq
                  regionSize =
len(uptakeData)*self.scanData.nmDCM.SliceThickness**3      #get volume size in cubic
mm
                  if len(uptakeData) == 0:
                        uptakeData = numpy.zeros(1)
                  self.ui.nmTable.setItem(i, 0, QtGui.QTableWidgetItem(str(
regionSize )))
                  self.ui.nmTable.setItem(i, 1, QtGui.QTableWidgetItem(str(
numpy.sum(uptakeData) )))
                  if regionSize == 0:
                        self.ui.nmTable.setItem(i, 2,
QtGui.QTableWidgetItem('0.0'))
```

```
                        else:
                                self.ui.nmTable.setItem(i, 2, QtGui.QTableWidgetItem(str(
numpy.sum(uptakeData)/regionSize )))
                        self.ui.nmTable.setItem(i, 3, QtGui.QTableWidgetItem(str(
numpy.min(uptakeData) )))
                        self.ui.nmTable.setItem(i, 4, QtGui.QTableWidgetItem(str(
scipy.stats.scoreatpercentile(uptakeData, 25) )))
                        self.ui.nmTable.setItem(i, 5, QtGui.QTableWidgetItem(str(
numpy.median(uptakeData) )))
                        self.ui.nmTable.setItem(i, 6, QtGui.QTableWidgetItem(str(
scipy.stats.scoreatpercentile(uptakeData, 75) )))
                        self.ui.nmTable.setItem(i, 7, QtGui.QTableWidgetItem(str(
numpy.max(uptakeData) )))
                        self.ui.nmTable.setItem(i, 8, QtGui.QTableWidgetItem(str(
numpy.std(uptakeData) )))
                        self.ui.nmTable.setItem(i, 9, QtGui.QTableWidgetItem(str(
numpy.var(uptakeData) )))

                #resize table
                self.ui.nmTable.resizeColumnsToContents()

        def loadModel(self,filepath=None,**kwargs):
                #TODO: split this function up
                '''
                set stlShortcutFilepath to a stl path so that loading dicom will skip
actually polygonizing volume field
                '''
                if filepath is None:
                        filepaths, flter =
QtGui.QFileDialog.getOpenFileNames(caption="Select a model to load", filter="All
supported Files (*.stl *.binvox *.dcm);;STL Files (*.stl);;Binvox (*.binvox);;DICOM
(*.dcm);;Any File (*.*)") #@UnusedVariable
                        for filepath in filepaths:
                                self.loadModel(filepath)
                        return
                if len(filepath) == 0:
                        return
                model = None
                jName = 'Model Base'
                fileName, fileExtension = os.path.splitext(filepath) #@UnusedVariable
                if fileExtension == '.stl':
                        model = readSTLfile(filepath,True,**kwargs)
                        jName = 'Model Base'
                elif fileExtension == '.binvox':
                        bv = BinVox(filepath,True)
#                       model = bv.createVoxelModel()
                        mesh = Polygonise(bv.data, 0.5)
                        model = mesh.isosurface(**kwargs)
                        jName = 'BinVox Base'
                elif fileExtension == '.dcm':
                        dcm = dicom.read_file(filepath)
                        field = numpy.array(dcm.pixel_array, dtype=numpy.float)
                        field = field * dcm.RescaleSlope + dcm.RescaleIntercept

                        if 'sigma' in kwargs:
```

```
                        sigma = kwargs['sigma']
                else:
                        sigma, ok = QtGui.QInputDialog.getInt(self, 'Gaussian
Filter Sigma','Enter Sigma for Gaussian Filter, 0 means no filter',
maxValue=10,minValue=0,value=2)
                        if not ok:
                                sigma = 2
                if sigma != 0:
                        field = gaussian_filter(field, sigma)

                if 'resampleFactor' in kwargs:
                        resampleFactor = kwargs['resampleFactor']
                else:
                        resampleFactor, ok = QtGui.QInputDialog.getInt(None,
'Rescale',"Enter Rescale factor. 1 means don't resample, 2 means 1/2 points, 3 means
1/3 points,...", value=3, minValue=1, maxValue=min(field.shape))
                        if not ok:
                                resampleFactor=3
                if resampleFactor>1:
                        from scipy import ndimage
                        #resample at half the original size so as to make
polygonising much faster
                        #TODO: ensure the math on this for values other than 2
                        coord =
numpy.mgrid[0:len(field):resampleFactor,0:len(field[0]):resampleFactor,0:len(field[0,
0]):resampleFactor]
                        field = ndimage.map_coordinates(field, coord)
                if 'stlShortcutFilepath' in kwargs and
os.path.exists(kwargs['stlShortcutFilepath']):
                        model = self.loadModel(kwargs['stlShortcutFilepath'])
                        model.referenceVolume = field
                else:
                        #TODO ask for isolevel
                        mesh = Polygonise(field, 350.0)
                        pd =  QtGui.QProgressDialog("Polygonising DICOM file",
"Cancel", 0, 100)
                        pd.setCancelButton(None)
                        model =
mesh.isosurface(progressDialogCallback=pd.setValue,**kwargs)
                        model.invertNormals()
                        pd.setValue(100)
                jName = 'DICOM Base'

        if model is not None:
                self.referenceModelJoint =
Joint(name=jName,parentJoint=self.glWidget.worldJoint)
                model.setJoint(self.referenceModelJoint)
                self.setupSceneControls()
                self.glWidget.resizeGL(self.glWidget.width(),
self.glWidget.height())
                self.glWidget.updateGL()

        return model

    def loadAtlas(self, filepath=None):
```

```
        UserAtlas(filepath, self.glWidget.worldJoint)
        #reset view
        self.glWidget.resizeGL(self.glWidget.width(), self.glWidget.height())
        self.glWidget.updateGL()
        self.setView()
        self.setupSceneControls()

    def setModelVisibility(self, state):
        if self.selectedModel is None:
            return
        if state == QtCore.Qt.CheckState.Checked:
            self.selectedModel.visible = True
        else:
            self.selectedModel.visible = False
        self.glWidget.updateGL()

    def setColorModel(self, state):
        self.ui.chooseAmbientColorMat.setEnabled(True)
        self.ui.matShinySlider.setEnabled(True)
        self.ui.matShinySpinBox.setEnabled(True)
        if state == QtCore.Qt.CheckState.Checked:
            self.glWidget.setColorDrivenMaterial(True)
            self.ui.chooseDiffuseColorMat.setEnabled(False)
            self.ui.chooseSpecularColorMat.setEnabled(False)
            self.ui.chooseEmissiveColorMat.setEnabled(False)
        else:
            self.glWidget.setColorDrivenMaterial(False)
            self.ui.chooseDiffuseColorMat.setEnabled(True)
            self.ui.chooseSpecularColorMat.setEnabled(True)
            self.ui.chooseEmissiveColorMat.setEnabled(True)

    def setRotateCoord(self, state):
        self.ui.jointXRotSlider.setValue(0)
        self.ui.jointXRotSpinBox.setValue(0.0)
        self.ui.jointYRotSlider.setValue(0)
        self.ui.jointYRotSpinBox.setValue(0.0)
        self.ui.jointZRotSlider.setValue(0)
        self.ui.jointZRotSpinBox.setValue(0.0)
        if state == QtCore.Qt.CheckState.Checked:
            self.ui.xAxisLabel.setText('Elevation')
            self.ui.yAxisLabel.setText('Azimuth')
            self.ui.zAxisLabel.setText('Spin')
            self.ui.rotationOrder.setCurrentIndex(0)
            self.ui.rotationOrder.setEnabled(False)
            self.ui.jointXRotSlider.setMaximum(90)
            self.ui.jointXRotSlider.setMinimum(-90)
            self.ui.jointXRotSpinBox.setMaximum(90.0)
            self.ui.jointXRotSpinBox.setMinimum(-90.0)
        else:
            self.ui.xAxisLabel.setText('X')
            self.ui.yAxisLabel.setText('Y')
            self.ui.zAxisLabel.setText('Z')
            self.ui.rotationOrder.setCurrentIndex(0)
            self.ui.rotationOrder.setEnabled(True)
            self.ui.jointXRotSlider.setMaximum(360)
```

```
                    self.ui.jointXRotSlider.setMinimum(0)
                    self.ui.jointXRotSpinBox.setMaximum(360.0)
                    self.ui.jointXRotSpinBox.setMinimum(0.0)

        def rotationOrderChanged(self, order):
                self.rotateJointSpinBox()

        def rotateTolSlider(self, value=None):

        QtCore.QObject.disconnect(self.ui.tolXRotSpinBox,QtCore.SIGNAL("valueChanged(d
ouble)"), self.rotateTolSpinBox)

        QtCore.QObject.disconnect(self.ui.tolYRotSpinBox,QtCore.SIGNAL("valueChanged(d
ouble)"), self.rotateTolSpinBox)

        QtCore.QObject.disconnect(self.ui.tolZRotSpinBox,QtCore.SIGNAL("valueChanged(d
ouble)"), self.rotateTolSpinBox)
                x = self.ui.tolXRotSlider.value()
                y = self.ui.tolYRotSlider.value()
                z = self.ui.tolZRotSlider.value()
                self.ui.tolXRotSpinBox.setValue(x)
                self.ui.tolYRotSpinBox.setValue(y)
                self.ui.tolZRotSpinBox.setValue(z)

        QtCore.QObject.connect(self.ui.tolXRotSpinBox,QtCore.SIGNAL("valueChanged(doub
le)"), self.rotateTolSpinBox)

        QtCore.QObject.connect(self.ui.tolYRotSpinBox,QtCore.SIGNAL("valueChanged(doub
le)"), self.rotateTolSpinBox)

        QtCore.QObject.connect(self.ui.tolZRotSpinBox,QtCore.SIGNAL("valueChanged(doub
le)"), self.rotateTolSpinBox)

                self.updateTolVec()

        def rotateTolSpinBox(self, value=None):

        QtCore.QObject.disconnect(self.ui.tolXRotSlider,QtCore.SIGNAL("valueChanged(in
t)"), self.rotateTolSlider)

        QtCore.QObject.disconnect(self.ui.tolYRotSlider,QtCore.SIGNAL("valueChanged(in
t)"), self.rotateTolSlider)

        QtCore.QObject.disconnect(self.ui.tolZRotSlider,QtCore.SIGNAL("valueChanged(in
t)"), self.rotateTolSlider)
                x = self.ui.tolXRotSpinBox.value()
                y = self.ui.tolYRotSpinBox.value()
                z = self.ui.tolZRotSpinBox.value()
                self.ui.tolXRotSlider.setValue(x)
                self.ui.tolYRotSlider.setValue(y)
                self.ui.tolZRotSlider.setValue(z)

        QtCore.QObject.connect(self.ui.tolXRotSlider,QtCore.SIGNAL("valueChanged(int)"
), self.rotateTolSlider)
```

```
	QtCore.QObject.connect(self.ui.tolYRotSlider,QtCore.SIGNAL("valueChanged(int)"
), self.rotateTolSlider)

	QtCore.QObject.connect(self.ui.tolZRotSlider,QtCore.SIGNAL("valueChanged(int)"
), self.rotateTolSlider)

		self.updateTolVec()

	def updateTolVec(self):
		#TODO: fill in stuff
		rx = math.radians(self.ui.tolXRotSpinBox.value())
		ry = math.radians(self.ui.tolYRotSpinBox.value())
		rz = math.radians(self.ui.tolZRotSpinBox.value())
		l = self.ui.tolVecLength.value()
		x = self.ui.tolVecXOrigin.value()
		y = self.ui.tolVecYOrigin.value()
		z = self.ui.tolVecZOrigin.value()

		if self.tolJoint is None:
			self.makeTolVec()

		R = numpyTransform.rotation(rz, [0,0,1], N=4) *
numpyTransform.rotation(ry, [0,1,0], N=4) * numpyTransform.rotation(rx, [1,0,0], N=4)
		vec = l * numpy.array([0.0,0.0,1.0])
		vec = numpyTransform.transformPoints(R, vec[numpy.newaxis,:]).squeeze()
		self.ui.tolVecX.setValue(vec[0])
		self.ui.tolVecY.setValue(vec[1])
		self.ui.tolVecZ.setValue(vec[2])

		self.tolJoint.translate([x,y,z], absolute=True)
		self.tolJoint.rotate(R, relative=False)
		self.tolJoint.scale(l)

		#reset view
#		self.glWidget.resizeGL(self.glWidget.width(), self.glWidget.height())
		self.glWidget.updateGL()
#		self.setView()
#		self.setupSceneControls()

	def makeTolVec(self):
		a = math.radians(self.ui.tolVecAngle.value())
		if self.tolJoint is None:
			self.tolJoint = Joint([0.0, 0.0, 0.0],
parentJoint=self.glWidget.worldJoint, name='Tolerance Joint')
		model = TriModel.createTolRegion(a, name='Tolerance Region',
color=[1,0,0])
		if len(self.tolJoint.models) > 3:
			del self.tolJoint.models[-1]
		model.setJoint(self.tolJoint)

		#reset view
		self.glWidget.resizeGL(self.glWidget.width(), self.glWidget.height())
		self.glWidget.updateGL()
		self.setView()
```

```
            self.setupSceneControls()

        def rotateJointSlider(self, value=None):
            x = self.ui.jointXRotSlider.value()
            y = self.ui.jointYRotSlider.value()
            z = self.ui.jointZRotSlider.value()
            self.rotateJoint(x,y,z)

        def rotateJointSpinBox(self, value=None):
            x = self.ui.jointXRotSpinBox.value()
            y = self.ui.jointYRotSpinBox.value()
            z = self.ui.jointZRotSpinBox.value()
            self.rotateJoint(x,y,z)

        def rotateJoint(self,x,y,z):
            if self.selectedJoint is not None:
                if self.ui.sphericalCoord.checkState() ==
QtCore.Qt.CheckState.Checked:
                    sc = True
                else:
                    sc = False
#                   start = time.time()
                    if self.glWidget.useCallLists:
                        self.selectedJoint.rotate(x, y, z, sphericalCoord=sc,
relative=False, unitsDegrees=True, angleOrder=self.ui.rotationOrder.currentText(),
updateModels=False)
                    else:
                        self.selectedJoint.rotate(x, y, z, sphericalCoord=sc,
relative=False, unitsDegrees=True, angleOrder=self.ui.rotationOrder.currentText())
#                   print 'Time to rotate: %f' % (time.time()-start)
                    self.updateJointPropertiesTab()
#                   start = time.time()
                    self.glWidget.updateGL()
#                   print 'Time to update OpenGL: %f' % (time.time()-start)

        def moveJointX(self,deltaX=None,joint=None):
            if deltaX is None:
                deltaX=self.ui.jointXPosStepSize.value()
            if joint is None:
                joint = self.selectedJoint
            if joint is not None:
                joint.translate([deltaX,0.0,0.0],False)
                self.glWidget.updateGL()
                self.updateJointPropertiesTab()

        def moveJointY(self,deltaY=None,joint=None):
            if deltaY is None:
                deltaY=self.ui.jointYPosStepSize.value()
            if joint is None:
                joint = self.selectedJoint
            if joint is not None:
                joint.translate([0.0,deltaY,0.0],False)
                self.glWidget.updateGL()
                self.updateJointPropertiesTab()
```

```
	def moveJointZ(self,deltaZ=None,joint=None):
		if deltaZ is None:
			deltaZ=self.ui.jointZPosStepSize.value()
		if joint is None:
			joint = self.selectedJoint
		if joint is not None:
			joint.translate([0.0,0.0,deltaZ],False)
			self.glWidget.updateGL()
			self.updateJointPropertiesTab()

	def itemChangedCB(self):
		if self.ui.itemTreeList.currentItem() is not None:
			obj = self.ui.itemTreeList.currentItem().obj
			if isinstance(obj, Joint):
				self.selectedJoint = obj
				self.ui.jointXRotSlider.setEnabled(True)
				self.ui.jointYRotSlider.setEnabled(True)
				self.ui.jointZRotSlider.setEnabled(True)
				self.ui.jointXRotSpinBox.setEnabled(True)
				self.ui.jointYRotSpinBox.setEnabled(True)
				self.ui.jointZRotSpinBox.setEnabled(True)
				if self.ui.sphericalCoord.checkState() ==
QtCore.Qt.CheckState.Unchecked:
					self.ui.rotationOrder.setEnabled(True)
				self.updateJointPropertiesTab()
			elif isinstance(obj, TriModel.TriModel):
				self.selectedModel = obj
				self.ui.modelVisible.setEnabled(True)
				self.updateModelPropertiesTab()

	def setSlice100Axis(self, index):
		if self.scanData is not None and self.scanData.slice100Joint is not
None:  #ensure there is a joint to move
			#disconnect signals before changing them so as not to cause
recursion

	QtCore.QObject.disconnect(self.ui.spinBox100AxisSlice,QtCore.SIGNAL("valueChan
ged(int)"), self.setSlice100Axis)

	QtCore.QObject.disconnect(self.ui.slider100AxisSlice,QtCore.SIGNAL("valueChang
ed(int)"), self.setSlice100Axis)
			self.ui.spinBox100AxisSlice.setValue(index)
			self.ui.slider100AxisSlice.setValue(index)

	QtCore.QObject.connect(self.ui.spinBox100AxisSlice,QtCore.SIGNAL("valueChanged
(int)"), self.setSlice100Axis)

	QtCore.QObject.connect(self.ui.slider100AxisSlice,QtCore.SIGNAL("valueChanged(
int)"), self.setSlice100Axis)
			self.scanData.slice100Joint.translate([index, 0, 0],
absolute=True)
			self.scanData.setSlice100Index(index)
			self.glWidget.updateGL()

	def setSlice010Axis(self, index):
```

```
            if self.scanData is not None and self.scanData.slice100Joint is not
None:  #ensure there is a joint to move
                  #disconnect signals before changing them so as not to cause
recursion

      QtCore.QObject.disconnect(self.ui.spinBox010AxisSlice,QtCore.SIGNAL("valueChan
ged(int)"), self.setSlice010Axis)

      QtCore.QObject.disconnect(self.ui.slider010AxisSlice,QtCore.SIGNAL("valueChang
ed(int)"), self.setSlice010Axis)
                  self.ui.spinBox010AxisSlice.setValue(index)
                  self.ui.slider010AxisSlice.setValue(index)

      QtCore.QObject.connect(self.ui.spinBox010AxisSlice,QtCore.SIGNAL("valueChanged
(int)"), self.setSlice010Axis)

      QtCore.QObject.connect(self.ui.slider010AxisSlice,QtCore.SIGNAL("valueChanged(
int)"), self.setSlice010Axis)
                  self.scanData.slice010Joint.translate([0, index, 0],
absolute=True)
                  self.scanData.setSlice010Index(index)
                  self.glWidget.updateGL()

      def setSlice001Axis(self, index):
            if self.scanData is not None and self.scanData.slice100Joint is not
None:  #ensure there is a joint to move
                  #disconnect signals before changing them so as not to cause
recursion

      QtCore.QObject.disconnect(self.ui.spinBox001AxisSlice,QtCore.SIGNAL("valueChan
ged(int)"), self.setSlice001Axis)

      QtCore.QObject.disconnect(self.ui.slider001AxisSlice,QtCore.SIGNAL("valueChang
ed(int)"), self.setSlice001Axis)
                  self.ui.spinBox001AxisSlice.setValue(index)
                  self.ui.slider001AxisSlice.setValue(index)

      QtCore.QObject.connect(self.ui.spinBox001AxisSlice,QtCore.SIGNAL("valueChanged
(int)"), self.setSlice001Axis)

      QtCore.QObject.connect(self.ui.slider001AxisSlice,QtCore.SIGNAL("valueChanged(
int)"), self.setSlice001Axis)
                  self.scanData.slice001Joint.translate([0, 0, index],
absolute=True)
                  self.scanData.setSlice001Index(index)
                  self.glWidget.updateGL()

      def updateJointPropertiesTab(self):
            #prevent callbacks from happening
            QtCore.QObject.disconnect(self.ui.jointScaleX,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
            QtCore.QObject.disconnect(self.ui.jointScaleY,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
            QtCore.QObject.disconnect(self.ui.jointScaleZ,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
```

```
            QtCore.QObject.disconnect(self.ui.jointName,
QtCore.SIGNAL("editingFinished()"), self.setJointName)
            self.ui.jointName.setText(self.selectedJoint.name)
            self.ui.jointScaleX.setValue(self.selectedJoint.scaleMat[0,0])
            self.ui.jointScaleY.setValue(self.selectedJoint.scaleMat[1,1])
            self.ui.jointScaleZ.setValue(self.selectedJoint.scaleMat[2,2])
            QtCore.QObject.connect(self.ui.jointScaleX,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
            QtCore.QObject.connect(self.ui.jointScaleY,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
            QtCore.QObject.connect(self.ui.jointScaleZ,
QtCore.SIGNAL("valueChanged(double)"), self.setJointScale)
            QtCore.QObject.connect(self.ui.jointName,
QtCore.SIGNAL("editingFinished()"), self.setJointName)

            #Be sure that setting the slider value does not cause recursion

      QtCore.QObject.disconnect(self.ui.jointXRotSlider,QtCore.SIGNAL("valueChanged(
int)"), self.rotateJointSlider)

      QtCore.QObject.disconnect(self.ui.jointYRotSlider,QtCore.SIGNAL("valueChanged(
int)"), self.rotateJointSlider)

      QtCore.QObject.disconnect(self.ui.jointZRotSlider,QtCore.SIGNAL("valueChanged(
int)"), self.rotateJointSlider)

      self.ui.jointXRotSlider.setValue(round(math.degrees(self.selectedJoint.xAngle)
))

      self.ui.jointYRotSlider.setValue(round(math.degrees(self.selectedJoint.yAngle)
))

      self.ui.jointZRotSlider.setValue(round(math.degrees(self.selectedJoint.zAngle)
))

      QtCore.QObject.connect(self.ui.jointXRotSlider,QtCore.SIGNAL("valueChanged(int
)"), self.rotateJointSlider)

      QtCore.QObject.connect(self.ui.jointYRotSlider,QtCore.SIGNAL("valueChanged(int
)"), self.rotateJointSlider)

      QtCore.QObject.connect(self.ui.jointZRotSlider,QtCore.SIGNAL("valueChanged(int
)"), self.rotateJointSlider)

            #Be sure that setting the spinbox value does not cause recursion

      QtCore.QObject.disconnect(self.ui.jointXRotSpinBox,QtCore.SIGNAL("valueChanged
(double)"), self.rotateJointSpinBox)

      QtCore.QObject.disconnect(self.ui.jointYRotSpinBox,QtCore.SIGNAL("valueChanged
(double)"), self.rotateJointSpinBox)

      QtCore.QObject.disconnect(self.ui.jointZRotSpinBox,QtCore.SIGNAL("valueChanged
(double)"), self.rotateJointSpinBox)
```

```
        self.ui.jointXRotSpinBox.setValue(float(math.degrees(self.selectedJoint.xAngle
)))

        self.ui.jointYRotSpinBox.setValue(float(math.degrees(self.selectedJoint.yAngle
)))

        self.ui.jointZRotSpinBox.setValue(float(math.degrees(self.selectedJoint.zAngle
)))

        QtCore.QObject.connect(self.ui.jointXRotSpinBox,QtCore.SIGNAL("valueChanged(do
uble)"), self.rotateJointSpinBox)

        QtCore.QObject.connect(self.ui.jointYRotSpinBox,QtCore.SIGNAL("valueChanged(do
uble)"), self.rotateJointSpinBox)

        QtCore.QObject.connect(self.ui.jointZRotSpinBox,QtCore.SIGNAL("valueChanged(do
uble)"), self.rotateJointSpinBox)

                self.ui.jointXPos.setValue(self.selectedJoint.location[0])
                self.ui.jointYPos.setValue(self.selectedJoint.location[1])
                self.ui.jointZPos.setValue(self.selectedJoint.location[2])
                self.ui.jointQuatW.setValue(self.selectedJoint.orientation.w)
                self.ui.jointQuatX.setValue(self.selectedJoint.orientation.x)
                self.ui.jointQuatY.setValue(self.selectedJoint.orientation.y)
                self.ui.jointQuatZ.setValue(self.selectedJoint.orientation.z)

        def updateModelPropertiesTab(self):
                #prevent callback from happening
                QtCore.QObject.disconnect(self.ui.modelName,
QtCore.SIGNAL("editingFinished()"), self.setModelName)
                self.ui.modelName.setText(self.selectedModel.name)
                QtCore.QObject.connect(self.ui.modelName,
QtCore.SIGNAL("editingFinished()"), self.setModelName)
                if self.selectedModel.visible:
                        self.ui.modelVisible.setCheckState(QtCore.Qt.CheckState.Checked)
                else:

        self.ui.modelVisible.setCheckState(QtCore.Qt.CheckState.Unchecked)
#               self.ui.modelScale.setValue(self.selectedModel.scale)
                r = int(self.selectedModel.ambientColor[0] * 255)
                g = int(self.selectedModel.ambientColor[1] * 255)
                b = int(self.selectedModel.ambientColor[2] * 255)
                a = int(self.selectedModel.ambientColor[3] * 255)
                self.setButtonIconToColor(self.ui.chooseAmbientColorMat,
QtGui.QColor.fromRgb(r, g, b, a))
                self.ui.modelAlpha.setValue(a*255)

                r = int(self.selectedModel.diffuseColor[0] * 255)
                g = int(self.selectedModel.diffuseColor[1] * 255)
                b = int(self.selectedModel.diffuseColor[2] * 255)
                a = int(self.selectedModel.diffuseColor[3] * 255)
                self.setButtonIconToColor(self.ui.chooseDiffuseColorMat,
QtGui.QColor.fromRgb(r, g, b, a))
```

```
            r = int(self.selectedModel.specularColor[0] * 255)
            g = int(self.selectedModel.specularColor[1] * 255)
            b = int(self.selectedModel.specularColor[2] * 255)
            a = int(self.selectedModel.specularColor[3] * 255)
            self.setButtonIconToColor(self.ui.chooseSpecularColorMat,
QtGui.QColor.fromRgb(r, g, b, a))

            r = int(self.selectedModel.emissionColor[0] * 255)
            g = int(self.selectedModel.emissionColor[1] * 255)
            b = int(self.selectedModel.emissionColor[2] * 255)
            a = int(self.selectedModel.emissionColor[3] * 255)
            self.setButtonIconToColor(self.ui.chooseEmissiveColorMat,
QtGui.QColor.fromRgb(r, g, b, a))

            #don't need to call callback, shininess is already set

      QtCore.QObject.disconnect(self.ui.matShinySlider,QtCore.SIGNAL("valueChanged(i
nt)"), self.setShininessColorMat)

      QtCore.QObject.disconnect(self.ui.matShinySpinBox,QtCore.SIGNAL("valueChanged(
double)"), self.setShininessColorMat)

      self.ui.matShinySlider.setValue(int(round(self.selectedModel.shininess)))
            self.ui.matShinySpinBox.setValue(self.selectedModel.shininess)

      QtCore.QObject.connect(self.ui.matShinySlider,QtCore.SIGNAL("valueChanged(int)
"), self.setShininessColorMat)

      QtCore.QObject.connect(self.ui.matShinySpinBox,QtCore.SIGNAL("valueChanged(dou
ble)"), self.setShininessColorMat)

      def updateLightPropertiesTab(self,i):
            if self.glWidget.lights[i].enabled:
                  self.ui.lightEnable.setCheckState(QtCore.Qt.CheckState.Checked)
            else:
                  self.ui.lightEnable.setCheckState(QtCore.Qt.CheckState.Unchecked)
            if self.glWidget.lights[i].directional:

      self.ui.directionalLight.setCheckState(QtCore.Qt.CheckState.Checked)
            else:

      self.ui.directionalLight.setCheckState(QtCore.Qt.CheckState.Unchecked)

            r = int(self.glWidget.lights[i].specularColor[0] * 255)
            g = int(self.glWidget.lights[i].specularColor[1] * 255)
            b = int(self.glWidget.lights[i].specularColor[2] * 255)
            a = int(self.glWidget.lights[i].specularColor[3] * 255)
            self.setButtonIconToColor(self.ui.chooseSpecularColorCam,
QtGui.QColor.fromRgb(r, g, b, a))

            r = int(self.glWidget.lights[i].diffuseColor[0] * 255)
            g = int(self.glWidget.lights[i].diffuseColor[1] * 255)
            b = int(self.glWidget.lights[i].diffuseColor[2] * 255)
            a = int(self.glWidget.lights[i].diffuseColor[3] * 255)
```

```
            self.setButtonIconToColor(self.ui.chooseDiffuseColorCam,
QtGui.QColor.fromRgb(r, g, b, a))

            r = int(self.glWidget.lights[i].ambientColor[0] * 255)
            g = int(self.glWidget.lights[i].ambientColor[1] * 255)
            b = int(self.glWidget.lights[i].ambientColor[2] * 255)
            a = int(self.glWidget.lights[i].ambientColor[3] * 255)
            self.setButtonIconToColor(self.ui.chooseAmbientColorCam,
QtGui.QColor.fromRgb(r, g, b, a))

            r = int(self.glWidget.lights[i].emissiveColor[0] * 255)
            g = int(self.glWidget.lights[i].emissiveColor[1] * 255)
            b = int(self.glWidget.lights[i].emissiveColor[2] * 255)
            a = int(self.glWidget.lights[i].emissiveColor[3] * 255)
            self.setButtonIconToColor(self.ui.chooseEmissiveColorCam,
QtGui.QColor.fromRgb(r, g, b, a))

            self.ui.lightXPos.setValue(self.glWidget.lights[i].position[0])
            self.ui.lightYPos.setValue(self.glWidget.lights[i].position[1])
            self.ui.lightZPos.setValue(self.glWidget.lights[i].position[2])

      def setView(self, view='ortho1'):
            self.ui.setView.setCurrentIndex(0)
            vMin, vMax = self.glWidget.getBoundingBox()
            self.glWidget.camera.setView(vMin, vMax, view=view)
            self.glWidget.updateGL()

      def setButtonIconToColor(self, button, color):
            size = button.iconSize()
            pixmap = QtGui.QPixmap(size.width(), size.height())
            pixmap.fill(color)
            button.setIcon(QtGui.QIcon(pixmap))

      def setLightXPos(self, pos):
            i = self.ui.lightNum.value()
            self.glWidget.lights[i].position[0] = pos
            self.glWidget.lights[i].updateOpenGl()
            self.glWidget.updateGL()
      def setLightYPos(self,pos):
            i = self.ui.lightNum.value()
            self.glWidget.lights[i].position[1] = pos
            self.glWidget.lights[i].updateOpenGl()
            self.glWidget.updateGL()
      def setLightZPos(self,pos):
            i = self.ui.lightNum.value()
            self.glWidget.lights[i].position[2] = pos
            self.glWidget.lights[i].updateOpenGl()
            self.glWidget.updateGL()
      def setLightEnable(self,state):
            i = self.ui.lightNum.value()
            if state == QtCore.Qt.CheckState.Checked:
                  self.glWidget.lights[i].enabled = True
            else:
                  self.glWidget.lights[i].enabled = False
            self.glWidget.lights[i].updateOpenGl()
```

```
            self.glWidget.updateGL()
      def setLightDirectional(self, state):
            i = self.ui.lightNum.value()
            if state == QtCore.Qt.CheckState.Checked:
                  self.glWidget.lights[i].directional = True
                  self.glWidget.lights[i].position[3] = 0.0
            else:
                  self.glWidget.lights[i].directional = False
                  self.glWidget.lights[i].position[3] = 1.0
            self.glWidget.lights[i].updateOpenGl()
            self.glWidget.updateGL()

      def setAmbientColorMat(self):
            if self.selectedModel is None:
                  return
#           QtGui.QColorDialog.setOption(QtGui.QColorDialog.ShowAlphaChannel)
            colorDialog = QtGui.QColorDialog()
            colorDialog.setOption(QtGui.QColorDialog.ShowAlphaChannel,True)
            color = colorDialog.getColor()
#           color =
QtGui.QColorDialog().getColor(QtGui.QColorDialog.ShowAlphaChannel)
#           color = QtGui.QColorDialog().getColor()

            if color.isValid():
                  self.setButtonIconToColor(self.ui.chooseAmbientColorMat, color)
                  self.selectedModel.ambientColor = numpy.array(color.getRgb()) /
255.0
                  if
self.ui.enableColorDrivenModelsCheckBox.checkState()==QtCore.Qt.CheckState.Checked:
                        self.setButtonIconToColor(self.ui.chooseDiffuseColorMat,
color)
                        self.selectedModel.diffuseColor =
numpy.array(color.getRgb()) / 255.0
                  self.glWidget.updateGL()

      def setDiffuseColorMat(self):
            if self.selectedModel is None:
                  return
            color = QtGui.QColorDialog.getColor()
            if color.isValid():
                  self.setButtonIconToColor(self.ui.chooseDiffuseColorMat, color)
                  self.selectedModel.diffuseColor = numpy.array(color.getRgb()) /
255.0
                  self.glWidget.updateGL()

      def setEmissiveColorMat(self):
            if self.selectedModel is None:
                  return
            color = QtGui.QColorDialog.getColor()
            if color.isValid():
                  self.setButtonIconToColor(self.ui.chooseEmissionColorMat, color)
                  self.selectedModel.emissionColor = numpy.array(color.getRgb()) /
255.0
                  self.glWidget.updateGL()
```

```
	def setSpecularColorMat(self):
		if self.selectedModel is None:
			return
		color = QtGui.QColorDialog.getColor()
		if color.isValid():
			self.setButtonIconToColor(self.ui.chooseSpecularColorMat, color)
			self.selectedModel.specularColor = numpy.array(color.getRgb()) /
255.0
			self.glWidget.updateGL()

	def setShininessColorMat(self, value):
		if self.selectedModel is None:
			return
		#prevent recursion

	QtCore.QObject.disconnect(self.ui.matShinySlider,QtCore.SIGNAL("valueChanged(i
nt)"), self.setShininessColorMat)

	QtCore.QObject.disconnect(self.ui.matShinySpinBox,QtCore.SIGNAL("valueChanged(
double)"), self.setShininessColorMat)
		self.ui.matShinySlider.setValue(round(value))
		self.ui.matShinySpinBox.setValue(float(value))

	QtCore.QObject.connect(self.ui.matShinySlider,QtCore.SIGNAL("valueChanged(int)
"), self.setShininessColorMat)

	QtCore.QObject.connect(self.ui.matShinySpinBox,QtCore.SIGNAL("valueChanged(dou
ble)"), self.setShininessColorMat)

		self.selectedModel.shininess = float(value)
		self.glWidget.updateGL()

	def setSpecularColorCam(self):
		color = QtGui.QColorDialog.getColor()
		if color.isValid():
			self.setButtonIconToColor(self.ui.chooseSpecularColorCam, color)
			i = self.ui.lightNum.value()
			self.glWidget.lights[i].specularColor =
numpy.array(color.getRgb()) / 255.0
			self.glWidget.lights[i].updateOpenGl()
			self.glWidget.updateGL()

	def setDiffuseColorCam(self):
		color = QtGui.QColorDialog.getColor()
		if color.isValid():
			self.setButtonIconToColor(self.ui.chooseDiffuseColorCam, color)
			i = self.ui.lightNum.value()
			self.glWidget.lights[i].diffuseColor =
numpy.array(color.getRgb()) / 255.0
			self.glWidget.lights[i].updateOpenGl()
			self.glWidget.updateGL()

	def setAmbientColorCam(self):
		color = QtGui.QColorDialog.getColor()
		if color.isValid():
```

```
                    self.setButtonIconToColor(self.ui.chooseAmbientColorCam, color)
                    i = self.ui.lightNum.value()
                    self.glWidget.lights[i].ambientColor =
numpy.array(color.getRgb()) / 255.0
                    self.glWidget.lights[i].updateOpenGl()
                    self.glWidget.updateGL()

      def setEmissiveColorCam(self):
            color = QtGui.QColorDialog.getColor()
            if color.isValid():
                    self.setButtonIconToColor(self.ui.chooseEmissiveColorCam, color)
                    i = self.ui.lightNum.value()
                    self.glWidget.lights[i].emissiveColor =
numpy.array(color.getRgb()) / 255.0
                    self.glWidget.lights[i].updateOpenGl()
                    self.glWidget.updateGL()

      def populateItemTree(self, joint):
            if joint is not None:
                    self.ui.itemTreeList.clear()
                    self._recursPopulateItemTree(joint)

      def _recursPopulateItemTree(self, joint):
            leaf = modelTreeItem(self.ui.itemTreeList, joint)
            for model in joint.models:
                    modelTreeItem(leaf, model)
            for cJoint in joint.childJoints:
                    self._recursPopulateItemTree(cJoint)

      def createDefaultScene(self):
            #remove current scene
            self.glWidget.clearScene()

            #create and connect joints
            joint1_2 = Joint([0.5,0.5,3.5], parentJoint=self.glWidget.worldJoint,
showAxis = True, axisScale=0.5,name='Joint 1-2')
            joint2_3 = Joint([0.5,0.5,7.5], parentJoint=joint1_2, showAxis = True,
axisScale=0.5,name='Joint 2-3')

            #create bone models

      TriModel.createRectangularSolid([1,1,3],[0,0,0],joint=self.glWidget.worldJoint
,name='Bone 1',color=[0.0,170.0/255, 1.0])

      TriModel.createRectangularSolid([1,1,3],[0,0,4],joint=joint1_2,name='Bone
2',color=[1.0,170.0/255, 0.0])

      TriModel.createRectangularSolid([1,1,3],[0,0,8],joint=joint2_3,name='Bone
3',color=[170.0/255, 0.0, 1.0])

            #reset view
            self.glWidget.resizeGL(self.glWidget.width(), self.glWidget.height())
            self.setView()
            self.setupSceneControls()
```

```
        def clearScene(self):
                self.glWidget.clearScene()
                self.setupSceneControls()

class modelTreeItem(QtGui.QTreeWidgetItem):
        def __init__(self,parent=None, obj=None):
                QtGui.QTreeWidgetItem.__init__(self,parent)
                self.obj=obj
                self.setText(0,self.obj.name)

def main():
        app = QtGui.QApplication(sys.argv)
        myapp = MyMainWindow()
        myapp.show()
        sys.exit(app.exec_())

def mainProfile():
        import cProfile
        '''
        run
                python -m cProfile -o pyAtlasSegmentation.TimeProfile
pyAtlasSegmentation.py
        then run
                python runsnake.py pyAtlasSegmentation.TimeProfile
        '''
        if os.path.exists('pyAtlasSegmentation.TimeProfile'):
                os.remove('pyAtlasSegmentation.TimeProfile')
        cProfile.runctx( 'main()', globals(), locals(),
filename='pyAtlasSegmentation.TimeProfile')
        os.system('runsnake pyAtlasSegmentation.TimeProfile')

if __name__ == '__main__':
        main()
#       mainProfile()
```

TriModel.py

```
'''
Created on Jun 30, 2011

@author: grant
'''
import math, numpy,time
from OpenGL import GL, GLU
#TODO: try and remove cgkit, use numpy matrix instead
from cgkit.cgtypes import quat
from scipy.spatial import cKDTree as KDTree
import numpyTransform

class TriModel:
	'''A 3D model constructed from triangle faces'''
	def __init__(self, *args, **kwargs):
		'''
		Function Signatures:
		TriModel(vertexList, triangleVertexIndexList)
		TriModel(vertexList, triangleVertexIndexList, joint)

		Arguments {default value}:

		vertexList
			Nx3 array of vertices of the form [[x1,y1,z1], [x2,y2,z2], ...]
		triangleVertexIndexList
			Nx3 array of indices where TriangleVertexIndexList[i] represents
the
			ith triangle face of the model in the form
			[vertex1, vertex2, vertex3] where vertex 1, 2, and 3 are indices
of
			the vertexList argument
		joint
			Joint object which serves as center of rotation and determines
model
			orientation

		Keywords:
		normalVectors
			Nx3 array of indices where normalVectors[i] represents the normal
			vector of the ith triangle face in the form [x, y, z]
		removeDuplicates
			removes duplicate vertices from vertexList. {False}
		name
			name of the model
		visible
			boolean that tells if the model should be drawn or not
		color
			array of floats range 0.0-1.0 of from [R,G,B] or [R,G,B,A].
			If A is not given it is defaults to 1.0. {0.8,0.8,0.8,1.0}
		alpha
			transparency level of model, range 0.0-1.0, transparent to opaque
respectively
		updateOnlyFromGrandparentJoints
```

```
                        If set, orientation changes to parent joint will not cause model
                        to move, only orientation to grandparent joints will move model.
                        This is useful for models that represent things independent of
joint
                        orientation, such as axis models. {False}
                '''
                self.OriginalVertexList = numpy.array(args[0])
                self.TriangleVertexIndexList = numpy.array(args[1])
                self.VertexList = self.OriginalVertexList.copy()
                self.transformedVertexList = self.OriginalVertexList.copy()
                self.minPoint = numpy.min(self.OriginalVertexList,axis=0)
                self.maxPoint = numpy.max(self.OriginalVertexList,axis=0)
                self.maxPointTransformed = self.maxPoint.copy()
                self.minPointTransformed = self.minPoint.copy()
                if 'normalVectors' in kwargs and numpy.sum(kwargs['normalVectors']) !=
0:     #make sure normal vectors are actually there
                        self.NormVectors = numpy.array(kwargs['normalVectors'])
                else:
                        self.NormVectors =
numpy.zeros((len(self.TriangleVertexIndexList),3))
                        self.calulateNormals()
                self.OriginalNormVectors = self.NormVectors.copy()
                self.joint = None
                self.CoG = None
                self.PCAvectors = None
                self.kdtree = None
                self.initialRotationCenter = numpy.array([0.0,0.0,0.0])
                if 'removeDuplicates' in kwargs and
isinstance(kwargs['removeDuplicates'], bool) and kwargs['removeDuplicates']:
                        self._removeDuplicateVertices()
                if 'name' in kwargs:
                        self.name = kwargs['name']
                else:
                        self.name = 'model'
                if 'visible' in kwargs:
                        self.visible = kwargs['visible']
                else:
                        self.visible = True
                if 'referenceVolume' in kwargs:
                        self.referenceVolume = kwargs['referenceVolume']
                else:
                        self.referenceVolume = None
                if 'textureID' in kwargs:
                        self.textureID = kwargs['textureID']    #must be a properly
initialized OpenGL 2D texture
                else:
                        self.textureID = None
                if 'displayAsPoints' in kwargs:
                        self.displayAsPoints = kwargs['displayAsPoints']
                else:
                        self.displayAsPoints=False

                if len(args)>2:
                        try:
                                self.setJoint(args[2])
```

```
                        except Exception:
                                pass
                elif 'joint' in kwargs:
                        self.setJoint(kwargs['joint'])

                self.ambientColor = [0.8,0.8,0.8,1.0]
                self.diffuseColor = [0.8,0.8,0.8,1.0]
                self.emissionColor = [0.0,0.0,0.0,1.0]
                self.specularColor = [1.0,1.0,1.0,1.0]
                self.shininess = 128.0
                if 'color' in kwargs and kwargs['color'] is not None:
                        color = kwargs['color']
                        if len(color) == 3:
                                color.append(1.0)
                                self.ambientColor = color
                                self.diffuseColor = color[:]
                        elif len(color) == 4:
                                self.ambientColor = color
                                self.diffuseColor = color[:]
                if 'alpha' in kwargs and kwargs['alpha'] is not None:
                        self.ambientColor[3] = kwargs['alpha']
                        self.diffuseColor[3] = kwargs['alpha']

                #only update from grandparentjoint, not parent joint
                if 'updateOnlyFromGrandparentJoints' in kwargs and
kwargs['updateOnlyFromGrandparentJoints'] is True:
                        self.updateOnlyFromGrandparentJoints = True
                else:
                        self.updateOnlyFromGrandparentJoints = False

                self.pointSize = 1.0

                #openGL Display List
                self.displayList = None
                if 'skipOpenGL' not in kwargs:
                        self.createOpenGLDisplayList()

        def __del__(self):
                if self.displayList is not None:
                        GL.glDeleteLists(self.displayList,1)

        def setScale(self,scale,absolute=True):
                #FIXME: add ability to scale model in addition to joints, this is not
applied to transformation function
                if absolute:
                        self.scale = scale
                else:
                        self.scale *= scale

        def calulateNormals(self):
                for i in xrange(len(self.TriangleVertexIndexList)):
                        v0 = self.VertexList[self.TriangleVertexIndexList[i,0]]
                        v1 = self.VertexList[self.TriangleVertexIndexList[i,1]]
                        v2 = self.VertexList[self.TriangleVertexIndexList[i,2]]
                        self.NormVectors[i] = numpy.cross(v2-v0, v1-v0)
```

```
	def invertNormals(self):
		self.OriginalNormVectors = -1.0*self.OriginalNormVectors
		self.NormVectors = self.OriginalNormVectors.copy()
		self.createOpenGLDisplayList()

	def setJoint(self, joint):
		self.initialRotationCenter = joint.location.copy()
		self.joint = joint
		self.joint.models.append(self)

	def transformVertices(self, transform = numpy.matrix(numpy.identity(4)),
modelID=None):
		if modelID is not None and modelID != id(self):
			return
		self.transformedVertexList = numpyTransform.transformPoints(transform,
self.OriginalVertexList)
		self.maxPointTransformed = numpyTransform.transformPoints(transform,
self.maxPoint[numpy.newaxis,:])
		self.minPointTransformed = numpyTransform.transformPoints(transform,
self.minPoint[numpy.newaxis,:])

		#TODO: not sure that this is necessary
		self.kdtree = KDTree(self.transformedVertexList)

	def getBoundingBox(self, transform = numpy.matrix(numpy.identity(4))):
		if self.minPoint is None or self.maxPoint is None:
			return None, None
		minP = numpy.matrix([self.minPoint[0], self.minPoint[1],
self.minPoint[2], 1.0])
		maxP = numpy.matrix([self.maxPoint[0], self.maxPoint[1],
self.maxPoint[2], 1.0])
		minP = transform * minP.T
		maxP = transform * maxP.T
		minP = numpy.array(minP).squeeze()[:3]
		maxP = numpy.array(maxP).squeeze()[:3]
		return minP, maxP

	#TODO: Try using VBO to render dynamic objects, and displaylists for static
objects
	def createOpenGLDisplayList(self):
		if self.displayList is None:
			self.displayList = GL.glGenLists(1)
			error = GL.glGetError()
			if error != GL.GL_NO_ERROR:
				print "An OpenGL error has occurred: ",
GLU.gluErrorString(error)
				return
		GL.glNewList(self.displayList, GL.GL_COMPILE)
		if self.displayAsPoints:
			GL.glBegin(GL.GL_POINTS)
		else:
			GL.glBegin(GL.GL_TRIANGLES)
		for i in xrange(len(self.TriangleVertexIndexList)):
			if len(self.TriangleVertexIndexList[i]) != 3:
```

```
                        continue
                GL.glNormal3fv(self.NormVectors[i])

        GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,0]])

        GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,1]])

        GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,2]])
                GL.glEnd()
                if self.displayAsPoints:
                        GL.glBegin(GL.GL_POINTS)
                else:
                        GL.glBegin(GL.GL_QUADS)
                for i in xrange(len(self.TriangleVertexIndexList)):
                        if len(self.TriangleVertexIndexList[i]) != 4:
                                continue

                        GL.glNormal3fv(self.NormVectors[i])
                        if self.textureID is not None:
                                GL.glTexCoord2d(0.0,0.0)

        GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,0]])
                        if self.textureID is not None:
                                GL.glTexCoord2d(0.0,1.0)

        GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,1]])
                        if self.textureID is not None:
                                GL.glTexCoord2d(1.0,1.0)

        GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,2]])
                        if self.textureID is not None:
                                GL.glTexCoord2d(1.0,0.0)

        GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,3]])
                GL.glEnd()

                GL.glEndList()

        #Display initial model, but modify its position/orientation with openGL matrix
transformations
        def OpenGLPaint(self, colorDrivenMaterial=None, useCallLists=True):
                if self.visible:
                        #TODO: add support for updateOnlyFrom GrandparentJoints flag
#                       localTransform = grandParentTransformMat
#                       #set orientation
#                       if not self.updateOnlyFromGrandparentJoints:
#                               localTransform *= self.joint.
#                               q = self.joint.orientation

                        #set point size if required
                        if self.displayAsPoints:
                                GL.glPointSize(self.pointSize)

                        #set up model material
                        if colorDrivenMaterial is not None:
```

```
                        GL.glMaterialf(GL.GL_FRONT_AND_BACK, GL.GL_SHININESS,
self.shininess);
                        if colorDrivenMaterial:
                                GL.glColorMaterial(GL.GL_FRONT_AND_BACK,
GL.GL_AMBIENT_AND_DIFFUSE)
                                GL.glMaterialfv(GL.GL_FRONT_AND_BACK,
GL.GL_EMISSION, [0.0,0.0,0.0,1.0]);
                                GL.glMaterialfv(GL.GL_FRONT_AND_BACK,
GL.GL_SPECULAR, [1.0,1.0,1.0,1.0]);
                                GL.glColor4fv(self.ambientColor)
                        else:
                                GL.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_AMBIENT,
self.ambientColor);
                                GL.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_DIFFUSE,
self.diffuseColor);
                                GL.glMaterialfv(GL.GL_FRONT_AND_BACK,
GL.GL_EMISSION, self.emissionColor);
                                GL.glMaterialfv(GL.GL_FRONT_AND_BACK,
GL.GL_SPECULAR, self.specularColor);

                    if self.textureID is not None:
                            GL.glBindTexture( GL.GL_TEXTURE_2D, self.textureID )#bind
our texture to our shape

                    if useCallLists:    #use call lists of original model, much
faster for displaying
                        #change openGL matrix to match model orientation and
position
                        GL.glPushMatrix()
                        #matrix transformations steps: (applied in reverse order)
                        #1: move model initial joint (rotation) center to origin
                        #2: scale model
                        #3: rotate model to new orientation
                        #4: move model to parent joint position
#
      GL.glTranslatef(self.joint.location[0],self.joint.location[1],self.joint.locat
ion[2])
#                       GL.glMultMatrixf(numpy.array(q.toMat4()))
#                       scale = self.scale*self.joint.scale    #scale is
combination of model scale and parent joint scale
#                       GL.glScalef(scale,scale,scale)
#                       GL.glTranslatef(-self.initialRotationCenter[0],-
self.initialRotationCenter[1],-self.initialRotationCenter[2])

                        if self.displayList is not None:
                                GL.glCallList(self.displayList)

                        #go back to default matrix
                        GL.glPopMatrix()
                    else:  #don't use call lists, display recalculated vertexes
                        #TODO: test how scaling works when vertexes are
recalculated
                        #TODO: Test how joint translation works
                        GL.glBegin(GL.GL_TRIANGLES)
                        for i in xrange(len(self.TriangleVertexIndexList)):
```

```
                                    GL.glNormal3fv(self.NormVectors[i])

            GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,0]])

            GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,1]])

            GL.glVertex3fv(self.VertexList[self.TriangleVertexIndexList[i,2]])
                            GL.glEnd()

        def _removeDuplicateVertices(self):
                #FIXME: not currently working
                origTriangleVertexIndexListShape = self.TriangleVertexIndexList.shape
                self.TriangleVertexIndexList = self.TriangleVertexIndexList.flatten()
                i=0;
                while i < len(self.OriginalVertexList)-1:
                        v = self.OriginalVertexList[i]
                        sameValIndex = []
                        for j in xrange(i+1,len(self.OriginalVertexList)):
                                if (v==self.OriginalVertexList[j]).all():
                                        sameValIndex.append(j)
                                        #change vertex index numbers in
TriangleVertexIndexList
                                        sameTriVertexIndex =
numpy.where(self.TriangleVertexIndexList==j)
                                        self.TriangleVertexIndexList[sameTriVertexIndex] = i
                        #remove duplicate vertices
                        self.OriginalVertexList =
numpy.delete(self.OriginalVertexList,sameValIndex,0)
                        i+=1
                self.VertexList = self.OriginalVertexList.copy()
                #change shape of TriangleVertexIndexList back to original of Nx3
                self.TriangleVertexIndexList.shape = origTriangleVertexIndexListShape

        def updateFromJoint(self):
                #TODO: I think this has been replaced by the transformVertices function
                if self.joint is None:
                        return
                if self.updateOnlyFromGrandparentJoints:
                        if self.joint.parentJoint is None:
                                q = quat(1,0,0,0)
                        else:
                                q = self.joint.parentJoint.orientation
                else:
                        q=self.joint.orientation
                #update vertex
                for i in xrange(len(self.OriginalVertexList)):
                        scale = self.scale*self.joint.scale     #scale is combination of
model scale factor and joint scale factor
                        v = scale*(self.OriginalVertexList[i] -
self.initialRotationCenter)      #get scaled vector based on initial rotation center
                        v = numpy.array(q.rotateVec(v))  #apply current orientation
                        self.VertexList[i] = v + self.joint.location  #move vector to
joint location

                #update normals (orientation only, not scaled or moved)
```

```
            for i in xrange(len(self.OriginalNormVectors)):
                  n = self.OriginalNormVectors[i]
                  n = numpy.array(q.rotateVec(n))
                  self.NormVectors[i] = n

      def calculateCenterOfGravity(self,recalculate=False):
            '''
            estimates CoG by sum(2*triangle area * triangle Cog) / sum(2*triangle
area)
            from http://paulbourke.net/geometry/polyarea/
            assumes uniform density
            '''
            #TODO: I don't think this is used anymore
            if self.CoG is None or recalculate is True:
                  t1 = time.time()
                  num=0.0
                  den = 0.0
                  for tri in self.TriangleVertexIndexList:
                        centerVertex = (self.OriginalVertexList[tri[0]] +
self.OriginalVertexList[tri[1]] + self.OriginalVertexList[tri[2]]) / 3
                        v = numpy.cross(self.OriginalVertexList[tri[1]]-
self.OriginalVertexList[tri[0]] , self.OriginalVertexList[tri[2]]-
self.OriginalVertexList[tri[0]])
                        faceArea2x = numpy.sqrt(numpy.vdot(v,v))
                        num += faceArea2x * centerVertex
                        den += faceArea2x
                  t2 = time.time()
                  print "Time to cal CoG:",t2-t1
                  self.CoG = num/den
#                   self.CoG = numpy.sum(self.OriginalVertexList) /
len(self.OriginalVertexList)
            return self.CoG

      def PCA(self,recalculate=False):
            '''
            Perform Principle component analysis to get 3 major axis
            Equations from paper 'A robust mesh watermarking scheme based on PCA' by
Bin Yang, Xiao-Qian Li, Wei Li
            '''
            if self.PCAvectors is None or recalculate is True:
#                   vc = self.calculateCenterOfGravity()
#                   #TODO: this might take up a lot of memory, check if its faster to
do the offset all at once, or as we go
#                   vertList = self.OriginalVertexList - vc
                  vertList = self.OriginalVertexList.copy()

                  C = numpy.cov(vertList, rowvar=0)
                  d,v=numpy.linalg.eig(C)

                  #ordering from greatest to least is not guaranteed, manually sort
                  indx=numpy.argsort(-d)
                  d=d[indx]
                  v = v[:,indx]
                  self.PCAvectors = v.T
                  print 'Principle Component Analysis axis:'
```

```
                print self.PCAvectors
            return self.PCAvectors

      def compareToTransformedPoints(self, point, currentClosestSqDistance=None,
currentModelID=None, modelName=''):
            #calculate closest distance from point to a vertex in this model
            #TODO: faster way? check to see distance to bounding box planes. If
these are not better than currentClosestDistance, don't need to check every vertex of
this model

            #check point distance to bounding box planes, if none of them is shorter
than currentClosest distance, no need to keep looking at this bone
            bbPoints = numpy.array([[self.transformedMinPoint[0],
      self.transformedMinPoint[1],     self.transformedMinPoint[2]],
                                                [self.transformedMinPoint[0],
      self.transformedMinPoint[1],     self.transformedMaxPoint[2]],
                                                [self.transformedMinPoint[0],
      self.transformedMaxPoint[1],     self.transformedMinPoint[2]],
                                                [self.transformedMinPoint[0],
      self.transformedMaxPoint[1],     self.transformedMaxPoint[2]],
                                                [self.transformedMaxPoint[0],
      self.transformedMinPoint[1],     self.transformedMinPoint[2]],
                                                [self.transformedMaxPoint[0],
      self.transformedMinPoint[1],     self.transformedMaxPoint[2]],
                                                [self.transformedMaxPoint[0],
      self.transformedMaxPoint[1],     self.transformedMinPoint[2]],
                                                [self.transformedMaxPoint[0],
      self.transformedMaxPoint[1],     self.transformedMaxPoint[2]]])
            placesPointNumbers = numpy.array([     [0,1,3,2],
                                                                  [0,2,6,4],
                                                                  [0,4,5,1],
                                                                  [2,3,7,6],
                                                                  [1,5,7,3],
                                                                  [4,6,7,5]])

            if currentClosestSqDistance is not None and currentModelID is not None:
                  for i in xrange(placesPointNumbers.shape[0]):
                        v1 = bbPoints[placesPointNumbers[i,1]] -
bbPoints[placesPointNumbers[i,0]]
                        v2 = bbPoints[placesPointNumbers[i,3]] -
bbPoints[placesPointNumbers[i,0]]
                        v = numpy.cross(v1, v2)
                        w = point-bbPoints[placesPointNumbers[i,0]]
                        distance = numpy.abs(numpy.dot(v,w))/numpy.linalg.norm(v)
                        if distance < currentClosestSqDistance:
                              break
                  else:
                        #no distance is less then currentClosest, just return
#                       print 'Skipped Bone %d %s' % (id(self), self.name)
                        return currentClosestSqDistance, currentModelID, modelName

            #this bone might have a vertex closer, try looking
            minDistance = numpy.min(numpy.sqrt(numpy.sum((point-
self.transformedVertexList)**2, axis=1)))
```

```
            if currentClosestSqDistance is None or minDistance <
currentClosestSqDistance:
                  currentClosestSqDistance = minDistance
                  currentModelID = id(self)
                  modelName = self.name
            return currentClosestSqDistance, currentModelID, modelName

      def compareToTransformedPointsKDTrees(self, point,
currentClosestSqDistance=None, currentModelID=None, modelName=''):
            distance, i = self.kdtree.query(point)
            if currentClosestSqDistance is None or distance <
currentClosestSqDistance:
                  currentClosestSqDistance = distance
                  currentModelID = id(self)
                  modelName = self.name
            return currentClosestSqDistance, currentModelID, modelName

def createRectangularSolid(*args, **kwargs):
      '''
      Function Signatures:
      createRectangularSolid(...)
      createRectangularSolid(dimensions, ...)
      createRectangularSolid(dimensions, offset, ...)

      Arguments {default value}:

      dimensions
            array like object of form [length, width, height] that describes the
            dimensions of the model to be created. {1,1,1}
      offset
            array like object of form [x, y, z] that describes the
            offset applied to each vertex of the model. {0,0,0}

      Keywords:
      joint
            Joint object that the model will rotate around. {None}
      '''
      if len(args) > 0:
            dimensions = numpy.array(args[0], dtype=float)
      else:
            dimensions = numpy.array([1.0,1.0,1.0])

      x = dimensions[0]
      y = dimensions[1]
      z = dimensions[2]
      vList = []

      #z=0 plane
      f = numpy.array([[0.0,0.0,0.0],
                                          [x,0.0,0.0],
                                          [x,y,0.0]])
      vList.extend(f)
      f = numpy.array([[0.0,0.0,0.0],
                                          [x,y,0.0],
                                          [0.0,y,0.0]])
```

```
vList.extend(f)
#z=z plane
f = numpy.array([[x,y,z],
                                        [x,0.0,z],
                                        [0.0,0.0,z]])
vList.extend(f)
f = numpy.array([[0.0,y,z],
                                        [x,y,z],
                                        [0.0,0.0,z]])
vList.extend(f)
#y=0 plane
f = numpy.array([[0.0,0.0,0.0],
                                        [0.0,0.0,z],
                                        [x,0.0,z]])
vList.extend(f)
f = numpy.array([[0.0,0.0,0.0],
                                        [x,0.0,z],
                                        [x,0.0,0.0]])
vList.extend(f)
#y=y plane
f = numpy.array([[x,y,z],
                                        [0.0,y,z],
                                        [0.0,y,0.0]])
vList.extend(f)
f = numpy.array([[x,y,0.0],
                                        [x,y,z],
                                        [0.0,y,0.0]])
vList.extend(f)
#x=0 plane
f = numpy.array([[0.0,0.0,0.0],
                                        [0.0,y,0.0],
                                        [0.0,y,z]])
vList.extend(f)
f = numpy.array([[0.0,0.0,0.0],
                                        [0.0,y,z],
                                        [0.0,0,z]])
vList.extend(f)
#x=x plane
f = numpy.array([[x,y,z],
                                        [x,y,0.0],
                                        [x,0.0,0.0]])
vList.extend(f)
f = numpy.array([[x,0.0,z],
                                        [x,y,z],
                                        [x,0,0.0]])
vList.extend(f)

#add offset
if len(args) > 1:
       vList += numpy.array(args[1], dtype=float)
triVert = numpy.array(range(len(vList)))
triVert = triVert.reshape( [len(vList)/3,3] )
if 'joint' in kwargs:
       rect = TriModel(vList, triVert, kwargs['joint'],**kwargs)
else:
```

```
            rect = TriModel(vList, triVert,**kwargs)
        return rect

def createCylinder(*args, **kwargs):
        '''
        Function Signatures:
        createCylinder(...)
        createCylinder(length, ...)
        createCylinder(length, radius, ...)
        createCylinder(length, radius, offset, ...)

        Arguments {default value}:

        length
                length of cylinder along z axis {1.0}
        radius
                radius of cylinder ends {1.0}
        offset
                offsets the center of the bottom cylinder cap.
                Given in form [x,y,z]. { [0.0,0.0,0.0] }

        Keywords:
        joint
                Joint object that the model will rotate around. {None}
        ngon
                number of edges used to approximate the circle endcaps {10}
        '''
        if len(args) > 0:
                length = float(args[0])
        else:
                length = 1.0
        if len(args) > 1:
                r = float(args[1])
        else:
                r = 1.0
        if 'ngon' in kwargs:
                N = int(kwargs['ngon'])
        elif 'Ngon' in kwargs:
                N = int(kwargs['Ngon'])
        else:
                N = 10
        if N < 3:
                N = 3

        vList = []

        #draw end caps
        for n in xrange(1,N-1):
                x1 = r*math.cos(2*math.pi*n/N)
                y1 = r*math.sin(2*math.pi*n/N)
                x2 = r*math.cos(2*math.pi*(n+1)/N)
                y2 = r*math.sin(2*math.pi*(n+1)/N)
                #draw top circle
                f = [[r, 0.0,length],
                        [x1, y1, length],
```

```
                [x2, y2, length]]
            vList.extend(f)
            #draw bottom circle
            f = [[r,0.0,0.0],
                [x2,y2, 0.0],
                [x1,y1,0.0]]
            vList.extend(f)

        #draw sides
        for n in xrange(0,N):
            x1 = r*math.cos(2*math.pi*n/N)
            y1 = r*math.sin(2*math.pi*n/N)
            if n == N-1:
                x2 = r
                y2 = 0.0
            else:
                x2 = r*math.cos(2*math.pi*(n+1)/N)
                y2 = r*math.sin(2*math.pi*(n+1)/N)
            f =[[x1,y1,0.0],
                [x2,y2,0.0],
                [x2,y2,length]]
            vList.extend(f)
            f =[[x1,y1,0.0],
                [x2,y2,length],
                [x1,y1,length]]
            vList.extend(f)

        vList = numpy.array(vList)
        #add offset
        if len(args) > 2:
            vList += numpy.array(args[2], dtype=float)
        triVert = numpy.array(range(len(vList)))
        triVert = triVert.reshape( [len(vList)/3,3] )
        return TriModel(vList, triVert, **kwargs)

def createCone(*args, **kwargs):
        '''
        Function Signatures:
        createCone(...)
        createCone(radius, ...)
        createCone(radius, height, ...)
        createCone(radius, height, offset, ...)

        Arguments {default value}:

        radius
            radius of cone {1.0}
        height
            height of cone {twice value as radius}
        offset
            offsets the center of the bottom cylinder cap.
            Given in form [x,y,z]. { [0.0,0.0,0.0] }

        Keywords:
        joint
```

```
        Joint object that the model will rotate around. {None}
ngon
        number of edges used to approximate the circle bottom {10}
axis
        axis this cone is parallel to, 'x', 'y', or 'z'
'''
if len(args) > 0:
        r = float(args[0])
else:
        r = 1.0
if len(args) > 1:
        height = float(args[1])
else:
        height = r*2.0
if 'ngon' in kwargs:
        N = int(kwargs['ngon'])
elif 'Ngon' in kwargs:
        N = int(kwargs['Ngon'])
else:
        N = 10
if N < 3:
        N = 3
if 'axis' in kwargs:
        if kwargs['axis'].lower()=='x':
                initialOrintation = quat(math.pi/2,[0,1,0])
        elif kwargs['axis'].lower()=='y':
                initialOrintation = quat(-math.pi/2,[1,0,0])
        else:
                initialOrintation = quat(1)
else:
        initialOrintation = quat(1)
vList = []
normList = []

for n in xrange(N):
        x1 = r*math.cos(2*math.pi*n/N)
        y1 = r*math.sin(2*math.pi*n/N)
        x2 = r*math.cos(2*math.pi*(n+1)/N)
        y2 = r*math.sin(2*math.pi*(n+1)/N)
        #draw cone
        f = [[0.0, 0.0,height],
                [x2, y2, 0.0],
                [x1, y1, 0.0]]
        vList.extend(f)
        f=numpy.array(f)
        norm = numpy.cross(f[2]-f[0], f[1]-f[0])
        normList.append(norm)
        #draw circle
        f = [[0.0,0.0,0.0],
                [x1,y1, 0.0],
                [x2,y2,0.0]]
        vList.extend(f)
        f=numpy.array(f)
        norm = numpy.cross(f[2]-f[0], f[1]-f[0])
        normList.append(norm)
```

```
	vList = numpy.array(vList)
	normList = numpy.array(normList)

	#change vertex to new orientation
	for i in xrange(len(vList)):
		vList[i] = numpy.array(initialOrintation.rotateVec(vList[i]))

	#add offset
	if len(args) > 2:
		vList += numpy.array(args[2], dtype=float)
	triVert = numpy.array(range(len(vList)))
	triVert = triVert.reshape( [len(vList)/3,3] )
	if 'joint' in kwargs:
		rect = TriModel(vList, triVert, kwargs['joint'], normalVectors=normList,
**kwargs)
	else:
		rect = TriModel(vList, triVert, normalVectors=normList, **kwargs)
	return rect

def createTolRegion( angle, AN=9, RN=8, **kwargs):
	angleSteps = numpy.linspace(0, angle, AN)
	rotSteps = numpy.linspace(0.0, 2*math.pi, RN, endpoint=False)
	v = []
	tri = []
	for ai in xrange(angleSteps.shape[0]):
		if ai == 0:  #first point
			v.append(numpy.array([0.0, 0.0, 1.0]))
			continue
		a = angleSteps[ai]

		for ri in xrange(rotSteps.shape[0]):
			r = rotSteps[ri]
			#generate new vertex
			vtemp = numpy.array([0.0, 0.0, 1.0])
			T = numpyTransform.rotation(r, [0,0,1], N=3) *
numpyTransform.rotation(a, [1,0,0], N=3)
			vtemp = (T*numpy.matrix(vtemp).T).getA().squeeze()
			v.append(vtemp)
			vi = len(v)-1
			if ai == 1:  #first ring is special case
				if ri == 0:  #first rotation, there are only two points,
not enough to make a triangle
					continue
				tri.append( [0, vi, vi-1] )
				if ri == RN-1:      #last triangle in first ring
					tri.append( [0,1,RN] )
			else: #normal case
				if ri < RN-1:
					tri.append( [vi, vi-RN, vi-RN+1] )
				else:
					tri.append( [vi, vi-RN, vi-RN-RN+1] )
				if ri > 0:   #first rotation point can only make one
triangle
					tri.append( [vi, vi-1, vi-RN] )
```

```
                        if ri == RN-1:      #need to do that triangle that wasn't
possible the on first rotation point
                                tri.append( [vi, vi-RN+1-RN, vi-RN+1] )

        else:  #close shape on last ring
                v.append(numpy.array([0.0, 0.0, 0.0])) #last point
                vi = len(v)-1
                for ri in xrange(RN-1):
                        tri.append( [vi, vi-RN+ri, vi-RN+1+ri] )
                else:  #make last triangle
                        tri.append( [vi, vi-1,vi-RN] )

        v = numpy.array(v)
        tri = numpy.array(tri)

        #TODO: align to vec

        return TriModel(v, tri, **kwargs)
```

numpyTransform.py

```
'''
Created on Feb 4, 2012

@author: Jeff
'''

import numpy, math

def rotationMat2Euler(rotMat):
	'''
	Conventions:
	Coord Syste: right hand
	rotMat = Ry*Rz*Rx
	ref: http://www.euclideanspace.com/maths/geometry/rotations/euler/index.htm

	rotMat must be a rotation matrix only, i.e. no scaling

	angles in radians
	return (angleXaxis, angleYaxis, angleZaxis)
	'''
	rotMat, tx, ty, tz, sx, sy,sz = decomposeMatrix(rotMat)
	if (rotMat[1,0] > 0.998): #singularity at north pole
		angley = math.atan2(rotMat[0,2],rotMat[2,2])
		anglez = math.pi/2
		anglex = 0.0
	elif (rotMat[1,0] < -0.998):     #singularity at south pole
		angley = math.atan2(rotMat[0,2],rotMat[2,2])
		anglez = -math.pi/2
		anglex = 0.0
	else:
		angley = math.atan2(-rotMat[2,0],rotMat[0,0])
		anglex = math.atan2(-rotMat[1,2],rotMat[1,1])
		anglez = math.asin(rotMat[1,0])
	return anglex, angley, anglez

def pointsInToleranceRange(points, Vec, angle, trans):
	'''
	assumptions
	primaryVec starts at origin
	length of primary vec defines how far out points can be
	trans is in form [x,y,z] where each element is +/- limit of movement in that
direction
	'''
	pindx = numpy.zeros(points.shape[0], dtype=numpy.bool)

	bbPoints = numpy.array([[-trans[0],    -trans[1],   -trans[2]],
	                        [-trans[0],   -trans[1],    trans[2]],
	                        [-trans[0],    trans[1],   -trans[2]],
	                        [-trans[0],    trans[1],    trans[2]],
	                        [ trans[0],   -trans[1],   -trans[2]],
	                        [ trans[0],   -trans[1],    trans[2]],
	                        [ trans[0],    trans[1],   -trans[2]],
	                        [ trans[0],    trans[1],    trans[2]]])
```

```
        VecLen=numpy.sqrt(numpy.sum(Vec**2))
        VecNorm = Vec/VecLen

        #if point is within tolerance box its ok
        pindx = numpy.logical_or(pindx, numpy.all(trans-numpy.abs(points) >= 0,
axis=1))

        pvecbb = numpy.array([points[j]-bbPoints for j in xrange(points.shape[0])])
        pvecLen=numpy.sqrt(numpy.sum(pvecbb**2, axis=2))
        #pvec is on the corner of the tolerance box, its ok
        pindx = numpy.logical_or(pindx, numpy.any(pvecLen==0.0, axis=1))

        pvecNorm = pvecbb/numpy.repeat(pvecLen[:,:,numpy.newaxis], 3, axis=2)
        vn=numpy.tile(VecNorm,
pvecNorm.shape[0]*pvecNorm.shape[1]).reshape(pvecNorm.shape)
        dot = numpy.sum(vn*pvecNorm, axis=2)    #dot product VecNorm*pvecNorm
        pangle = numpy.arccos(dot)
        angleComp = pangle <= angle
        lenComp = pvecLen <= VecLen
        #if, for any corner of tolerance box, angle between vectors is less than max
and pvec length is less than Vec length, point is ok
        pindx = numpy.logical_or(pindx, numpy.any(numpy.logical_and(angleComp,
lenComp), axis=1))

        #if there exists a case where for one vertex, the angle is bad but the length
is good and another vertex where the length is bad but the angle is good
        #then some point inside the box would work as an origin for pvec so this point
must be reachable
        indx = numpy.logical_xor(angleComp, lenComp)
        pindx = numpy.logical_or(pindx,
numpy.logical_and(numpy.any(numpy.logical_and(angleComp, indx), axis=1),
numpy.any(numpy.logical_and(lenComp, indx), axis=1)))

        return pindx

def pointsInToleranceRange2(points, Vec, angle, trans):
        '''
        This is the same as pointsInToleranceRange() except that it looks through
every vertex and therefore is much slower.
        '''
        pindx = numpy.zeros(points.shape[0], dtype=numpy.bool)

        bbPoints = numpy.array([[-trans[0],    -trans[1],   -trans[2]],
                                [-trans[0],  -trans[1],    trans[2]],
                                [-trans[0],   trans[1],   -trans[2]],
                                [-trans[0],   trans[1],    trans[2]],
                                [ trans[0],  -trans[1],   -trans[2]],
                                [ trans[0],  -trans[1],    trans[2]],
                                [ trans[0],   trans[1],   -trans[2]],
                                [ trans[0],   trans[1],    trans[2]]])
        VecLen=numpy.linalg.norm(Vec)
        VecNorm = Vec/VecLen
        pointsInTolBox = 0
        pointsOnCornersOfTolBox = 0
        pointsReachableFromCorners = 0
```

```
	pointsReachableFromInsideTolBox = 0

	#find closest point on trans bounding box
	for i in xrange(points.shape[0]):
		point = points[i]

		#if point is within tolerance box its ok
		if numpy.all(trans-numpy.abs(point) >= 0):
			pindx[i] = True
			pointsInTolBox+=1
			continue
		pvecbb = point-bbPoints
		pvecLen=numpy.apply_along_axis(numpy.linalg.norm, 1, pvecbb)

		#pvec is on the corner of the tolerance box, its ok
		if numpy.any(pvecLen == 0.0):
			pindx[i] = True
			pointsOnCornersOfTolBox+=1
			continue
		pvecNorm = numpy.array([pvecbb[j]/pvecLen[j] for j in
xrange(pvecbb.shape[0])])
		pangle = numpy.array([numpy.arccos(numpy.dot(VecNorm, pvecNorm[j])) for
j in xrange(pvecNorm.shape[0])])
		angleComp = pangle <= angle
		lenComp = pvecLen <= VecLen
		#if, for any corner of tolerance box, angle between vectors is less than
max and pvec length is less than Vec length, point is ok
		if numpy.any(numpy.logical_and(angleComp, lenComp)):
			pindx[i] = True
			pointsReachableFromCorners+=1
			continue
		#if there exists a case where for one vertex, the angle is bad but the
length is good and another vertex where the length is bad but the angle is good
		#then some point inside the box would work as an origin for pvec so this
point must be reachable
		indx = numpy.logical_xor(angleComp, lenComp)
		if numpy.any(angleComp[indx]) and numpy.any(lenComp[indx]):
			pindx[i] = True
			pointsReachableFromInsideTolBox+=1
			continue

#	print 'Points in Tolerance Box:', pointsInTolBox
#	print 'Points on Corner of Tolerance Box:', pointsOnCornersOfTolBox
#	print 'Points Reachable from Corners of Tolerance Box:',
pointsReachableFromCorners
#	print 'Points Reachable from Inside Tolerance Box:',
pointsReachableFromInsideTolBox

	return pindx

def pointsInBox(points, boxDiagPointA, boxDiagPointB, scale=1.0):
	diagVec = numpy.append(boxDiagPointA, boxDiagPointB)
	diagVec = diagVec.reshape((2,3))
	boxMin = diagVec.min(axis=0)
	boxMax = diagVec.max(axis=0)
```

```
	pointsIndx = numpy.ones(points.shape[0], dtype=numpy.bool)
	boxCenter = (boxMax+boxMin)/2
	boxMax = scale*(boxMax-boxCenter) + boxCenter
	boxMin = scale*(boxMin-boxCenter) + boxCenter

	bbPoints = numpy.array([[boxMin[0],    boxMin[1],   boxMin[2]],
                                        [boxMin[0],  boxMin[1],   boxMax[2]],
                                        [boxMin[0],  boxMax[1],   boxMin[2]],
                                        [boxMin[0],  boxMax[1],   boxMax[2]],
                                        [boxMax[0],  boxMin[1],   boxMin[2]],
                                        [boxMax[0],  boxMin[1],   boxMax[2]],
                                        [boxMax[0],  boxMax[1],   boxMin[2]],
                                        [boxMax[0],  boxMax[1],   boxMax[2]]])
	facePointIndx = numpy.array([    [0,1,3,2],
                                                [0,2,6,4],
                                                [0,4,5,1],
                                                [2,3,7,6],
                                                [1,5,7,3],
                                                [4,6,7,5]])

	#remove model data that is in front the neck
	for plane in xrange(facePointIndx.shape[0]):
		v1 = bbPoints[facePointIndx[plane,3]] - bbPoints[facePointIndx[plane,0]]
		v2 = bbPoints[facePointIndx[plane,1]] - bbPoints[facePointIndx[plane,0]]
		norm = numpy.cross(v1, v2)#normal points inside
		planePoint = bbPoints[facePointIndx[plane,0]]
		pointVecs = points - planePoint
		distance = numpy.dot(pointVecs, norm)
		pointsIndx[distance < 0] = False #if distance is < 0 then point is on
outside of box
	return pointsIndx

def transformPoints(transform, points):
	'''
	transform must be a 4x4 matrix/array
	points must be a Nx3 or Nx4 matrix/array
	'''
	points = numpy.array(points)
	ndim = points.ndim
	if ndim == 1:
		points = points[numpy.newaxis,:]
	A = numpy.matrix(transform)

	B = numpy.ones((points.shape[0], 4))
	B[:,:3] = points
	B = numpy.matrix(B).T

	transformedPoints = ((A * B).T).getA()
	transformedPoints = transformedPoints[:,:3]
	if ndim == 1:
		transformedPoints = transformedPoints.squeeze()
	return transformedPoints

def translation(*args):
	transMat = numpy.matrix(numpy.identity(4, dtype=numpy.double))
```

```
	if len(args) == 3:
		transMat[0,3] = args[0]
		transMat[1,3] = args[1]
		transMat[2,3] = args[2]
	else:
		transMat[0,3] = args[0][0]
		transMat[1,3] = args[0][1]
		transMat[2,3] = args[0][2]
	return transMat

def rotation(angle, axis,N=3):
	'''
	rotationMatrix generates a NxN rotation matrix (default 3x3)
	axis is of form [x,y,z]
	angle is in radians
	equation from
http://www.euclideanspace.com/maths/geometry/rotations/conversions/angleToMatrix/inde
x.htm
	'''
	c = math.cos(angle)
	s = math.sin(angle)
	t = 1-c
	axis = numpy.array(axis, dtype=numpy.double)
	axis /= math.sqrt(numpy.dot(axis,axis.conj()))	#normalize axis
	x = axis[0]
	y = axis[1]
	z = axis[2]
	rotMat = numpy.matrix(numpy.identity(N))
	rotMat[0,0] = t*x*x + c
	rotMat[0,1] = t*x*y - z*s
	rotMat[0,2] = t*x*z + y*s
	rotMat[1,0] = t*x*y + z*s
	rotMat[1,1] = t*y*y + c
	rotMat[1,2] = t*y*z - x*s
	rotMat[2,0] = t*x*z - y*s
	rotMat[2,1] = t*y*z + x*s
	rotMat[2,2] = t*z*z + c
	return rotMat

def decomposeMatrix(m):
	m = numpy.matrix(m)
	#force m to be 4x4 matrix
	if m.shape != (4,4):
		mt = numpy.matrix(numpy.identity(4))
		mt[:m.shape[0],:m.shape[1]] = m
		m = mt
	sx = numpy.linalg.norm(m[0:3,0])
	sy = numpy.linalg.norm(m[0:3,1])
	sz = numpy.linalg.norm(m[0:3,2])

	#remove scaling
	srev = scaling([sx, sy, sz], N=4).I
	m *= srev

	return m[:3,:3], m[0,3], m[1,3], m[2,3], sx, sy, sz
```

```
def axisAngleFromMatrix(rotMatrix, angleInDegrees=False):
	'''
	gives axis angle rotation for the passed rotation matrix
	return axis, angle
	axis is array of length 3
	equation from
http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToAngle/inde
x.htm
	'''

	rotMatrix, tx, ty, tz, sx, sy,sz = decomposeMatrix(rotMatrix)

	m = numpy.matrix(rotMatrix)
	if m.shape[0] < 3 or m.shape[1] < 3:
		raise Exception('Incorrect matrix size')
	angle = math.acos(( m[0,0] + m[1,1] + m[2,2] - 1)/2)
	if angleInDegrees:
		angle = math.degrees(angle)
	if angle != 0:
		axis = [(m[2,1] - m[1,2])/math.sqrt((m[2,1] - m[1,2])**2+(m[0,2] -
m[2,0])**2+(m[1,0] - m[0,1])**2),
				(m[0,2] - m[2,0])/math.sqrt((m[2,1] - m[1,2])**2+(m[0,2] -
m[2,0])**2+(m[1,0] - m[0,1])**2),
				(m[1,0] - m[0,1])/math.sqrt((m[2,1] - m[1,2])**2+(m[0,2] -
m[2,0])**2+(m[1,0] - m[0,1])**2)]
	else:
		axis = [1.0, 0.0, 0.0]
	return axis, angle

def coordinateSystemConversionMatrix(currentAxes, newAxes, N=3):
	'''
	This system generates a NxN matrix defines the rigid rotation transform
	required to change currentAxis into newAxis, this transform can be used
	on points to transform them from the currentAxis coordinate system to
	the newAxis coordinate system

	equation from http://www.j3d.org/matrix_faq/matrfaq_latest.html
	Question 'Q40. How do I use matrices to convert one coordinate system to
another?'
	'''
	currentAxes = numpy.matrix(currentAxes)
	newAxes = numpy.matrix(newAxes)

	M = newAxes * currentAxes.I
	if N is not None or (M.shape[0] != N and M.shape[1] != N ):
		Mtmp = numpy.identity(N)
		for i in xrange(M.shape[0]):
			if i >= N:
				break
			for j in xrange(M.shape[1]):
				if j >= N:
					break
				Mtmp[i,j] = M[i,j]
```

```
            M = numpy.matrix(Mtmp)
        return M

def scaling(scale, N=3):
        '''
        scale can be a single number or an array of three values [x, y, z]
        '''
        M = numpy.matrix(numpy.identity(N, dtype=numpy.double))
        scale = numpy.array(scale)
        if scale.ndim == 0:
            M[0,0] = scale
            M[1,1] = scale
            M[2,2] = scale
        elif scale.ndim == 1:
            M[0,0] = scale[0]
            M[1,1] = scale[1]
            M[2,2] = scale[2]
        return M

def findDistanceToNearestNeighbor(point, targetPointList):
        minDistance = numpy.min(numpy.sqrt(numpy.sum((point-targetPointList)**2,
axis=1)))
        return minDistance

def findNearestNeighbor(point, targetPointList):
        minDistance = None
        for targetPoint in targetPointList:
            dist = numpy.linalg.norm(targetPoint-point)
            if minDistance is None or dist < minDistance:
                minDistance = dist
                nearestNeighbor = targetPoint
        return nearestNeighbor
```

openGLWidget.py

```
# -*- coding:utf-8 -*-
"""
Created on Aug 25, 2011

@author: grant
"""
import sys, math, numpy
from openGLUtils import Camera, OpenGlLight
from PySide import QtCore, QtGui, QtOpenGL
from OpenGL import GL, GLU
from Joint import Joint
import TriModel

class GLWidget(QtOpenGL.QGLWidget):
	def __init__(self, parent=None):
		QtOpenGL.QGLWidget.__init__(self, parent)
		self.setFocusPolicy(QtCore.Qt.ClickFocus)
		self.keyStates = {}
		self.keyStates[16777249] = False
		self.camera = Camera()
		self.worldJoint = None

	self.setSizePolicy(QtGui.QSizePolicy.Policy.Expanding,QtGui.QSizePolicy.Policy
.Expanding)
		self.initializeOpenGlLights()
		self.colorDrivenMaterial = None
		self.prespective = True
		self.useCallLists=True

	def clearScene(self):
		if self.worldJoint is not None:
			#remove current scene
			self.worldJoint.childJoints = []
			for model in self.worldJoint.models:
				if model.name[:5] != 'axis_':
					self.worldJoint.models.remove(model)

	def setColorDrivenMaterial(self, enabled=False):
		self.colorDrivenMaterial = enabled
		if enabled:
			GL.glEnable(GL.GL_COLOR_MATERIAL)
		else:
			GL.glDisable(GL.GL_COLOR_MATERIAL)

	def initializeOpenGlLights(self):
		self.lights = []
		for i in xrange(8):
			self.lights.append(OpenGlLight(i))
		self.lights[0].enabled = True
		self.lights[0].diffuseColor = [1.0,1.0,1.0,1.0]
		self.lights[0].specularColor = [1.0,1.0,1.0,1.0]
		self.lights[0].position = [1.0,0.5,1.0,0.0]
```

```
	def minimumSizeHint(self):
		#inherited from QGLWidget, sets minimum size
		return QtCore.QSize(50, 50)

	def sizeHint(self):
		#inherited from QGLWidget, sets start size
		return QtCore.QSize(1600, 1600)

	def initializeGL(self):
		self.worldJoint = Joint(name='WorldJoint',showAxis = True,
axisScale=0.7)

		GL.glClearColor(0.0,0.0,0.0,1.0)
#		GL.glClearColor(1.0,1.0,1.0,1.0)
		GL.glClearDepth(1);
		GL.glClear(GL.GL_COLOR_BUFFER_BIT)
		GL.glEnable(GL.GL_BLEND)  #enable blending
		GL.glBlendFunc(GL.GL_SRC_ALPHA,GL.GL_ONE_MINUS_SRC_ALPHA)  #set blend
function
		GL.glEnable (GL.GL_DEPTH_TEST);
		GL.glEnable(GL.GL_LIGHTING) #enable the lighting
#		GL.glEnable(GL.GL_LIGHT0) #enable LIGHT0, our Diffuse Light
		for light in self.lights:
			light.updateOpenGl()
		GL.glShadeModel(GL.GL_SMOOTH) #set the shader to smooth shader
		GL.glEnable(GL.GL_NORMALIZE)
		GL.glEnable(GL.GL_TEXTURE_2D)
		vMin, vMax = self.getBoundingBox()
		self.camera.setView(vMin, vMax, view='ortho1')

	def getBoundingBox(self):
		vMin = None
		vMax = None

		if self.worldJoint is not None:
			vMin, vMax = self.worldJoint.getBoundingBox()

		if vMin is None:
			vMin = numpy.array([0.0,0.0,0.0])
		if vMax is None:
			vMax = numpy.array([1.0,1.0,1.0])
		return vMin, vMax

	def paintGL(self):
		GL.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT)
		GL.glLoadIdentity()
		self.camera.openGLTransform()
		if self.worldJoint is not None:
			self.worldJoint.OpenGLPaint(self.colorDrivenMaterial,
self.useCallLists)

	def resizeGL(self, width, height):
		if width == 0 or height == 0:
			return
		#Set our viewport to the size of our window
```

```
            GL.glViewport(0, 0, width, height)
            #Switch to the projection matrix so that we can manipulate how our scene
is viewed
            GL.glMatrixMode(GL.GL_PROJECTION)
            #Reset the projection matrix to the identity matrix so that we don't get
any artifacts (cleaning up)
            GL.glLoadIdentity()
            #Set the Field of view angle (in degrees), the aspect ratio of our
window, and the new and far planes
            vMin, vMax = self.getBoundingBox()
            near = 1.0
            far = 100.0
            aspectRatio = float(width)/float(height)
            if vMax is not None and vMin is not None:
                  d = vMax-vMin
                  d = math.sqrt(d[0]**2+d[1]**2+d[2]**2)
                  far = d*3.0
#           print 'Far plane: ',far
            if self.prespective:
                  GLU.gluPerspective(60, aspectRatio, near, far)
            else:
                  h = (vMax[1]-vMin[1])*aspectRatio
                  center = (vMax[1]-vMin[1])/2
                  GL.glOrtho( center-h/2, center+h/2, vMin[1], vMax[1], near, far)

            #Switch back to the model view matrix, so that we can start drawing
shapes correctly
            GL.glMatrixMode(GL.GL_MODELVIEW)

      def keyPressEvent(self, event):
            global camAngle
            key = event.key()
            txt = event.text()
            self.keyStates[key] = True
            if key == 16777235: #Up
                  self.camera.rotateAroundAxis(math.pi/180.0,[1.0,0.0,0.0])
            elif key == 16777237:     #Down
                  self.camera.rotateAroundAxis(-math.pi/180.0,[1.0,0.0,0.0])
            elif key == 16777234:     #Left
                  self.camera.rotateAroundAxis(math.pi/180.0,[0.0,1.0,0.0])
            elif key == 16777236:     #Right
                  self.camera.rotateAroundAxis(-math.pi/180.0,[0.0,1.0,0.0])
            elif key == 16777249:     #crtl
                  pass
            elif txt=='s':
                  pass
            else:
#                 print key, event.text()
                  pass

            #force redraw of scene
            self.updateGL()

      def keyReleaseEvent(self,event):
            key = event.key()
```

```
                self.keyStates[key] = False

        def mousePressEvent(self, event):
                self.lastPos = event.pos()
        def mouseMoveEvent(self, event):
                dx = event.x() - self.lastPos.x()
                dy = event.y() - self.lastPos.y()
                if (event.buttons() & QtCore.Qt.LeftButton) and
(self.keyStates[16777249] == False):
                        self.camera.rotateAroundAxis(math.radians(-1*dy), [1,0,0])
                        self.camera.rotateAroundAxis(math.radians(-1*dx), [0,1,0])
                        #force redraw of scene
                        self.updateGL()
                elif (event.buttons() & QtCore.Qt.MidButton) or ((event.buttons() &
QtCore.Qt.LeftButton) and (self.keyStates[16777249])):
                        #Pan if middle mouse button is pressed or left button is pressed
and control is pressed
                        #screen height in units is same as distance from camLoc to
camFocus if view angle is 60 degrees
                        unitsPerPixel = numpy.linalg.norm(self.camera.camLoc-
self.camera.camFocus)/self.height()
                        self.camera.pan([-dx*unitsPerPixel, dy*unitsPerPixel,0.0])
                        #force redraw of scene
                        self.updateGL()
                self.lastPos = event.pos()

        def wheelEvent(self, event):
                d=1.0+0.1*abs(event.delta())/120.0
                if event.delta()>0.0:
                        self.camera.zoom(1/d)
                else:
                        self.camera.zoom(d)
                #force redraw of scene
                self.updateGL()

class Window(QtGui.QWidget):
        def __init__(self, parent=None):
                QtGui.QWidget.__init__(self, parent)
                self.glWidget = GLWidget()
                createDefaultScene(self.glWidget.worldJoint)
                mainLayout = QtGui.QHBoxLayout()
                mainLayout.addWidget(self.glWidget)
                self.setLayout(mainLayout)
                self.setWindowTitle(self.tr("Qt openGL Viewer"))

def createDefaultScene(worldJoint):
        #create and connect joints
        joint1_2 = Joint([0.5,0.5,3.5], parentJoint=worldJoint, showAxis = True,
axisScale=0.5)
        joint2_3 = Joint([0.5,0.5,7.5], parentJoint=joint1_2, showAxis = True,
axisScale=0.5)

        #create bone models
        TriModel.createRectangularSolid([1,1,3],[0,0,0],joint=worldJoint)
        TriModel.createRectangularSolid([1,1,3],[0,0,4],joint=joint1_2)
```

```
        TriModel.createRectangularSolid([1,1,3],[0,0,8],joint=joint2_3)

if __name__ == '__main__':
        app = QtGui.QApplication(sys.argv)
        window = Window()
        window.show()
        sys.exit(app.exec_())
```

mlabraw.py

```
#!/usr/bin/env python

""" A quick and extremely dirty hack to wrap matlabpipe/matlabcom as if they
were mlabraw.

Author: Dani Valevski <daniva@gmail.com>
License: MIT
"""
import sys

is_win = sys.platform == 'win32'
if is_win:
	from matlabcom import MatlabCom as MatlabConnection
	from matlabcom import MatlabError as error
else:
	from matlabpipe import MatlabPipe as MatlabConnection
	from matlabpipe import MatlabError as error

try:
	import settings
except:
	class settings:
		MATLAB_PATH = 'guess'

def open(arg):
	if is_win:
		ret = MatlabConnection()
		ret.open()
	else:
		if settings.MATLAB_PATH != 'guess':
			matlab_path = settings.MATLAB_PATH + '/bin/matlab'
		else:
			matlab_path = 'guess'
		try:
			ret = MatlabConnection(matlab_path)
			ret.open()
		except:
			print 'Could not open matlab, is it in %s?' % matlab_path
	return ret

def close(matlab):
	matlab.close()

def eval(matlab, exp, log=False):
	if log or is_win:
		matlab.eval(exp)
	else:
		matlab.eval(exp, print_expression=False, on_new_output=None)
	return ''

def get(matlab, var_name):
	return matlab.get(var_name)
```

```
def put(matlab, var_name, val):
       matlab.put({var_name : val})
```

OpenGLGUI.py

```
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'MainWindow.ui'
#
# Created: Thu Apr 12 00:55:25 2012
#      by: pyside-uic 0.2.13 running on PySide 1.1.0
#
# WARNING! All changes made in this file will be lost!

from PySide import QtCore, QtGui

class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(1508, 828)
        self.centralwidget = QtGui.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.verticalLayout_6 = QtGui.QVBoxLayout(self.centralwidget)
        self.verticalLayout_6.setObjectName("verticalLayout_6")
        self.splitter = QtGui.QSplitter(self.centralwidget)
        self.splitter.setOrientation(QtCore.Qt.Horizontal)
        self.splitter.setObjectName("splitter")
        self.viewTabs = QtGui.QTabWidget(self.splitter)
        self.viewTabs.setMovable(True)
        self.viewTabs.setObjectName("viewTabs")
        self.openGlTab = QtGui.QWidget()
        self.openGlTab.setObjectName("openGlTab")
        self.verticalLayout_5 = QtGui.QVBoxLayout(self.openGlTab)
        self.verticalLayout_5.setObjectName("verticalLayout_5")
        self.horizontalLayout_2 = QtGui.QHBoxLayout()
        self.horizontalLayout_2.setObjectName("horizontalLayout_2")
        self.label_9 = QtGui.QLabel(self.openGlTab)
        self.label_9.setObjectName("label_9")
        self.horizontalLayout_2.addWidget(self.label_9)
        self.setView = QtGui.QComboBox(self.openGlTab)
        self.setView.setObjectName("setView")
        self.setView.addItem("")
        self.setView.setItemText(0, "")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.setView.addItem("")
        self.horizontalLayout_2.addWidget(self.setView)
```

```
        spacerItem = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.horizontalLayout_2.addItem(spacerItem)
        self.automate = QtGui.QPushButton(self.openGlTab)
        self.automate.setObjectName("automate")
        self.horizontalLayout_2.addWidget(self.automate)
        self.loadModel = QtGui.QPushButton(self.openGlTab)
        self.loadModel.setObjectName("loadModel")
        self.horizontalLayout_2.addWidget(self.loadModel)
        self.loadAtlas = QtGui.QPushButton(self.openGlTab)
        self.loadAtlas.setObjectName("loadAtlas")
        self.horizontalLayout_2.addWidget(self.loadAtlas)
        self.defaultScene = QtGui.QPushButton(self.openGlTab)
        self.defaultScene.setObjectName("defaultScene")
        self.horizontalLayout_2.addWidget(self.defaultScene)
        self.clearScene = QtGui.QPushButton(self.openGlTab)
        self.clearScene.setObjectName("clearScene")
        self.horizontalLayout_2.addWidget(self.clearScene)
        self.GrabScreen = QtGui.QPushButton(self.openGlTab)
        self.GrabScreen.setObjectName("GrabScreen")
        self.horizontalLayout_2.addWidget(self.GrabScreen)
        self.verticalLayout_5.addLayout(self.horizontalLayout_2)
        self.viewTabs.addTab(self.openGlTab, "")
        self.tab_7 = QtGui.QWidget()
        self.tab_7.setObjectName("tab_7")
        self.verticalLayout_9 = QtGui.QVBoxLayout(self.tab_7)
        self.verticalLayout_9.setObjectName("verticalLayout_9")
        self.horizontalLayout_10 = QtGui.QHBoxLayout()
        self.horizontalLayout_10.setObjectName("horizontalLayout_10")
        self.label_74 = QtGui.QLabel(self.tab_7)
        font = QtGui.QFont()
        font.setPointSize(12)
        font.setWeight(75)
        font.setBold(True)
        self.label_74.setFont(font)
        self.label_74.setObjectName("label_74")
        self.horizontalLayout_10.addWidget(self.label_74)
        spacerItem1 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.horizontalLayout_10.addItem(spacerItem1)
        self.copyTable = QtGui.QPushButton(self.tab_7)
        self.copyTable.setObjectName("copyTable")
        self.horizontalLayout_10.addWidget(self.copyTable)
        self.saveCSV = QtGui.QPushButton(self.tab_7)
        self.saveCSV.setObjectName("saveCSV")
        self.horizontalLayout_10.addWidget(self.saveCSV)
        self.verticalLayout_9.addLayout(self.horizontalLayout_10)
        self.nmTable = QtGui.QTableWidget(self.tab_7)
        self.nmTable.setEditTriggers(QtGui.QAbstractItemView.NoEditTriggers)
        self.nmTable.setObjectName("nmTable")
        self.nmTable.setColumnCount(0)
        self.nmTable.setRowCount(0)
        self.verticalLayout_9.addWidget(self.nmTable)
        self.viewTabs.addTab(self.tab_7, "")
        self.verticalLayout_6.addWidget(self.splitter)
```

```
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtGui.QMenuBar(MainWindow)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 1508, 21))
        self.menubar.setObjectName("menubar")
        MainWindow.setMenuBar(self.menubar)
        self.dockItemList = QtGui.QDockWidget(MainWindow)
        self.dockItemList.setFloating(False)

self.dockItemList.setFeatures(QtGui.QDockWidget.DockWidgetFloatable|QtGui.QDockWidget
.DockWidgetMovable)
        self.dockItemList.setObjectName("dockItemList")
        self.dockWidgetContents_2 = QtGui.QWidget()
        self.dockWidgetContents_2.setObjectName("dockWidgetContents_2")
        self.gridLayout = QtGui.QGridLayout(self.dockWidgetContents_2)
        self.gridLayout.setObjectName("gridLayout")
        self.itemTreeList = QtGui.QTreeWidget(self.dockWidgetContents_2)
        sizePolicy = QtGui.QSizePolicy(QtGui.QSizePolicy.Preferred,
QtGui.QSizePolicy.Expanding)
        sizePolicy.setHorizontalStretch(0)
        sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.itemTreeList.sizePolicy().hasHeightForWidth())
        self.itemTreeList.setSizePolicy(sizePolicy)
        self.itemTreeList.setToolTip("")
        self.itemTreeList.setColumnCount(1)
        self.itemTreeList.setObjectName("itemTreeList")
        self.itemTreeList.headerItem().setText(0, "1")
        self.itemTreeList.header().setVisible(False)
        self.gridLayout.addWidget(self.itemTreeList, 0, 0, 1, 1)
        self.dockItemList.setWidget(self.dockWidgetContents_2)
        MainWindow.addDockWidget(QtCore.Qt.DockWidgetArea(8), self.dockItemList)
        self.dockCamera = QtGui.QDockWidget(MainWindow)
        self.dockCamera.setFeatures(QtGui.QDockWidget.AllDockWidgetFeatures)
        self.dockCamera.setObjectName("dockCamera")
        self.dockWidgetContents_8 = QtGui.QWidget()
        self.dockWidgetContents_8.setObjectName("dockWidgetContents_8")
        self.verticalLayout_8 = QtGui.QVBoxLayout(self.dockWidgetContents_8)
        self.verticalLayout_8.setObjectName("verticalLayout_8")
        self.camUpdate = QtGui.QPushButton(self.dockWidgetContents_8)
        self.camUpdate.setObjectName("camUpdate")
        self.verticalLayout_8.addWidget(self.camUpdate)
        self.gridLayout_10 = QtGui.QGridLayout()
        self.gridLayout_10.setObjectName("gridLayout_10")
        self.label_39 = QtGui.QLabel(self.dockWidgetContents_8)

self.label_39.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_39.setObjectName("label_39")
        self.gridLayout_10.addWidget(self.label_39, 2, 0, 1, 1)
        self.label_40 = QtGui.QLabel(self.dockWidgetContents_8)

self.label_40.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_40.setObjectName("label_40")
        self.gridLayout_10.addWidget(self.label_40, 3, 0, 1, 1)
```

```
        self.label_41 = QtGui.QLabel(self.dockWidgetContents_8)

self.label_41.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_41.setObjectName("label_41")
        self.gridLayout_10.addWidget(self.label_41, 4, 0, 1, 1)
        self.camXPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camXPos.setEnabled(True)
        self.camXPos.setWrapping(False)
        self.camXPos.setReadOnly(False)
        self.camXPos.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camXPos.setDecimals(10)
        self.camXPos.setMinimum(-1000000.0)
        self.camXPos.setMaximum(1000000.0)
        self.camXPos.setObjectName("camXPos")
        self.gridLayout_10.addWidget(self.camXPos, 2, 1, 1, 1)
        self.camYPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camYPos.setEnabled(True)
        self.camYPos.setReadOnly(False)
        self.camYPos.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camYPos.setDecimals(10)
        self.camYPos.setMinimum(-1000000.0)
        self.camYPos.setMaximum(1000000.0)
        self.camYPos.setObjectName("camYPos")
        self.gridLayout_10.addWidget(self.camYPos, 3, 1, 1, 1)
        self.camZPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camZPos.setEnabled(True)
        self.camZPos.setReadOnly(False)
        self.camZPos.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camZPos.setDecimals(10)
        self.camZPos.setMinimum(-1000000.0)
        self.camZPos.setMaximum(1000000.0)
        self.camZPos.setObjectName("camZPos")
        self.gridLayout_10.addWidget(self.camZPos, 4, 1, 1, 1)
        self.label_42 = QtGui.QLabel(self.dockWidgetContents_8)
        self.label_42.setObjectName("label_42")
        self.gridLayout_10.addWidget(self.label_42, 1, 1, 1, 1)
        spacerItem2 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_10.addItem(spacerItem2, 2, 2, 1, 1)
        self.verticalLayout_8.addLayout(self.gridLayout_10)
        self.line_9 = QtGui.QFrame(self.dockWidgetContents_8)
        self.line_9.setFrameShape(QtGui.QFrame.HLine)
        self.line_9.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_9.setObjectName("line_9")
        self.verticalLayout_8.addWidget(self.line_9)
        self.gridLayout_11 = QtGui.QGridLayout()
        self.gridLayout_11.setObjectName("gridLayout_11")
        self.label_43 = QtGui.QLabel(self.dockWidgetContents_8)

self.label_43.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_43.setObjectName("label_43")
        self.gridLayout_11.addWidget(self.label_43, 1, 0, 1, 1)
        self.label_44 = QtGui.QLabel(self.dockWidgetContents_8)
```

```
self.label_44.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_44.setObjectName("label_44")
        self.gridLayout_11.addWidget(self.label_44, 2, 0, 1, 1)
        self.label_45 = QtGui.QLabel(self.dockWidgetContents_8)

self.label_45.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_45.setObjectName("label_45")
        self.gridLayout_11.addWidget(self.label_45, 3, 0, 1, 1)
        self.camXLookAt = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camXLookAt.setEnabled(True)
        self.camXLookAt.setWrapping(False)
        self.camXLookAt.setReadOnly(False)
        self.camXLookAt.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camXLookAt.setDecimals(10)
        self.camXLookAt.setMinimum(-1000000.0)
        self.camXLookAt.setMaximum(1000000.0)
        self.camXLookAt.setObjectName("camXLookAt")
        self.gridLayout_11.addWidget(self.camXLookAt, 1, 1, 1, 1)
        self.camYLookAt = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camYLookAt.setEnabled(True)
        self.camYLookAt.setReadOnly(False)
        self.camYLookAt.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camYLookAt.setDecimals(10)
        self.camYLookAt.setMinimum(-1000000.0)
        self.camYLookAt.setMaximum(1000000.0)
        self.camYLookAt.setObjectName("camYLookAt")
        self.gridLayout_11.addWidget(self.camYLookAt, 2, 1, 1, 1)
        self.camZLookAt = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camZLookAt.setEnabled(True)
        self.camZLookAt.setReadOnly(False)
        self.camZLookAt.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camZLookAt.setDecimals(10)
        self.camZLookAt.setMinimum(-1000000.0)
        self.camZLookAt.setMaximum(1000000.0)
        self.camZLookAt.setObjectName("camZLookAt")
        self.gridLayout_11.addWidget(self.camZLookAt, 3, 1, 1, 1)
        self.label_46 = QtGui.QLabel(self.dockWidgetContents_8)
        self.label_46.setObjectName("label_46")
        self.gridLayout_11.addWidget(self.label_46, 0, 1, 1, 1)
        spacerItem3 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_11.addItem(spacerItem3, 1, 2, 1, 1)
        self.verticalLayout_8.addLayout(self.gridLayout_11)
        self.line_10 = QtGui.QFrame(self.dockWidgetContents_8)
        self.line_10.setFrameShape(QtGui.QFrame.HLine)
        self.line_10.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_10.setObjectName("line_10")
        self.verticalLayout_8.addWidget(self.line_10)
        self.gridLayout_12 = QtGui.QGridLayout()
        self.gridLayout_12.setObjectName("gridLayout_12")
        self.label_47 = QtGui.QLabel(self.dockWidgetContents_8)
```

```
self.label_47.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_47.setObjectName("label_47")
        self.gridLayout_12.addWidget(self.label_47, 1, 0, 1, 1)
        self.label_48 = QtGui.QLabel(self.dockWidgetContents_8)

self.label_48.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_48.setObjectName("label_48")
        self.gridLayout_12.addWidget(self.label_48, 2, 0, 1, 1)
        self.label_49 = QtGui.QLabel(self.dockWidgetContents_8)

self.label_49.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_49.setObjectName("label_49")
        self.gridLayout_12.addWidget(self.label_49, 3, 0, 1, 1)
        self.camXUp = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camXUp.setEnabled(True)
        self.camXUp.setWrapping(False)
        self.camXUp.setReadOnly(False)
        self.camXUp.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camXUp.setDecimals(10)
        self.camXUp.setMinimum(-1000000.0)
        self.camXUp.setMaximum(1000000.0)
        self.camXUp.setObjectName("camXUp")
        self.gridLayout_12.addWidget(self.camXUp, 1, 1, 1, 1)
        self.camYUp = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camYUp.setEnabled(True)
        self.camYUp.setReadOnly(False)
        self.camYUp.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camYUp.setDecimals(10)
        self.camYUp.setMinimum(-1000000.0)
        self.camYUp.setMaximum(1000000.0)
        self.camYUp.setObjectName("camYUp")
        self.gridLayout_12.addWidget(self.camYUp, 2, 1, 1, 1)
        self.camZUp = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camZUp.setEnabled(True)
        self.camZUp.setReadOnly(False)
        self.camZUp.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camZUp.setDecimals(10)
        self.camZUp.setMinimum(-1000000.0)
        self.camZUp.setMaximum(1000000.0)
        self.camZUp.setObjectName("camZUp")
        self.gridLayout_12.addWidget(self.camZUp, 3, 1, 1, 1)
        self.label_50 = QtGui.QLabel(self.dockWidgetContents_8)
        self.label_50.setObjectName("label_50")
        self.gridLayout_12.addWidget(self.label_50, 0, 1, 1, 1)
        spacerItem4 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_12.addItem(spacerItem4, 1, 2, 1, 1)
        self.verticalLayout_8.addLayout(self.gridLayout_12)
        self.line_8 = QtGui.QFrame(self.dockWidgetContents_8)
        self.line_8.setFrameShape(QtGui.QFrame.HLine)
        self.line_8.setFrameShadow(QtGui.QFrame.Sunken)
```

```
        self.line_8.setObjectName("line_8")
        self.verticalLayout_8.addWidget(self.line_8)
        self.label_59 = QtGui.QLabel(self.dockWidgetContents_8)
        self.label_59.setObjectName("label_59")
        self.verticalLayout_8.addWidget(self.label_59)
        self.gridLayout_13 = QtGui.QGridLayout()
        self.gridLayout_13.setObjectName("gridLayout_13")
        self.label_55 = QtGui.QLabel(self.dockWidgetContents_8)
        self.label_55.setObjectName("label_55")
        self.gridLayout_13.addWidget(self.label_55, 0, 0, 1, 1)
        self.label_56 = QtGui.QLabel(self.dockWidgetContents_8)
        self.label_56.setObjectName("label_56")
        self.gridLayout_13.addWidget(self.label_56, 1, 0, 1, 1)
        self.label_57 = QtGui.QLabel(self.dockWidgetContents_8)
        self.label_57.setObjectName("label_57")
        self.gridLayout_13.addWidget(self.label_57, 2, 0, 1, 1)
        self.camQuatW = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camQuatW.setReadOnly(False)
        self.camQuatW.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camQuatW.setDecimals(10)
        self.camQuatW.setMinimum(-1000000.0)
        self.camQuatW.setMaximum(1000000.0)
        self.camQuatW.setObjectName("camQuatW")
        self.gridLayout_13.addWidget(self.camQuatW, 0, 1, 1, 1)
        spacerItem5 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_13.addItem(spacerItem5, 0, 2, 1, 1)
        self.camQuatX = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camQuatX.setReadOnly(False)
        self.camQuatX.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camQuatX.setDecimals(10)
        self.camQuatX.setMinimum(-1000000.0)
        self.camQuatX.setMaximum(1000000.0)
        self.camQuatX.setObjectName("camQuatX")
        self.gridLayout_13.addWidget(self.camQuatX, 1, 1, 1, 1)
        self.camQuatY = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camQuatY.setReadOnly(False)
        self.camQuatY.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camQuatY.setDecimals(10)
        self.camQuatY.setMinimum(-1000000.0)
        self.camQuatY.setMaximum(1000000.0)
        self.camQuatY.setObjectName("camQuatY")
        self.gridLayout_13.addWidget(self.camQuatY, 2, 1, 1, 1)
        self.camQuatZ = QtGui.QDoubleSpinBox(self.dockWidgetContents_8)
        self.camQuatZ.setReadOnly(False)
        self.camQuatZ.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.camQuatZ.setDecimals(10)
        self.camQuatZ.setMinimum(-1000000.0)
        self.camQuatZ.setMaximum(1000000.0)
        self.camQuatZ.setObjectName("camQuatZ")
        self.gridLayout_13.addWidget(self.camQuatZ, 3, 1, 1, 1)
        self.label_58 = QtGui.QLabel(self.dockWidgetContents_8)
        self.label_58.setObjectName("label_58")
        self.gridLayout_13.addWidget(self.label_58, 3, 0, 1, 1)
        self.verticalLayout_8.addLayout(self.gridLayout_13)
```

```
        self.line_11 = QtGui.QFrame(self.dockWidgetContents_8)
        self.line_11.setFrameShape(QtGui.QFrame.HLine)
        self.line_11.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_11.setObjectName("line_11")
        self.verticalLayout_8.addWidget(self.line_11)
        self.camFunction = QtGui.QLineEdit(self.dockWidgetContents_8)
        self.camFunction.setObjectName("camFunction")
        self.verticalLayout_8.addWidget(self.camFunction)
        spacerItem6 = QtGui.QSpacerItem(20, 40, QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Expanding)
        self.verticalLayout_8.addItem(spacerItem6)
        self.dockCamera.setWidget(self.dockWidgetContents_8)
        MainWindow.addDockWidget(QtCore.Qt.DockWidgetArea(8), self.dockCamera)
        self.dockLights = QtGui.QDockWidget(MainWindow)
        self.dockLights.setFeatures(QtGui.QDockWidget.AllDockWidgetFeatures)
        self.dockLights.setObjectName("dockLights")
        self.dockWidgetContents_9 = QtGui.QWidget()
        self.dockWidgetContents_9.setObjectName("dockWidgetContents_9")
        self.verticalLayout_7 = QtGui.QVBoxLayout(self.dockWidgetContents_9)
        self.verticalLayout_7.setObjectName("verticalLayout_7")
        self.horizontalLayout_3 = QtGui.QHBoxLayout()
        self.horizontalLayout_3.setObjectName("horizontalLayout_3")
        self.label = QtGui.QLabel(self.dockWidgetContents_9)
        self.label.setObjectName("label")
        self.horizontalLayout_3.addWidget(self.label)
        self.lightNum = QtGui.QSpinBox(self.dockWidgetContents_9)
        self.lightNum.setMaximum(7)
        self.lightNum.setObjectName("lightNum")
        self.horizontalLayout_3.addWidget(self.lightNum)
        spacerItem7 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.horizontalLayout_3.addItem(spacerItem7)
        self.verticalLayout_7.addLayout(self.horizontalLayout_3)
        self.lightEnable = QtGui.QCheckBox(self.dockWidgetContents_9)
        self.lightEnable.setObjectName("lightEnable")
        self.verticalLayout_7.addWidget(self.lightEnable)
        self.line = QtGui.QFrame(self.dockWidgetContents_9)
        self.line.setFrameShape(QtGui.QFrame.HLine)
        self.line.setFrameShadow(QtGui.QFrame.Sunken)
        self.line.setObjectName("line")
        self.verticalLayout_7.addWidget(self.line)
        self.gridLayout_2 = QtGui.QGridLayout()
        self.gridLayout_2.setObjectName("gridLayout_2")
        self.label_3 = QtGui.QLabel(self.dockWidgetContents_9)
        self.label_3.setAlignment(QtCore.Qt.AlignCenter)
        self.label_3.setObjectName("label_3")
        self.gridLayout_2.addWidget(self.label_3, 2, 0, 1, 1)
        self.label_5 = QtGui.QLabel(self.dockWidgetContents_9)
        self.label_5.setEnabled(False)
        self.label_5.setAlignment(QtCore.Qt.AlignCenter)
        self.label_5.setObjectName("label_5")
        self.gridLayout_2.addWidget(self.label_5, 4, 0, 1, 1)
        self.chooseDiffuseColorCam = QtGui.QPushButton(self.dockWidgetContents_9)
        self.chooseDiffuseColorCam.setText("")
        self.chooseDiffuseColorCam.setObjectName("chooseDiffuseColorCam")
```

```
        self.gridLayout_2.addWidget(self.chooseDiffuseColorCam, 2, 1, 1, 1)
        self.chooseEmissiveColorCam = QtGui.QPushButton(self.dockWidgetContents_9)
        self.chooseEmissiveColorCam.setEnabled(False)
        self.chooseEmissiveColorCam.setText("")
        self.chooseEmissiveColorCam.setObjectName("chooseEmissiveColorCam")
        self.gridLayout_2.addWidget(self.chooseEmissiveColorCam, 4, 1, 1, 1)
        spacerItem8 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_2.addItem(spacerItem8, 1, 2, 1, 1)
        self.label_11 = QtGui.QLabel(self.dockWidgetContents_9)
        font = QtGui.QFont()
        font.setPointSize(10)
        font.setWeight(75)
        font.setBold(True)
        self.label_11.setFont(font)
        self.label_11.setAlignment(QtCore.Qt.AlignCenter)
        self.label_11.setObjectName("label_11")
        self.gridLayout_2.addWidget(self.label_11, 0, 0, 1, 1)
        self.chooseSpecularColorCam = QtGui.QPushButton(self.dockWidgetContents_9)
        self.chooseSpecularColorCam.setEnabled(True)
        sizePolicy = QtGui.QSizePolicy(QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Fixed)
        sizePolicy.setHorizontalStretch(0)
        sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.chooseSpecularColorCam.sizePolicy().hasHeightForWid
th())
        self.chooseSpecularColorCam.setSizePolicy(sizePolicy)
        self.chooseSpecularColorCam.setText("")
        self.chooseSpecularColorCam.setObjectName("chooseSpecularColorCam")
        self.gridLayout_2.addWidget(self.chooseSpecularColorCam, 3, 1, 1, 1)
        self.label_2 = QtGui.QLabel(self.dockWidgetContents_9)
        self.label_2.setAlignment(QtCore.Qt.AlignCenter)
        self.label_2.setObjectName("label_2")
        self.gridLayout_2.addWidget(self.label_2, 3, 0, 1, 1)
        self.label_4 = QtGui.QLabel(self.dockWidgetContents_9)
        self.label_4.setAlignment(QtCore.Qt.AlignCenter)
        self.label_4.setObjectName("label_4")
        self.gridLayout_2.addWidget(self.label_4, 1, 0, 1, 1)
        self.chooseAmbientColorCam = QtGui.QPushButton(self.dockWidgetContents_9)
        self.chooseAmbientColorCam.setText("")
        self.chooseAmbientColorCam.setObjectName("chooseAmbientColorCam")
        self.gridLayout_2.addWidget(self.chooseAmbientColorCam, 1, 1, 1, 1)
        self.verticalLayout_7.addLayout(self.gridLayout_2)
        self.line_2 = QtGui.QFrame(self.dockWidgetContents_9)
        self.line_2.setFrameShape(QtGui.QFrame.HLine)
        self.line_2.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_2.setObjectName("line_2")
        self.verticalLayout_7.addWidget(self.line_2)
        self.gridLayout_3 = QtGui.QGridLayout()
        self.gridLayout_3.setObjectName("gridLayout_3")
        self.label_6 = QtGui.QLabel(self.dockWidgetContents_9)
        self.label_6.setAlignment(QtCore.Qt.AlignCenter)
        self.label_6.setObjectName("label_6")
        self.gridLayout_3.addWidget(self.label_6, 6, 0, 1, 1)
```

```
        self.label_7 = QtGui.QLabel(self.dockWidgetContents_9)
        self.label_7.setAlignment(QtCore.Qt.AlignCenter)
        self.label_7.setObjectName("label_7")
        self.gridLayout_3.addWidget(self.label_7, 5, 0, 1, 1)
        self.label_8 = QtGui.QLabel(self.dockWidgetContents_9)
        self.label_8.setAlignment(QtCore.Qt.AlignCenter)
        self.label_8.setObjectName("label_8")
        self.gridLayout_3.addWidget(self.label_8, 4, 0, 1, 1)
        self.lightXPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_9)
        self.lightXPos.setMinimum(-1000000.0)
        self.lightXPos.setMaximum(1000000.0)
        self.lightXPos.setObjectName("lightXPos")
        self.gridLayout_3.addWidget(self.lightXPos, 4, 1, 1, 1)
        spacerItem9 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_3.addItem(spacerItem9, 4, 2, 1, 1)
        self.lightYPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_9)
        self.lightYPos.setMinimum(-1000000.0)
        self.lightYPos.setMaximum(1000000.0)
        self.lightYPos.setObjectName("lightYPos")
        self.gridLayout_3.addWidget(self.lightYPos, 5, 1, 1, 1)
        self.lightZPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_9)
        self.lightZPos.setMinimum(-1000000.0)
        self.lightZPos.setMaximum(1000000.0)
        self.lightZPos.setObjectName("lightZPos")
        self.gridLayout_3.addWidget(self.lightZPos, 6, 1, 1, 1)
        self.label_10 = QtGui.QLabel(self.dockWidgetContents_9)
        font = QtGui.QFont()
        font.setWeight(75)
        font.setBold(True)
        self.label_10.setFont(font)
        self.label_10.setAlignment(QtCore.Qt.AlignCenter)
        self.label_10.setObjectName("label_10")
        self.gridLayout_3.addWidget(self.label_10, 3, 0, 1, 1)
        self.verticalLayout_7.addLayout(self.gridLayout_3)
        self.directionalLight = QtGui.QCheckBox(self.dockWidgetContents_9)
        self.directionalLight.setChecked(True)
        self.directionalLight.setObjectName("directionalLight")
        self.verticalLayout_7.addWidget(self.directionalLight)
        spacerItem10 = QtGui.QSpacerItem(20, 40, QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Expanding)
        self.verticalLayout_7.addItem(spacerItem10)
        self.dockLights.setWidget(self.dockWidgetContents_9)
        MainWindow.addDockWidget(QtCore.Qt.DockWidgetArea(8), self.dockLights)
        self.dockModel = QtGui.QDockWidget(MainWindow)
        self.dockModel.setFeatures(QtGui.QDockWidget.AllDockWidgetFeatures)
        self.dockModel.setObjectName("dockModel")
        self.dockWidgetContents_10 = QtGui.QWidget()
        self.dockWidgetContents_10.setObjectName("dockWidgetContents_10")
        self.verticalLayout = QtGui.QVBoxLayout(self.dockWidgetContents_10)
        self.verticalLayout.setObjectName("verticalLayout")
        self.modelName = QtGui.QLineEdit(self.dockWidgetContents_10)
        font = QtGui.QFont()
        font.setPointSize(14)
        self.modelName.setFont(font)
```

```
        self.modelName.setObjectName("modelName")
        self.verticalLayout.addWidget(self.modelName)
        self.modelVisible = QtGui.QCheckBox(self.dockWidgetContents_10)
        self.modelVisible.setEnabled(False)
        font = QtGui.QFont()
        font.setWeight(75)
        font.setBold(True)
        self.modelVisible.setFont(font)
        self.modelVisible.setObjectName("modelVisible")
        self.verticalLayout.addWidget(self.modelVisible)
        self.enableColorDrivenModelsCheckBox =
QtGui.QCheckBox(self.dockWidgetContents_10)
        self.enableColorDrivenModelsCheckBox.setEnabled(True)
        self.enableColorDrivenModelsCheckBox.setChecked(True)

self.enableColorDrivenModelsCheckBox.setObjectName("enableColorDrivenModelsCheckBox")
        self.verticalLayout.addWidget(self.enableColorDrivenModelsCheckBox)
        self.line_3 = QtGui.QFrame(self.dockWidgetContents_10)
        self.line_3.setFrameShape(QtGui.QFrame.HLine)
        self.line_3.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_3.setObjectName("line_3")
        self.verticalLayout.addWidget(self.line_3)
        self.gridLayout_6 = QtGui.QGridLayout()
        self.gridLayout_6.setObjectName("gridLayout_6")
        self.label_18 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_18.setEnabled(True)
        self.label_18.setAlignment(QtCore.Qt.AlignCenter)
        self.label_18.setObjectName("label_18")
        self.gridLayout_6.addWidget(self.label_18, 2, 0, 1, 1)
        self.label_23 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_23.setEnabled(True)
        self.label_23.setAlignment(QtCore.Qt.AlignCenter)
        self.label_23.setObjectName("label_23")
        self.gridLayout_6.addWidget(self.label_23, 4, 0, 1, 1)
        self.chooseDiffuseColorMat = QtGui.QPushButton(self.dockWidgetContents_10)
        self.chooseDiffuseColorMat.setEnabled(False)
        self.chooseDiffuseColorMat.setText("")
        self.chooseDiffuseColorMat.setCheckable(False)
        self.chooseDiffuseColorMat.setChecked(False)
        self.chooseDiffuseColorMat.setAutoDefault(False)
        self.chooseDiffuseColorMat.setDefault(False)
        self.chooseDiffuseColorMat.setFlat(False)
        self.chooseDiffuseColorMat.setObjectName("chooseDiffuseColorMat")
        self.gridLayout_6.addWidget(self.chooseDiffuseColorMat, 2, 1, 1, 1)
        self.chooseEmissiveColorMat = QtGui.QPushButton(self.dockWidgetContents_10)
        self.chooseEmissiveColorMat.setEnabled(False)
        self.chooseEmissiveColorMat.setText("")
        self.chooseEmissiveColorMat.setObjectName("chooseEmissiveColorMat")
        self.gridLayout_6.addWidget(self.chooseEmissiveColorMat, 4, 1, 1, 1)
        spacerItem11 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_6.addItem(spacerItem11, 1, 2, 1, 1)
        self.label_24 = QtGui.QLabel(self.dockWidgetContents_10)
        font = QtGui.QFont()
        font.setPointSize(10)
```

```
        font.setWeight(75)
        font.setBold(True)
        self.label_24.setFont(font)
        self.label_24.setAlignment(QtCore.Qt.AlignCenter)
        self.label_24.setObjectName("label_24")
        self.gridLayout_6.addWidget(self.label_24, 0, 0, 1, 1)
        self.chooseSpecularColorMat = QtGui.QPushButton(self.dockWidgetContents_10)
        self.chooseSpecularColorMat.setEnabled(False)
        sizePolicy = QtGui.QSizePolicy(QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Fixed)
        sizePolicy.setHorizontalStretch(0)
        sizePolicy.setVerticalStretch(0)

sizePolicy.setHeightForWidth(self.chooseSpecularColorMat.sizePolicy().hasHeightForWid
th())
        self.chooseSpecularColorMat.setSizePolicy(sizePolicy)
        self.chooseSpecularColorMat.setText("")
        self.chooseSpecularColorMat.setObjectName("chooseSpecularColorMat")
        self.gridLayout_6.addWidget(self.chooseSpecularColorMat, 3, 1, 1, 1)
        self.label_17 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_17.setEnabled(True)
        self.label_17.setAlignment(QtCore.Qt.AlignCenter)
        self.label_17.setObjectName("label_17")
        self.gridLayout_6.addWidget(self.label_17, 3, 0, 1, 1)
        self.label_22 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_22.setAlignment(QtCore.Qt.AlignCenter)
        self.label_22.setObjectName("label_22")
        self.gridLayout_6.addWidget(self.label_22, 1, 0, 1, 1)
        self.chooseAmbientColorMat = QtGui.QPushButton(self.dockWidgetContents_10)
        self.chooseAmbientColorMat.setEnabled(False)
        self.chooseAmbientColorMat.setText("")
        self.chooseAmbientColorMat.setObjectName("chooseAmbientColorMat")
        self.gridLayout_6.addWidget(self.chooseAmbientColorMat, 1, 1, 1, 1)
        self.verticalLayout.addLayout(self.gridLayout_6)
        self.horizontalLayout_8 = QtGui.QHBoxLayout()
        self.horizontalLayout_8.setObjectName("horizontalLayout_8")
        self.label_32 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_32.setObjectName("label_32")
        self.horizontalLayout_8.addWidget(self.label_32)
        self.modelAlpha = QtGui.QSlider(self.dockWidgetContents_10)
        self.modelAlpha.setMaximum(255)
        self.modelAlpha.setOrientation(QtCore.Qt.Horizontal)
        self.modelAlpha.setObjectName("modelAlpha")
        self.horizontalLayout_8.addWidget(self.modelAlpha)
        self.verticalLayout.addLayout(self.horizontalLayout_8)
        self.gridLayout_7 = QtGui.QGridLayout()
        self.gridLayout_7.setObjectName("gridLayout_7")
        self.label_25 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_25.setObjectName("label_25")
        self.gridLayout_7.addWidget(self.label_25, 1, 0, 1, 1)
        self.label_26 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_26.setObjectName("label_26")
        self.gridLayout_7.addWidget(self.label_26, 1, 2, 1, 1)
        self.matShinySlider = QtGui.QSlider(self.dockWidgetContents_10)
        self.matShinySlider.setEnabled(False)
```

```
        self.matShinySlider.setMaximum(128)
        self.matShinySlider.setProperty("value", 128)
        self.matShinySlider.setOrientation(QtCore.Qt.Horizontal)
        self.matShinySlider.setObjectName("matShinySlider")
        self.gridLayout_7.addWidget(self.matShinySlider, 1, 1, 1, 1)
        self.horizontalLayout = QtGui.QHBoxLayout()
        self.horizontalLayout.setObjectName("horizontalLayout")
        self.label_27 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_27.setObjectName("label_27")
        self.horizontalLayout.addWidget(self.label_27)
        self.matShinySpinBox = QtGui.QDoubleSpinBox(self.dockWidgetContents_10)
        self.matShinySpinBox.setEnabled(False)
        self.matShinySpinBox.setMaximum(128.0)
        self.matShinySpinBox.setProperty("value", 128.0)
        self.matShinySpinBox.setObjectName("matShinySpinBox")
        self.horizontalLayout.addWidget(self.matShinySpinBox)
        self.gridLayout_7.addLayout(self.horizontalLayout, 0, 1, 1, 1)
        self.verticalLayout.addLayout(self.gridLayout_7)
        self.invertNormals = QtGui.QPushButton(self.dockWidgetContents_10)
        self.invertNormals.setObjectName("invertNormals")
        self.verticalLayout.addWidget(self.invertNormals)
        self.saveModel = QtGui.QPushButton(self.dockWidgetContents_10)
        self.saveModel.setObjectName("saveModel")
        self.verticalLayout.addWidget(self.saveModel)
        self.horizontalLayout_11 = QtGui.QHBoxLayout()
        self.horizontalLayout_11.setObjectName("horizontalLayout_11")
        self.label_75 = QtGui.QLabel(self.dockWidgetContents_10)
        self.label_75.setObjectName("label_75")
        self.horizontalLayout_11.addWidget(self.label_75)
        self.pointSize = QtGui.QDoubleSpinBox(self.dockWidgetContents_10)
        self.pointSize.setDecimals(3)
        self.pointSize.setSingleStep(0.1)
        self.pointSize.setProperty("value", 1.0)
        self.pointSize.setObjectName("pointSize")
        self.horizontalLayout_11.addWidget(self.pointSize)
        self.verticalLayout.addLayout(self.horizontalLayout_11)
        spacerItem12 = QtGui.QSpacerItem(20, 40, QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Expanding)
        self.verticalLayout.addItem(spacerItem12)
        self.dockModel.setWidget(self.dockWidgetContents_10)
        MainWindow.addDockWidget(QtCore.Qt.DockWidgetArea(8), self.dockModel)
        self.dockJoint = QtGui.QDockWidget(MainWindow)
        self.dockJoint.setFeatures(QtGui.QDockWidget.AllDockWidgetFeatures)
        self.dockJoint.setObjectName("dockJoint")
        self.dockWidgetContents_11 = QtGui.QWidget()
        self.dockWidgetContents_11.setObjectName("dockWidgetContents_11")
        self.verticalLayout_2 = QtGui.QVBoxLayout(self.dockWidgetContents_11)
        self.verticalLayout_2.setObjectName("verticalLayout_2")
        self.jointName = QtGui.QLineEdit(self.dockWidgetContents_11)
        font = QtGui.QFont()
        font.setPointSize(14)
        self.jointName.setFont(font)
        self.jointName.setObjectName("jointName")
        self.verticalLayout_2.addWidget(self.jointName)
        self.line_4 = QtGui.QFrame(self.dockWidgetContents_11)
```

```
        self.line_4.setFrameShape(QtGui.QFrame.HLine)
        self.line_4.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_4.setObjectName("line_4")
        self.verticalLayout_2.addWidget(self.line_4)
        self.horizontalLayout_4 = QtGui.QHBoxLayout()
        self.horizontalLayout_4.setObjectName("horizontalLayout_4")
        self.label_28 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_28.setObjectName("label_28")
        self.horizontalLayout_4.addWidget(self.label_28)
        spacerItem13 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.horizontalLayout_4.addItem(spacerItem13)
        self.label_19 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_19.setObjectName("label_19")
        self.horizontalLayout_4.addWidget(self.label_19)
        self.rotationOrder = QtGui.QComboBox(self.dockWidgetContents_11)
        self.rotationOrder.setEnabled(False)
        self.rotationOrder.setObjectName("rotationOrder")
        self.rotationOrder.addItem("")
        self.rotationOrder.addItem("")
        self.rotationOrder.addItem("")
        self.rotationOrder.addItem("")
        self.rotationOrder.addItem("")
        self.rotationOrder.addItem("")
        self.horizontalLayout_4.addWidget(self.rotationOrder)
        self.verticalLayout_2.addLayout(self.horizontalLayout_4)
        self.sphericalCoord = QtGui.QCheckBox(self.dockWidgetContents_11)
        self.sphericalCoord.setChecked(True)
        self.sphericalCoord.setObjectName("sphericalCoord")
        self.verticalLayout_2.addWidget(self.sphericalCoord)
        self.gridLayout_8 = QtGui.QGridLayout()
        self.gridLayout_8.setObjectName("gridLayout_8")
        self.xAxisLabel = QtGui.QLabel(self.dockWidgetContents_11)
        self.xAxisLabel.setObjectName("xAxisLabel")
        self.gridLayout_8.addWidget(self.xAxisLabel, 1, 0, 1, 1)
        self.label_30 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_30.setObjectName("label_30")
        self.gridLayout_8.addWidget(self.label_30, 0, 0, 1, 1)
        self.jointXRotSpinBox = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointXRotSpinBox.setEnabled(False)
        self.jointXRotSpinBox.setMinimum(-90.0)
        self.jointXRotSpinBox.setMaximum(90.0)
        self.jointXRotSpinBox.setSingleStep(15.0)
        self.jointXRotSpinBox.setObjectName("jointXRotSpinBox")
        self.gridLayout_8.addWidget(self.jointXRotSpinBox, 1, 2, 1, 1)
        self.label_31 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_31.setObjectName("label_31")
        self.gridLayout_8.addWidget(self.label_31, 0, 2, 1, 1)
        self.zAxisLabel = QtGui.QLabel(self.dockWidgetContents_11)
        self.zAxisLabel.setObjectName("zAxisLabel")
        self.gridLayout_8.addWidget(self.zAxisLabel, 3, 0, 1, 1)
        self.yAxisLabel = QtGui.QLabel(self.dockWidgetContents_11)
        self.yAxisLabel.setObjectName("yAxisLabel")
        self.gridLayout_8.addWidget(self.yAxisLabel, 2, 0, 1, 1)
        self.jointYRotSlider = QtGui.QSlider(self.dockWidgetContents_11)
```

```
        self.jointYRotSlider.setEnabled(False)
        self.jointYRotSlider.setMaximum(360)
        self.jointYRotSlider.setOrientation(QtCore.Qt.Horizontal)
        self.jointYRotSlider.setObjectName("jointYRotSlider")
        self.gridLayout_8.addWidget(self.jointYRotSlider, 2, 1, 1, 1)
        self.jointZRotSlider = QtGui.QSlider(self.dockWidgetContents_11)
        self.jointZRotSlider.setEnabled(False)
        self.jointZRotSlider.setMaximum(360)
        self.jointZRotSlider.setOrientation(QtCore.Qt.Horizontal)
        self.jointZRotSlider.setObjectName("jointZRotSlider")
        self.gridLayout_8.addWidget(self.jointZRotSlider, 3, 1, 1, 1)
        self.jointYRotSpinBox = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointYRotSpinBox.setEnabled(False)
        self.jointYRotSpinBox.setMaximum(360.0)
        self.jointYRotSpinBox.setSingleStep(15.0)
        self.jointYRotSpinBox.setObjectName("jointYRotSpinBox")
        self.gridLayout_8.addWidget(self.jointYRotSpinBox, 2, 2, 1, 1)
        self.jointZRotSpinBox = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointZRotSpinBox.setEnabled(False)
        self.jointZRotSpinBox.setMaximum(360.0)
        self.jointZRotSpinBox.setSingleStep(15.0)
        self.jointZRotSpinBox.setObjectName("jointZRotSpinBox")
        self.gridLayout_8.addWidget(self.jointZRotSpinBox, 3, 2, 1, 1)
        self.jointXRotSlider = QtGui.QSlider(self.dockWidgetContents_11)
        self.jointXRotSlider.setEnabled(False)
        self.jointXRotSlider.setMinimum(-90)
        self.jointXRotSlider.setMaximum(90)
        self.jointXRotSlider.setOrientation(QtCore.Qt.Horizontal)
        self.jointXRotSlider.setObjectName("jointXRotSlider")
        self.gridLayout_8.addWidget(self.jointXRotSlider, 1, 1, 1, 1)
        self.verticalLayout_2.addLayout(self.gridLayout_8)
        self.line_5 = QtGui.QFrame(self.dockWidgetContents_11)
        self.line_5.setFrameShape(QtGui.QFrame.HLine)
        self.line_5.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_5.setObjectName("line_5")
        self.verticalLayout_2.addWidget(self.line_5)
        self.gridLayout_14 = QtGui.QGridLayout()
        self.gridLayout_14.setObjectName("gridLayout_14")
        self.label_51 = QtGui.QLabel(self.dockWidgetContents_11)

self.label_51.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_51.setObjectName("label_51")
        self.gridLayout_14.addWidget(self.label_51, 3, 0, 1, 1)
        self.label_52 = QtGui.QLabel(self.dockWidgetContents_11)

self.label_52.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_52.setObjectName("label_52")
        self.gridLayout_14.addWidget(self.label_52, 4, 0, 1, 1)
        self.label_53 = QtGui.QLabel(self.dockWidgetContents_11)

self.label_53.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_53.setObjectName("label_53")
```

```
        self.gridLayout_14.addWidget(self.label_53, 5, 0, 1, 1)
        spacerItem14 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_14.addItem(spacerItem14, 3, 2, 1, 1)
        self.jointScaleX = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointScaleX.setDecimals(4)
        self.jointScaleX.setMinimum(0.0001)
        self.jointScaleX.setMaximum(100.0)
        self.jointScaleX.setSingleStep(0.1)
        self.jointScaleX.setProperty("value", 1.0)
        self.jointScaleX.setObjectName("jointScaleX")
        self.gridLayout_14.addWidget(self.jointScaleX, 3, 1, 1, 1)
        self.label_60 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_60.setObjectName("label_60")
        self.gridLayout_14.addWidget(self.label_60, 0, 1, 1, 1)
        self.jointScaleY = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointScaleY.setDecimals(4)
        self.jointScaleY.setMinimum(0.0001)
        self.jointScaleY.setMaximum(100.0)
        self.jointScaleY.setSingleStep(0.1)
        self.jointScaleY.setProperty("value", 1.0)
        self.jointScaleY.setObjectName("jointScaleY")
        self.gridLayout_14.addWidget(self.jointScaleY, 4, 1, 1, 1)
        self.jointScaleZ = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointScaleZ.setDecimals(4)
        self.jointScaleZ.setMinimum(0.0001)
        self.jointScaleZ.setMaximum(100.0)
        self.jointScaleZ.setSingleStep(0.1)
        self.jointScaleZ.setProperty("value", 1.0)
        self.jointScaleZ.setObjectName("jointScaleZ")
        self.gridLayout_14.addWidget(self.jointScaleZ, 5, 1, 1, 1)
        self.label_29 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_29.setObjectName("label_29")
        self.gridLayout_14.addWidget(self.label_29, 2, 0, 1, 1)
        self.jointScaleXYZ = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointScaleXYZ.setDecimals(4)
        self.jointScaleXYZ.setMinimum(0.0001)
        self.jointScaleXYZ.setMaximum(100.0)
        self.jointScaleXYZ.setSingleStep(0.1)
        self.jointScaleXYZ.setProperty("value", 1.0)
        self.jointScaleXYZ.setObjectName("jointScaleXYZ")
        self.gridLayout_14.addWidget(self.jointScaleXYZ, 2, 1, 1, 1)
        self.verticalLayout_2.addLayout(self.gridLayout_14)
        self.line_7 = QtGui.QFrame(self.dockWidgetContents_11)
        self.line_7.setFrameShape(QtGui.QFrame.HLine)
        self.line_7.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_7.setObjectName("line_7")
        self.verticalLayout_2.addWidget(self.line_7)
        self.label_34 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_34.setToolTip("")
        self.label_34.setObjectName("label_34")
        self.verticalLayout_2.addWidget(self.label_34)
        self.gridLayout_9 = QtGui.QGridLayout()
        self.gridLayout_9.setObjectName("gridLayout_9")
        self.label_35 = QtGui.QLabel(self.dockWidgetContents_11)
```

```
self.label_35.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_35.setObjectName("label_35")
        self.gridLayout_9.addWidget(self.label_35, 1, 0, 1, 1)
        self.label_36 = QtGui.QLabel(self.dockWidgetContents_11)

self.label_36.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_36.setObjectName("label_36")
        self.gridLayout_9.addWidget(self.label_36, 2, 0, 1, 1)
        self.label_37 = QtGui.QLabel(self.dockWidgetContents_11)

self.label_37.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Ali
gnVCenter)
        self.label_37.setObjectName("label_37")
        self.gridLayout_9.addWidget(self.label_37, 3, 0, 1, 1)
        self.jointXPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointXPos.setEnabled(True)
        self.jointXPos.setWrapping(False)
        self.jointXPos.setReadOnly(True)
        self.jointXPos.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.jointXPos.setMinimum(-1000000.0)
        self.jointXPos.setMaximum(1000000.0)
        self.jointXPos.setObjectName("jointXPos")
        self.gridLayout_9.addWidget(self.jointXPos, 1, 1, 1, 1)
        self.jointYPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointYPos.setEnabled(True)
        self.jointYPos.setReadOnly(True)
        self.jointYPos.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.jointYPos.setMinimum(-1000000.0)
        self.jointYPos.setMaximum(1000000.0)
        self.jointYPos.setObjectName("jointYPos")
        self.gridLayout_9.addWidget(self.jointYPos, 2, 1, 1, 1)
        self.jointZPos = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointZPos.setEnabled(True)
        self.jointZPos.setReadOnly(True)
        self.jointZPos.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.jointZPos.setMinimum(-1000000.0)
        self.jointZPos.setMaximum(1000000.0)
        self.jointZPos.setObjectName("jointZPos")
        self.gridLayout_9.addWidget(self.jointZPos, 3, 1, 1, 1)
        self.label_33 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_33.setObjectName("label_33")
        self.gridLayout_9.addWidget(self.label_33, 0, 1, 1, 1)
        self.jointXPosStep = QtGui.QPushButton(self.dockWidgetContents_11)
        self.jointXPosStep.setObjectName("jointXPosStep")
        self.gridLayout_9.addWidget(self.jointXPosStep, 1, 3, 1, 1)
        spacerItem15 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_9.addItem(spacerItem15, 1, 4, 1, 1)
        self.jointXPosStepSize = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointXPosStepSize.setMinimum(-1000.0)
        self.jointXPosStepSize.setMaximum(1000.0)
        self.jointXPosStepSize.setObjectName("jointXPosStepSize")
```

```
        self.gridLayout_9.addWidget(self.jointXPosStepSize, 1, 2, 1, 1)
        self.label_38 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_38.setObjectName("label_38")
        self.gridLayout_9.addWidget(self.label_38, 0, 2, 1, 1)
        self.jointYPosStep = QtGui.QPushButton(self.dockWidgetContents_11)
        self.jointYPosStep.setObjectName("jointYPosStep")
        self.gridLayout_9.addWidget(self.jointYPosStep, 2, 3, 1, 1)
        self.jointZPosStep = QtGui.QPushButton(self.dockWidgetContents_11)
        self.jointZPosStep.setObjectName("jointZPosStep")
        self.gridLayout_9.addWidget(self.jointZPosStep, 3, 3, 1, 1)
        self.jointYPosStepSize = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointYPosStepSize.setMinimum(-1000.0)
        self.jointYPosStepSize.setMaximum(1000.0)
        self.jointYPosStepSize.setObjectName("jointYPosStepSize")
        self.gridLayout_9.addWidget(self.jointYPosStepSize, 2, 2, 1, 1)
        self.jointZPosStepSize = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointZPosStepSize.setMinimum(-1000.0)
        self.jointZPosStepSize.setMaximum(1000.0)
        self.jointZPosStepSize.setObjectName("jointZPosStepSize")
        self.gridLayout_9.addWidget(self.jointZPosStepSize, 3, 2, 1, 1)
        self.verticalLayout_2.addLayout(self.gridLayout_9)
        self.line_6 = QtGui.QFrame(self.dockWidgetContents_11)
        self.line_6.setFrameShape(QtGui.QFrame.HLine)
        self.line_6.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_6.setObjectName("line_6")
        self.verticalLayout_2.addWidget(self.line_6)
        self.label_12 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_12.setObjectName("label_12")
        self.verticalLayout_2.addWidget(self.label_12)
        self.gridLayout_4 = QtGui.QGridLayout()
        self.gridLayout_4.setObjectName("gridLayout_4")
        self.label_13 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_13.setObjectName("label_13")
        self.gridLayout_4.addWidget(self.label_13, 0, 0, 1, 1)
        self.label_14 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_14.setObjectName("label_14")
        self.gridLayout_4.addWidget(self.label_14, 1, 0, 1, 1)
        self.label_15 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_15.setObjectName("label_15")
        self.gridLayout_4.addWidget(self.label_15, 2, 0, 1, 1)
        self.jointQuatW = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointQuatW.setReadOnly(True)
        self.jointQuatW.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.jointQuatW.setMinimum(-1000000.0)
        self.jointQuatW.setMaximum(1000000.0)
        self.jointQuatW.setObjectName("jointQuatW")
        self.gridLayout_4.addWidget(self.jointQuatW, 0, 1, 1, 1)
        spacerItem16 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_4.addItem(spacerItem16, 0, 2, 1, 1)
        self.jointQuatX = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointQuatX.setReadOnly(True)
        self.jointQuatX.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.jointQuatX.setMinimum(-1000000.0)
        self.jointQuatX.setMaximum(1000000.0)
```

```
        self.jointQuatX.setObjectName("jointQuatX")
        self.gridLayout_4.addWidget(self.jointQuatX, 1, 1, 1, 1)
        self.jointQuatY = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointQuatY.setReadOnly(True)
        self.jointQuatY.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.jointQuatY.setMinimum(-1000000.0)
        self.jointQuatY.setMaximum(1000000.0)
        self.jointQuatY.setObjectName("jointQuatY")
        self.gridLayout_4.addWidget(self.jointQuatY, 2, 1, 1, 1)
        self.jointQuatZ = QtGui.QDoubleSpinBox(self.dockWidgetContents_11)
        self.jointQuatZ.setReadOnly(True)
        self.jointQuatZ.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.jointQuatZ.setMinimum(-1000000.0)
        self.jointQuatZ.setMaximum(1000000.0)
        self.jointQuatZ.setObjectName("jointQuatZ")
        self.gridLayout_4.addWidget(self.jointQuatZ, 3, 1, 1, 1)
        self.label_16 = QtGui.QLabel(self.dockWidgetContents_11)
        self.label_16.setObjectName("label_16")
        self.gridLayout_4.addWidget(self.label_16, 3, 0, 1, 1)
        self.verticalLayout_2.addLayout(self.gridLayout_4)
        spacerItem17 = QtGui.QSpacerItem(20, 40, QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Expanding)
        self.verticalLayout_2.addItem(spacerItem17)
        self.dockJoint.setWidget(self.dockWidgetContents_11)
        MainWindow.addDockWidget(QtCore.Qt.DockWidgetArea(8), self.dockJoint)
        self.dockScene = QtGui.QDockWidget(MainWindow)
        self.dockScene.setFeatures(QtGui.QDockWidget.AllDockWidgetFeatures)
        self.dockScene.setObjectName("dockScene")
        self.dockWidgetContents_12 = QtGui.QWidget()
        self.dockWidgetContents_12.setObjectName("dockWidgetContents_12")
        self.verticalLayout_4 = QtGui.QVBoxLayout(self.dockWidgetContents_12)
        self.verticalLayout_4.setObjectName("verticalLayout_4")
        self.horizontalLayout_5 = QtGui.QHBoxLayout()
        self.horizontalLayout_5.setObjectName("horizontalLayout_5")
        self.label_20 = QtGui.QLabel(self.dockWidgetContents_12)
        self.label_20.setObjectName("label_20")
        self.horizontalLayout_5.addWidget(self.label_20)
        self.projection = QtGui.QComboBox(self.dockWidgetContents_12)
        self.projection.setObjectName("projection")
        self.projection.addItem("")
        self.projection.addItem("")
        self.horizontalLayout_5.addWidget(self.projection)
        spacerItem18 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.horizontalLayout_5.addItem(spacerItem18)
        self.verticalLayout_4.addLayout(self.horizontalLayout_5)
        self.useCallLists = QtGui.QCheckBox(self.dockWidgetContents_12)
        self.useCallLists.setChecked(True)
        self.useCallLists.setObjectName("useCallLists")
        self.verticalLayout_4.addWidget(self.useCallLists)
        self.line_12 = QtGui.QFrame(self.dockWidgetContents_12)
        self.line_12.setFrameShape(QtGui.QFrame.HLine)
        self.line_12.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_12.setObjectName("line_12")
        self.verticalLayout_4.addWidget(self.line_12)
```

```
        self.automateNextStep = QtGui.QPushButton(self.dockWidgetContents_12)
        self.automateNextStep.setObjectName("automateNextStep")
        self.verticalLayout_4.addWidget(self.automateNextStep)
        spacerItem19 = QtGui.QSpacerItem(20, 40, QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Expanding)
        self.verticalLayout_4.addItem(spacerItem19)
        self.dockScene.setWidget(self.dockWidgetContents_12)
        MainWindow.addDockWidget(QtCore.Qt.DockWidgetArea(8), self.dockScene)
        self.dockSlice = QtGui.QDockWidget(MainWindow)
        self.dockSlice.setFeatures(QtGui.QDockWidget.AllDockWidgetFeatures)
        self.dockSlice.setObjectName("dockSlice")
        self.dockWidgetContents_13 = QtGui.QWidget()
        self.dockWidgetContents_13.setObjectName("dockWidgetContents_13")
        self.verticalLayout_3 = QtGui.QVBoxLayout(self.dockWidgetContents_13)
        self.verticalLayout_3.setObjectName("verticalLayout_3")
        self.horizontalLayout_6 = QtGui.QHBoxLayout()
        self.horizontalLayout_6.setObjectName("horizontalLayout_6")
        self.label_21 = QtGui.QLabel(self.dockWidgetContents_13)
        self.label_21.setObjectName("label_21")
        self.horizontalLayout_6.addWidget(self.label_21)
        self.spinBox100AxisSlice = QtGui.QSpinBox(self.dockWidgetContents_13)

self.spinBox100AxisSlice.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|Qt
Core.Qt.AlignVCenter)
        self.spinBox100AxisSlice.setObjectName("spinBox100AxisSlice")
        self.horizontalLayout_6.addWidget(self.spinBox100AxisSlice)
        self.verticalLayout_3.addLayout(self.horizontalLayout_6)
        self.slider100AxisSlice = QtGui.QSlider(self.dockWidgetContents_13)
        self.slider100AxisSlice.setOrientation(QtCore.Qt.Horizontal)
        self.slider100AxisSlice.setObjectName("slider100AxisSlice")
        self.verticalLayout_3.addWidget(self.slider100AxisSlice)
        self.line_13 = QtGui.QFrame(self.dockWidgetContents_13)
        self.line_13.setFrameShape(QtGui.QFrame.HLine)
        self.line_13.setFrameShadow(QtGui.QFrame.Sunken)
        self.line_13.setObjectName("line_13")
        self.verticalLayout_3.addWidget(self.line_13)
        self.horizontalLayout_7 = QtGui.QHBoxLayout()
        self.horizontalLayout_7.setObjectName("horizontalLayout_7")
        self.label_61 = QtGui.QLabel(self.dockWidgetContents_13)
        self.label_61.setObjectName("label_61")
        self.horizontalLayout_7.addWidget(self.label_61)
        self.spinBox010AxisSlice = QtGui.QSpinBox(self.dockWidgetContents_13)

self.spinBox010AxisSlice.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|Qt
Core.Qt.AlignVCenter)
        self.spinBox010AxisSlice.setObjectName("spinBox010AxisSlice")
        self.horizontalLayout_7.addWidget(self.spinBox010AxisSlice)
        self.verticalLayout_3.addLayout(self.horizontalLayout_7)
        self.slider010AxisSlice = QtGui.QSlider(self.dockWidgetContents_13)
        self.slider010AxisSlice.setOrientation(QtCore.Qt.Horizontal)
        self.slider010AxisSlice.setObjectName("slider010AxisSlice")
        self.verticalLayout_3.addWidget(self.slider010AxisSlice)
        self.line_14 = QtGui.QFrame(self.dockWidgetContents_13)
        self.line_14.setFrameShape(QtGui.QFrame.HLine)
        self.line_14.setFrameShadow(QtGui.QFrame.Sunken)
```

```
        self.line_14.setObjectName("line_14")
        self.verticalLayout_3.addWidget(self.line_14)
        self.horizontalLayout_9 = QtGui.QHBoxLayout()
        self.horizontalLayout_9.setObjectName("horizontalLayout_9")
        self.label_62 = QtGui.QLabel(self.dockWidgetContents_13)
        self.label_62.setObjectName("label_62")
        self.horizontalLayout_9.addWidget(self.label_62)
        self.spinBox001AxisSlice = QtGui.QSpinBox(self.dockWidgetContents_13)

self.spinBox001AxisSlice.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|Qt
Core.Qt.AlignVCenter)
        self.spinBox001AxisSlice.setObjectName("spinBox001AxisSlice")
        self.horizontalLayout_9.addWidget(self.spinBox001AxisSlice)
        self.verticalLayout_3.addLayout(self.horizontalLayout_9)
        self.slider001AxisSlice = QtGui.QSlider(self.dockWidgetContents_13)
        self.slider001AxisSlice.setOrientation(QtCore.Qt.Horizontal)
        self.slider001AxisSlice.setObjectName("slider001AxisSlice")
        self.verticalLayout_3.addWidget(self.slider001AxisSlice)
        spacerItem20 = QtGui.QSpacerItem(20, 40, QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Expanding)
        self.verticalLayout_3.addItem(spacerItem20)
        self.dockSlice.setWidget(self.dockWidgetContents_13)
        MainWindow.addDockWidget(QtCore.Qt.DockWidgetArea(8), self.dockSlice)
        self.dockTol = QtGui.QDockWidget(MainWindow)
        self.dockTol.setObjectName("dockTol")
        self.dockWidgetContents = QtGui.QWidget()
        self.dockWidgetContents.setObjectName("dockWidgetContents")
        self.verticalLayout_11 = QtGui.QVBoxLayout(self.dockWidgetContents)
        self.verticalLayout_11.setObjectName("verticalLayout_11")
        self.verticalLayout_10 = QtGui.QVBoxLayout()
        self.verticalLayout_10.setObjectName("verticalLayout_10")
        self.label_65 = QtGui.QLabel(self.dockWidgetContents)
        self.label_65.setAlignment(QtCore.Qt.AlignCenter)
        self.label_65.setObjectName("label_65")
        self.verticalLayout_10.addWidget(self.label_65)
        self.gridLayout_5 = QtGui.QGridLayout()
        self.gridLayout_5.setObjectName("gridLayout_5")
        self.label_54 = QtGui.QLabel(self.dockWidgetContents)
        self.label_54.setAlignment(QtCore.Qt.AlignCenter)
        self.label_54.setObjectName("label_54")
        self.gridLayout_5.addWidget(self.label_54, 0, 0, 1, 1)
        self.label_63 = QtGui.QLabel(self.dockWidgetContents)
        self.label_63.setAlignment(QtCore.Qt.AlignCenter)
        self.label_63.setObjectName("label_63")
        self.gridLayout_5.addWidget(self.label_63, 0, 1, 1, 1)
        self.label_64 = QtGui.QLabel(self.dockWidgetContents)
        self.label_64.setAlignment(QtCore.Qt.AlignCenter)
        self.label_64.setObjectName("label_64")
        self.gridLayout_5.addWidget(self.label_64, 0, 2, 1, 1)
        self.tolVecXOrigin = QtGui.QDoubleSpinBox(self.dockWidgetContents)

self.tolVecXOrigin.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Q
t.AlignVCenter)
        self.tolVecXOrigin.setDecimals(4)
        self.tolVecXOrigin.setMinimum(-1000000.0)
```

```
        self.tolVecXOrigin.setMaximum(1000000.0)
        self.tolVecXOrigin.setObjectName("tolVecXOrigin")
        self.gridLayout_5.addWidget(self.tolVecXOrigin, 1, 0, 1, 1)
        self.tolVecYOrigin = QtGui.QDoubleSpinBox(self.dockWidgetContents)

self.tolVecYOrigin.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Q
t.AlignVCenter)
        self.tolVecYOrigin.setDecimals(4)
        self.tolVecYOrigin.setMinimum(-1000000.0)
        self.tolVecYOrigin.setMaximum(1000000.0)
        self.tolVecYOrigin.setObjectName("tolVecYOrigin")
        self.gridLayout_5.addWidget(self.tolVecYOrigin, 1, 1, 1, 1)
        self.tolVecZOrigin = QtGui.QDoubleSpinBox(self.dockWidgetContents)

self.tolVecZOrigin.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Q
t.AlignVCenter)
        self.tolVecZOrigin.setDecimals(4)
        self.tolVecZOrigin.setMinimum(-1000000.0)
        self.tolVecZOrigin.setMaximum(1000000.0)
        self.tolVecZOrigin.setObjectName("tolVecZOrigin")
        self.gridLayout_5.addWidget(self.tolVecZOrigin, 1, 2, 1, 1)
        self.verticalLayout_10.addLayout(self.gridLayout_5)
        self.verticalLayout_11.addLayout(self.verticalLayout_10)
        self.gridLayout_16 = QtGui.QGridLayout()
        self.gridLayout_16.setObjectName("gridLayout_16")
        self.label_66 = QtGui.QLabel(self.dockWidgetContents)
        self.label_66.setObjectName("label_66")
        self.gridLayout_16.addWidget(self.label_66, 0, 1, 1, 1)
        self.tolVecLength = QtGui.QDoubleSpinBox(self.dockWidgetContents)

self.tolVecLength.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt
.AlignVCenter)
        self.tolVecLength.setMaximum(1000.0)
        self.tolVecLength.setSingleStep(5.0)
        self.tolVecLength.setProperty("value", 10.0)
        self.tolVecLength.setObjectName("tolVecLength")
        self.gridLayout_16.addWidget(self.tolVecLength, 0, 2, 1, 1)
        spacerItem21 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.gridLayout_16.addItem(spacerItem21, 0, 0, 1, 1)
        self.label_69 = QtGui.QLabel(self.dockWidgetContents)
        self.label_69.setObjectName("label_69")
        self.gridLayout_16.addWidget(self.label_69, 1, 1, 1, 1)
        self.tolVecAngle = QtGui.QDoubleSpinBox(self.dockWidgetContents)

self.tolVecAngle.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.
AlignVCenter)
        self.tolVecAngle.setMaximum(180.0)
        self.tolVecAngle.setSingleStep(5.0)
        self.tolVecAngle.setProperty("value", 45.0)
        self.tolVecAngle.setObjectName("tolVecAngle")
        self.gridLayout_16.addWidget(self.tolVecAngle, 1, 2, 1, 1)
        self.verticalLayout_11.addLayout(self.gridLayout_16)
        self.gridLayout_15 = QtGui.QGridLayout()
        self.gridLayout_15.setObjectName("gridLayout_15")
```

```
        self.xAxisLabel_2 = QtGui.QLabel(self.dockWidgetContents)
        self.xAxisLabel_2.setObjectName("xAxisLabel_2")
        self.gridLayout_15.addWidget(self.xAxisLabel_2, 1, 0, 1, 1)
        self.label_67 = QtGui.QLabel(self.dockWidgetContents)
        self.label_67.setObjectName("label_67")
        self.gridLayout_15.addWidget(self.label_67, 0, 0, 1, 1)
        self.tolXRotSpinBox = QtGui.QDoubleSpinBox(self.dockWidgetContents)
        self.tolXRotSpinBox.setEnabled(True)

self.tolXRotSpinBox.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.
Qt.AlignVCenter)
        self.tolXRotSpinBox.setMinimum(-90.0)
        self.tolXRotSpinBox.setMaximum(90.0)
        self.tolXRotSpinBox.setSingleStep(15.0)
        self.tolXRotSpinBox.setObjectName("tolXRotSpinBox")
        self.gridLayout_15.addWidget(self.tolXRotSpinBox, 1, 2, 1, 1)
        self.label_68 = QtGui.QLabel(self.dockWidgetContents)
        self.label_68.setObjectName("label_68")
        self.gridLayout_15.addWidget(self.label_68, 0, 2, 1, 1)
        self.zAxisLabel_2 = QtGui.QLabel(self.dockWidgetContents)
        self.zAxisLabel_2.setObjectName("zAxisLabel_2")
        self.gridLayout_15.addWidget(self.zAxisLabel_2, 3, 0, 1, 1)
        self.yAxisLabel_2 = QtGui.QLabel(self.dockWidgetContents)
        self.yAxisLabel_2.setObjectName("yAxisLabel_2")
        self.gridLayout_15.addWidget(self.yAxisLabel_2, 2, 0, 1, 1)
        self.tolYRotSlider = QtGui.QSlider(self.dockWidgetContents)
        self.tolYRotSlider.setEnabled(True)
        self.tolYRotSlider.setMaximum(360)
        self.tolYRotSlider.setOrientation(QtCore.Qt.Horizontal)
        self.tolYRotSlider.setObjectName("tolYRotSlider")
        self.gridLayout_15.addWidget(self.tolYRotSlider, 2, 1, 1, 1)
        self.tolZRotSlider = QtGui.QSlider(self.dockWidgetContents)
        self.tolZRotSlider.setEnabled(True)
        self.tolZRotSlider.setMaximum(360)
        self.tolZRotSlider.setOrientation(QtCore.Qt.Horizontal)
        self.tolZRotSlider.setObjectName("tolZRotSlider")
        self.gridLayout_15.addWidget(self.tolZRotSlider, 3, 1, 1, 1)
        self.tolYRotSpinBox = QtGui.QDoubleSpinBox(self.dockWidgetContents)
        self.tolYRotSpinBox.setEnabled(True)

self.tolYRotSpinBox.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.
Qt.AlignVCenter)
        self.tolYRotSpinBox.setMaximum(360.0)
        self.tolYRotSpinBox.setSingleStep(15.0)
        self.tolYRotSpinBox.setObjectName("tolYRotSpinBox")
        self.gridLayout_15.addWidget(self.tolYRotSpinBox, 2, 2, 1, 1)
        self.tolZRotSpinBox = QtGui.QDoubleSpinBox(self.dockWidgetContents)
        self.tolZRotSpinBox.setEnabled(True)

self.tolZRotSpinBox.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.
Qt.AlignVCenter)
        self.tolZRotSpinBox.setMaximum(360.0)
        self.tolZRotSpinBox.setSingleStep(15.0)
        self.tolZRotSpinBox.setObjectName("tolZRotSpinBox")
        self.gridLayout_15.addWidget(self.tolZRotSpinBox, 3, 2, 1, 1)
```

```
        self.tolXRotSlider = QtGui.QSlider(self.dockWidgetContents)
        self.tolXRotSlider.setEnabled(True)
        self.tolXRotSlider.setMinimum(0)
        self.tolXRotSlider.setMaximum(360)
        self.tolXRotSlider.setOrientation(QtCore.Qt.Horizontal)
        self.tolXRotSlider.setObjectName("tolXRotSlider")
        self.gridLayout_15.addWidget(self.tolXRotSlider, 1, 1, 1, 1)
        self.verticalLayout_11.addLayout(self.gridLayout_15)
        self.verticalLayout_12 = QtGui.QVBoxLayout()
        self.verticalLayout_12.setObjectName("verticalLayout_12")
        self.label_70 = QtGui.QLabel(self.dockWidgetContents)
        self.label_70.setAlignment(QtCore.Qt.AlignCenter)
        self.label_70.setObjectName("label_70")
        self.verticalLayout_12.addWidget(self.label_70)
        self.gridLayout_17 = QtGui.QGridLayout()
        self.gridLayout_17.setObjectName("gridLayout_17")
        self.label_71 = QtGui.QLabel(self.dockWidgetContents)
        self.label_71.setAlignment(QtCore.Qt.AlignCenter)
        self.label_71.setObjectName("label_71")
        self.gridLayout_17.addWidget(self.label_71, 0, 0, 1, 1)
        self.label_72 = QtGui.QLabel(self.dockWidgetContents)
        self.label_72.setAlignment(QtCore.Qt.AlignCenter)
        self.label_72.setObjectName("label_72")
        self.gridLayout_17.addWidget(self.label_72, 0, 1, 1, 1)
        self.label_73 = QtGui.QLabel(self.dockWidgetContents)
        self.label_73.setAlignment(QtCore.Qt.AlignCenter)
        self.label_73.setObjectName("label_73")
        self.gridLayout_17.addWidget(self.label_73, 0, 2, 1, 1)
        self.tolVecX = QtGui.QDoubleSpinBox(self.dockWidgetContents)

self.tolVecX.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Alig
nVCenter)
        self.tolVecX.setReadOnly(True)
        self.tolVecX.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.tolVecX.setDecimals(4)
        self.tolVecX.setMinimum(-1000000.0)
        self.tolVecX.setMaximum(1000000.0)
        self.tolVecX.setObjectName("tolVecX")
        self.gridLayout_17.addWidget(self.tolVecX, 1, 0, 1, 1)
        self.tolVecY = QtGui.QDoubleSpinBox(self.dockWidgetContents)

self.tolVecY.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Alig
nVCenter)
        self.tolVecY.setReadOnly(True)
        self.tolVecY.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.tolVecY.setDecimals(4)
        self.tolVecY.setMinimum(-1000000.0)
        self.tolVecY.setMaximum(1000000.0)
        self.tolVecY.setObjectName("tolVecY")
        self.gridLayout_17.addWidget(self.tolVecY, 1, 1, 1, 1)
        self.tolVecZ = QtGui.QDoubleSpinBox(self.dockWidgetContents)

self.tolVecZ.setAlignment(QtCore.Qt.AlignRight|QtCore.Qt.AlignTrailing|QtCore.Qt.Alig
nVCenter)
        self.tolVecZ.setReadOnly(True)
```

```
        self.tolVecZ.setButtonSymbols(QtGui.QAbstractSpinBox.NoButtons)
        self.tolVecZ.setDecimals(4)
        self.tolVecZ.setMinimum(-1000000.0)
        self.tolVecZ.setMaximum(1000000.0)
        self.tolVecZ.setObjectName("tolVecZ")
        self.gridLayout_17.addWidget(self.tolVecZ, 1, 2, 1, 1)
        self.verticalLayout_12.addLayout(self.gridLayout_17)
        self.verticalLayout_11.addLayout(self.verticalLayout_12)
        spacerItem22 = QtGui.QSpacerItem(20, 40, QtGui.QSizePolicy.Minimum,
QtGui.QSizePolicy.Expanding)
        self.verticalLayout_11.addItem(spacerItem22)
        self.dockTol.setWidget(self.dockWidgetContents)
        MainWindow.addDockWidget(QtCore.Qt.DockWidgetArea(8), self.dockTol)

        self.retranslateUi(MainWindow)
        self.viewTabs.setCurrentIndex(0)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        MainWindow.setWindowTitle(QtGui.QApplication.translate("MainWindow", "Joint
Test", None, QtGui.QApplication.UnicodeUTF8))
        self.label_9.setText(QtGui.QApplication.translate("MainWindow", "Set View:",
None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(1, QtGui.QApplication.translate("MainWindow",
"front", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(2, QtGui.QApplication.translate("MainWindow",
"back", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(3, QtGui.QApplication.translate("MainWindow", "top",
None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(4, QtGui.QApplication.translate("MainWindow",
"bottom", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(5, QtGui.QApplication.translate("MainWindow",
"left", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(6, QtGui.QApplication.translate("MainWindow",
"right", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(7, QtGui.QApplication.translate("MainWindow",
"ortho1", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(8, QtGui.QApplication.translate("MainWindow",
"ortho2", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(9, QtGui.QApplication.translate("MainWindow",
"ortho3", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(10, QtGui.QApplication.translate("MainWindow",
"ortho4", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(11, QtGui.QApplication.translate("MainWindow",
"ortho5", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(12, QtGui.QApplication.translate("MainWindow",
"ortho6", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(13, QtGui.QApplication.translate("MainWindow",
"ortho7", None, QtGui.QApplication.UnicodeUTF8))
        self.setView.setItemText(14, QtGui.QApplication.translate("MainWindow",
"ortho8", None, QtGui.QApplication.UnicodeUTF8))
        self.automate.setText(QtGui.QApplication.translate("MainWindow", "Automate",
None, QtGui.QApplication.UnicodeUTF8))
        self.loadModel.setText(QtGui.QApplication.translate("MainWindow", "Load
Model", None, QtGui.QApplication.UnicodeUTF8))
```

```
        self.loadAtlas.setText(QtGui.QApplication.translate("MainWindow", "Load
Atlas", None, QtGui.QApplication.UnicodeUTF8))
        self.defaultScene.setText(QtGui.QApplication.translate("MainWindow", "Load
Default Scene", None, QtGui.QApplication.UnicodeUTF8))
        self.clearScene.setText(QtGui.QApplication.translate("MainWindow", "Clear
Scene", None, QtGui.QApplication.UnicodeUTF8))
        self.GrabScreen.setText(QtGui.QApplication.translate("MainWindow", "Screen
Grab", None, QtGui.QApplication.UnicodeUTF8))
        self.viewTabs.setTabText(self.viewTabs.indexOf(self.openGlTab),
QtGui.QApplication.translate("MainWindow", "3D View", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_74.setText(QtGui.QApplication.translate("MainWindow", "Uptake of
Radiotracer per Bone Region", None, QtGui.QApplication.UnicodeUTF8))
        self.copyTable.setText(QtGui.QApplication.translate("MainWindow", "Copy
Table", None, QtGui.QApplication.UnicodeUTF8))
        self.saveCSV.setText(QtGui.QApplication.translate("MainWindow", "Save CSV",
None, QtGui.QApplication.UnicodeUTF8))
        self.viewTabs.setTabText(self.viewTabs.indexOf(self.tab_7),
QtGui.QApplication.translate("MainWindow", "Nuclear Medicine Breakdown", None,
QtGui.QApplication.UnicodeUTF8))
        self.dockItemList.setWindowTitle(QtGui.QApplication.translate("MainWindow",
"Scene Items List", None, QtGui.QApplication.UnicodeUTF8))
        self.dockCamera.setWindowTitle(QtGui.QApplication.translate("MainWindow",
"Camera", None, QtGui.QApplication.UnicodeUTF8))
        self.camUpdate.setText(QtGui.QApplication.translate("MainWindow", "Update",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_39.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_40.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_41.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_42.setText(QtGui.QApplication.translate("MainWindow",
"CameraPosition", None, QtGui.QApplication.UnicodeUTF8))
        self.label_43.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_44.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_45.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_46.setText(QtGui.QApplication.translate("MainWindow", "Look At",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_47.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_48.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_49.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_50.setText(QtGui.QApplication.translate("MainWindow", "Up Vector",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_59.setText(QtGui.QApplication.translate("MainWindow", "Orientation
Quaternion", None, QtGui.QApplication.UnicodeUTF8))
        self.label_55.setText(QtGui.QApplication.translate("MainWindow", "W", None,
QtGui.QApplication.UnicodeUTF8))
```

```
        self.label_56.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_57.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_58.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.dockLights.setWindowTitle(QtGui.QApplication.translate("MainWindow",
"Lights", None, QtGui.QApplication.UnicodeUTF8))
        self.label.setText(QtGui.QApplication.translate("MainWindow", "Light Number",
None, QtGui.QApplication.UnicodeUTF8))
        self.lightEnable.setText(QtGui.QApplication.translate("MainWindow", "Light
Enabled", None, QtGui.QApplication.UnicodeUTF8))
        self.label_3.setText(QtGui.QApplication.translate("MainWindow", "Diffuse",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_5.setText(QtGui.QApplication.translate("MainWindow", "Emissive",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_11.setText(QtGui.QApplication.translate("MainWindow", "Color",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_2.setText(QtGui.QApplication.translate("MainWindow", "Specular",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_4.setText(QtGui.QApplication.translate("MainWindow", "Ambient",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_6.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_7.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_8.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_10.setText(QtGui.QApplication.translate("MainWindow", "Light\n"
"Position", None, QtGui.QApplication.UnicodeUTF8))
        self.directionalLight.setText(QtGui.QApplication.translate("MainWindow",
"Directional Light", None, QtGui.QApplication.UnicodeUTF8))
        self.dockModel.setWindowTitle(QtGui.QApplication.translate("MainWindow",
"Model", None, QtGui.QApplication.UnicodeUTF8))
        self.modelVisible.setText(QtGui.QApplication.translate("MainWindow",
"Visible", None, QtGui.QApplication.UnicodeUTF8))

self.enableColorDrivenModelsCheckBox.setText(QtGui.QApplication.translate("MainWindow
", "Enable Color Driven Models", None, QtGui.QApplication.UnicodeUTF8))
        self.label_18.setText(QtGui.QApplication.translate("MainWindow", "Diffuse",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_23.setText(QtGui.QApplication.translate("MainWindow", "Emissive",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_24.setText(QtGui.QApplication.translate("MainWindow", "Color",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_17.setText(QtGui.QApplication.translate("MainWindow", "Specular",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_22.setText(QtGui.QApplication.translate("MainWindow", "Ambient",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_32.setText(QtGui.QApplication.translate("MainWindow", "Alpha",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_25.setText(QtGui.QApplication.translate("MainWindow", "Shiny",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_26.setText(QtGui.QApplication.translate("MainWindow", "Not Shiny",
None, QtGui.QApplication.UnicodeUTF8))
```

```
        self.label_27.setText(QtGui.QApplication.translate("MainWindow", "Shininess",
None, QtGui.QApplication.UnicodeUTF8))
        self.invertNormals.setText(QtGui.QApplication.translate("MainWindow", "Invert
Normals", None, QtGui.QApplication.UnicodeUTF8))
        self.saveModel.setText(QtGui.QApplication.translate("MainWindow", "Save
Model", None, QtGui.QApplication.UnicodeUTF8))
        self.label_75.setText(QtGui.QApplication.translate("MainWindow", "Point
Size", None, QtGui.QApplication.UnicodeUTF8))
        self.dockJoint.setWindowTitle(QtGui.QApplication.translate("MainWindow",
"Joint", None, QtGui.QApplication.UnicodeUTF8))
        self.label_28.setText(QtGui.QApplication.translate("MainWindow", "Rotation",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_19.setText(QtGui.QApplication.translate("MainWindow", "Order",
None, QtGui.QApplication.UnicodeUTF8))
        self.rotationOrder.setItemText(0, QtGui.QApplication.translate("MainWindow",
"xyz", None, QtGui.QApplication.UnicodeUTF8))
        self.rotationOrder.setItemText(1, QtGui.QApplication.translate("MainWindow",
"xzy", None, QtGui.QApplication.UnicodeUTF8))
        self.rotationOrder.setItemText(2, QtGui.QApplication.translate("MainWindow",
"yxz", None, QtGui.QApplication.UnicodeUTF8))
        self.rotationOrder.setItemText(3, QtGui.QApplication.translate("MainWindow",
"yzx", None, QtGui.QApplication.UnicodeUTF8))
        self.rotationOrder.setItemText(4, QtGui.QApplication.translate("MainWindow",
"zxy", None, QtGui.QApplication.UnicodeUTF8))
        self.rotationOrder.setItemText(5, QtGui.QApplication.translate("MainWindow",
"zyx", None, QtGui.QApplication.UnicodeUTF8))
        self.sphericalCoord.setText(QtGui.QApplication.translate("MainWindow",
"Spherical coordinates", None, QtGui.QApplication.UnicodeUTF8))
        self.xAxisLabel.setText(QtGui.QApplication.translate("MainWindow",
"Elevation", None, QtGui.QApplication.UnicodeUTF8))
        self.label_30.setText(QtGui.QApplication.translate("MainWindow", "Axis",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_31.setText(QtGui.QApplication.translate("MainWindow", "Degrees",
None, QtGui.QApplication.UnicodeUTF8))
        self.zAxisLabel.setText(QtGui.QApplication.translate("MainWindow", "Spin",
None, QtGui.QApplication.UnicodeUTF8))
        self.yAxisLabel.setText(QtGui.QApplication.translate("MainWindow", "Azimuth",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_51.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_52.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_53.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_60.setText(QtGui.QApplication.translate("MainWindow", "Scale",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_29.setText(QtGui.QApplication.translate("MainWindow", "All", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_34.setText(QtGui.QApplication.translate("MainWindow", "Position",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_35.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_36.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
```

```
        self.label_37.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_33.setText(QtGui.QApplication.translate("MainWindow", "Position",
None, QtGui.QApplication.UnicodeUTF8))
        self.jointXPosStep.setText(QtGui.QApplication.translate("MainWindow", "Step
X", None, QtGui.QApplication.UnicodeUTF8))
        self.label_38.setText(QtGui.QApplication.translate("MainWindow", "Step Size",
None, QtGui.QApplication.UnicodeUTF8))
        self.jointYPosStep.setText(QtGui.QApplication.translate("MainWindow", "Step
Y", None, QtGui.QApplication.UnicodeUTF8))
        self.jointZPosStep.setText(QtGui.QApplication.translate("MainWindow", "Step
Z", None, QtGui.QApplication.UnicodeUTF8))
        self.label_12.setText(QtGui.QApplication.translate("MainWindow", "Orientation
Quaternion", None, QtGui.QApplication.UnicodeUTF8))
        self.label_13.setText(QtGui.QApplication.translate("MainWindow", "W", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_14.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_15.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_16.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.dockScene.setWindowTitle(QtGui.QApplication.translate("MainWindow",
"Scene", None, QtGui.QApplication.UnicodeUTF8))
        self.label_20.setText(QtGui.QApplication.translate("MainWindow",
"Projection", None, QtGui.QApplication.UnicodeUTF8))
        self.projection.setItemText(0, QtGui.QApplication.translate("MainWindow",
"Perspective", None, QtGui.QApplication.UnicodeUTF8))
        self.projection.setItemText(1, QtGui.QApplication.translate("MainWindow",
"Orthographic", None, QtGui.QApplication.UnicodeUTF8))
        self.useCallLists.setText(QtGui.QApplication.translate("MainWindow", "Use
Call Lists", None, QtGui.QApplication.UnicodeUTF8))
        self.automateNextStep.setText(QtGui.QApplication.translate("MainWindow",
"Next Step", None, QtGui.QApplication.UnicodeUTF8))
        self.dockSlice.setWindowTitle(QtGui.QApplication.translate("MainWindow",
"Slice View Settings", None, QtGui.QApplication.UnicodeUTF8))
        self.label_21.setText(QtGui.QApplication.translate("MainWindow", "[1,0,0]
Axis", None, QtGui.QApplication.UnicodeUTF8))
        self.label_61.setText(QtGui.QApplication.translate("MainWindow", "[0,1,0]
Axis", None, QtGui.QApplication.UnicodeUTF8))
        self.label_62.setText(QtGui.QApplication.translate("MainWindow", "[0,0,1]
Axis", None, QtGui.QApplication.UnicodeUTF8))
        self.dockTol.setWindowTitle(QtGui.QApplication.translate("MainWindow",
"Tolerance Controls", None, QtGui.QApplication.UnicodeUTF8))
        self.label_65.setText(QtGui.QApplication.translate("MainWindow", "Vector
Origin", None, QtGui.QApplication.UnicodeUTF8))
        self.label_54.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_63.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_64.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_66.setText(QtGui.QApplication.translate("MainWindow", "Vector
Length", None, QtGui.QApplication.UnicodeUTF8))
```

```
        self.label_69.setText(QtGui.QApplication.translate("MainWindow", "Angle
Tolerence", None, QtGui.QApplication.UnicodeUTF8))
        self.xAxisLabel_2.setText(QtGui.QApplication.translate("MainWindow", "X",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_67.setText(QtGui.QApplication.translate("MainWindow", "Axis",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_68.setText(QtGui.QApplication.translate("MainWindow", "Degrees",
None, QtGui.QApplication.UnicodeUTF8))
        self.zAxisLabel_2.setText(QtGui.QApplication.translate("MainWindow", "Z",
None, QtGui.QApplication.UnicodeUTF8))
        self.yAxisLabel_2.setText(QtGui.QApplication.translate("MainWindow", "Y",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_70.setText(QtGui.QApplication.translate("MainWindow", "Vector",
None, QtGui.QApplication.UnicodeUTF8))
        self.label_71.setText(QtGui.QApplication.translate("MainWindow", "X", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_72.setText(QtGui.QApplication.translate("MainWindow", "Y", None,
QtGui.QApplication.UnicodeUTF8))
        self.label_73.setText(QtGui.QApplication.translate("MainWindow", "Z", None,
QtGui.QApplication.UnicodeUTF8))


if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    MainWindow = QtGui.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

STL.py

```
# -*- coding:utf-8 -*-
"""
Created on Aug 8, 2011

@author: grant
"""
import struct
import time
from TriModel import TriModel
import numpy

class Face:
	def __init__(self, strData=None):
		if strData is not None:
			self._unpackData(strData)
		else:
			self.normVector = (0.0,0.0,0.0)
			self.vertex = [(0.0,0.0,0.0), \
					(0.0,0.0,0.0), \
					(0.0,0.0,0.0)]
			self.byteCnt = 0.0
	def _unpackData(self, strData):
		data = struct.unpack('<12fH',strData)
		self.normVector = (data[0],data[1],data[2])
		self.vertex = [(data[3],data[4],data[5]), \
				(data[6],data[7],data[8]), \
				(data[9],data[10],data[11])]
		self.byteCnt = data[12]

def STLparse(fileToParse):
	header = fileToParse.read(5)
	if header.lower() == 'solid':
		#ASCII STL file
		header = fileToParse.readline()
		faces=list()
		i=0
		while (True):
			line = fileToParse.readline().lower().strip().split()
			if line[0]=='facet' and line[1]=='normal':
				faces.append(Face())
				faces[i].normVector = (float(line[2]), float(line[3]),
float(line[4]))
				vi=0
			elif line[0]=='vertex':
				faces[i].vertex[vi]=(float(line[1]), float(line[2]),
float(line[3]))
				vi+=1
			elif line[0]=='endloop':
				vi=0
			elif line[0]=='endfacet':
				i+=1
			elif line[0]=='endsolid':
				break
```

```
                triNum = i
        else:
                #binary STL file
                header+=fileToParse.read(75)
                try:
                        triNum = struct.unpack('<I',fileToParse.read(4))
                        triNum = triNum[0]
                except:
                        print 'problem'

                faces=list()
                for i in xrange(0,triNum):
                        try:
                                data = fileToParse.read(50)
                                faces.append(Face(data))
                        except:
                                print 'Error reading face'

        return header, triNum, faces

def readSTLfile(filepath, verbose=False,**kwargs):
        if verbose:
                start = time.clock()
        f = open(filepath,'rb')
        header, triNum, faces = STLparse(f)
        f.close()
        if verbose:
                print "Time to Open %s: %0.6f" % (filepath, time.clock()-start)
                print header.strip()
                print 'Number of Triangles:', triNum

#       #list method
#       vertexList = []
#       triangleVertexIndexList = []
#       normVec = []
#       for face in faces:
#               vertexList.extend(face.vertex)
#               l=len(vertexList)
#               triangleVertexIndexList.append([l-3,l-2,l-1])
#               normVec.append(face.normVector)
#       model = TriModel(vertexList, triangleVertexIndexList, normalVectors=normVec,
name=header.strip(),**kwargs)

        #numpy method, not really any faster then list method
        vertexList = numpy.zeros((len(faces)*3,3))
        triangleVertexIndexList = numpy.array(range(len(faces)*3))
        triangleVertexIndexList.shape = (len(faces),3)
        normVec = numpy.zeros((len(faces),3))
        for i in xrange(len(faces)):
                vertexList[i*3:i*3+3,:] = faces[i].vertex
                normVec[i,:] = faces[i].normVector
        if 'joint' in kwargs:
                model = TriModel(vertexList, triangleVertexIndexList, kwargs['joint'],
normalVectors=normVec, name=header.strip(),**kwargs)
        else:
```

```
            model = TriModel(vertexList, triangleVertexIndexList,
normalVectors=normVec, name=header.strip(),**kwargs)
      if 'name' in kwargs:
            model.name = kwargs['name']

      return model

def saveSTLFile(model, filepath, binary=True):
      f = open(filepath,'w')
      if binary:
            #FIXME: saving as binary is not working
            header = 80*' '
            if len(model.name)>=5 and model.name[:5].lower()=='solid':
                  h = 'binary:'+model.name
                  header[:len(h)] = h
            else:
                  h = model.name
            header = h+header[len(h):]
            header = header[:80]
            f.write(struct.pack('<80s',str(header)))
#           f.write(header)
            f.write(struct.pack('<I',len(model.TriangleVertexIndexList)))
            for i in xrange(len(model.TriangleVertexIndexList)):
                  norm = model.NormVectors[i]
                  v0 = model.VertexList[model.TriangleVertexIndexList[i,0]]
                  v1 = model.VertexList[model.TriangleVertexIndexList[i,1]]
                  v2 = model.VertexList[model.TriangleVertexIndexList[i,2]]
                  f.write(struct.pack('<12fH',norm[0],norm[1],norm[2],
                        v0[0], v0[1], v0[2], v1[0], v1[1], v1[2], v2[0], v2[1],
v2[2], 0))
      else:
            f.write('solid '+model.name+'\n')
            for i in xrange(len(model.TriangleVertexIndexList)):
                  v0 = model.VertexList[model.TriangleVertexIndexList[i,0]]
                  v1 = model.VertexList[model.TriangleVertexIndexList[i,1]]
                  v2 = model.VertexList[model.TriangleVertexIndexList[i,2]]
                  f.write('facet normal %f %f %f\nouter loop\n' %
(model.NormVectors[i,0],model.NormVectors[i,1],model.NormVectors[i,2]))
                  f.write('vertex %f %f %f\n' % (v0[0], v0[1], v0[2]))
                  f.write('vertex %f %f %f\n' % (v1[0], v1[1], v1[2]))
                  f.write('vertex %f %f %f\n' % (v2[0], v2[1], v2[2]))
                  f.write('endloop\nendfacet\n')
            f.write('endsolid '+model.name)

      f.close()
```

polygonise.py

```
'''
Created on Oct 5, 2011

@author: grant

provides python methods to perform generate isosurfaces
using marching cubes or marching tetrahedrons
'''
import numpy, time
from TriModel import TriModel

edgeTable = [
0x0  , 0x109, 0x203, 0x30a, 0x406, 0x50f, 0x605, 0x70c,
0x80c, 0x905, 0xa0f, 0xb06, 0xc0a, 0xd03, 0xe09, 0xf00,
0x190, 0x99 , 0x393, 0x29a, 0x596, 0x49f, 0x795, 0x69c,
0x99c, 0x895, 0xb9f, 0xa96, 0xd9a, 0xc93, 0xf99, 0xe90,
0x230, 0x339, 0x33 , 0x13a, 0x636, 0x73f, 0x435, 0x53c,
0xa3c, 0xb35, 0x83f, 0x936, 0xe3a, 0xf33, 0xc39, 0xd30,
0x3a0, 0x2a9, 0x1a3, 0xaa , 0x7a6, 0x6af, 0x5a5, 0x4ac,
0xbac, 0xaa5, 0x9af, 0x8a6, 0xfaa, 0xea3, 0xda9, 0xca0,
0x460, 0x569, 0x663, 0x76a, 0x66 , 0x16f, 0x265, 0x36c,
0xc6c, 0xd65, 0xe6f, 0xf66, 0x86a, 0x963, 0xa69, 0xb60,
0x5f0, 0x4f9, 0x7f3, 0x6fa, 0x1f6, 0xff , 0x3f5, 0x2fc,
0xdfc, 0xcf5, 0xfff, 0xef6, 0x9fa, 0x8f3, 0xbf9, 0xaf0,
0x650, 0x759, 0x453, 0x55a, 0x256, 0x35f, 0x55 , 0x15c,
0xe5c, 0xf55, 0xc5f, 0xd56, 0xa5a, 0xb53, 0x859, 0x950,
0x7c0, 0x6c9, 0x5c3, 0x4ca, 0x3c6, 0x2cf, 0x1c5, 0xcc ,
0xfcc, 0xec5, 0xdcf, 0xcc6, 0xbca, 0xac3, 0x9c9, 0x8c0,
0x8c0, 0x9c9, 0xac3, 0xbca, 0xcc6, 0xdcf, 0xec5, 0xfcc,
0xcc , 0x1c5, 0x2cf, 0x3c6, 0x4ca, 0x5c3, 0x6c9, 0x7c0,
0x950, 0x859, 0xb53, 0xa5a, 0xd56, 0xc5f, 0xf55, 0xe5c,
0x15c, 0x55 , 0x35f, 0x256, 0x55a, 0x453, 0x759, 0x650,
0xaf0, 0xbf9, 0x8f3, 0x9fa, 0xef6, 0xfff, 0xcf5, 0xdfc,
0x2fc, 0x3f5, 0xff , 0x1f6, 0x6fa, 0x7f3, 0x4f9, 0x5f0,
0xb60, 0xa69, 0x963, 0x86a, 0xf66, 0xe6f, 0xd65, 0xc6c,
0x36c, 0x265, 0x16f, 0x66 , 0x76a, 0x663, 0x569, 0x460,
0xca0, 0xda9, 0xea3, 0xfaa, 0x8a6, 0x9af, 0xaa5, 0xbac,
0x4ac, 0x5a5, 0x6af, 0x7a6, 0xaa , 0x1a3, 0x2a9, 0x3a0,
0xd30, 0xc39, 0xf33, 0xe3a, 0x936, 0x83f, 0xb35, 0xa3c,
0x53c, 0x435, 0x73f, 0x636, 0x13a, 0x33 , 0x339, 0x230,
0xe90, 0xf99, 0xc93, 0xd9a, 0xa96, 0xb9f, 0x895, 0x99c,
0x69c, 0x795, 0x49f, 0x596, 0x29a, 0x393, 0x99 , 0x190,
0xf00, 0xe09, 0xd03, 0xc0a, 0xb06, 0xa0f, 0x905, 0x80c,
0x70c, 0x605, 0x50f, 0x406, 0x30a, 0x203, 0x109, 0x0 ]

triTable = [[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[9, 2, 10, 0, 2, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1],
```

```
[3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 11, 2, 8, 11, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 9, 0, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 11, 2, 1, 9, 11, 9, 8, 11, -1, -1, -1, -1, -1, -1, -1],
[3, 10, 1, 11, 10, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 10, 1, 0, 8, 10, 8, 11, 10, -1, -1, -1, -1, -1, -1, -1],
[3, 9, 0, 3, 11, 9, 11, 10, 9, -1, -1, -1, -1, -1, -1, -1],
[9, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 3, 0, 7, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 1, 9, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 1, 9, 4, 7, 1, 7, 3, 1, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 10, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[3, 4, 7, 3, 0, 4, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1],
[9, 2, 10, 9, 0, 2, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1],
[2, 10, 9, 2, 9, 7, 2, 7, 3, 7, 9, 4, -1, -1, -1, -1],
[8, 4, 7, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[11, 4, 7, 11, 2, 4, 2, 0, 4, -1, -1, -1, -1, -1, -1, -1],
[9, 0, 1, 8, 4, 7, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1],
[4, 7, 11, 9, 4, 11, 9, 11, 2, 9, 2, 1, -1, -1, -1, -1],
[3, 10, 1, 3, 11, 10, 7, 8, 4, -1, -1, -1, -1, -1, -1, -1],
[1, 11, 10, 1, 4, 11, 1, 0, 4, 7, 11, 4, -1, -1, -1, -1],
[4, 7, 8, 9, 0, 11, 9, 11, 10, 11, 0, 3, -1, -1, -1, -1],
[4, 7, 11, 4, 11, 9, 9, 11, 10, -1, -1, -1, -1, -1, -1, -1],
[9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[9, 5, 4, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 5, 4, 1, 5, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[8, 5, 4, 8, 3, 5, 3, 1, 5, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 10, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[3, 0, 8, 1, 2, 10, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1],
[5, 2, 10, 5, 4, 2, 4, 0, 2, -1, -1, -1, -1, -1, -1, -1],
[2, 10, 5, 3, 2, 5, 3, 5, 4, 3, 4, 8, -1, -1, -1, -1],
[9, 5, 4, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 11, 2, 0, 8, 11, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1],
[0, 5, 4, 0, 1, 5, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1],
[2, 1, 5, 2, 5, 8, 2, 8, 11, 4, 8, 5, -1, -1, -1, -1],
[10, 3, 11, 10, 1, 3, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1],
[4, 9, 5, 0, 8, 1, 8, 10, 1, 8, 11, 10, -1, -1, -1, -1],
[5, 4, 0, 5, 0, 11, 5, 11, 10, 11, 0, 3, -1, -1, -1, -1],
[5, 4, 8, 5, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1],
[9, 7, 8, 5, 7, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[9, 3, 0, 9, 5, 3, 5, 7, 3, -1, -1, -1, -1, -1, -1, -1],
[0, 7, 8, 0, 1, 7, 1, 5, 7, -1, -1, -1, -1, -1, -1, -1],
[1, 5, 3, 3, 5, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[9, 7, 8, 9, 5, 7, 10, 1, 2, -1, -1, -1, -1, -1, -1, -1],
[10, 1, 2, 9, 5, 0, 5, 3, 0, 5, 7, 3, -1, -1, -1, -1],
[8, 0, 2, 8, 2, 5, 8, 5, 7, 10, 5, 2, -1, -1, -1, -1],
[2, 10, 5, 2, 5, 3, 3, 5, 7, -1, -1, -1, -1, -1, -1, -1],
[7, 9, 5, 7, 8, 9, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1],
[9, 5, 7, 9, 7, 2, 9, 2, 0, 2, 7, 11, -1, -1, -1, -1],
[2, 3, 11, 0, 1, 8, 1, 7, 8, 1, 5, 7, -1, -1, -1, -1],
[11, 2, 1, 11, 1, 7, 7, 1, 5, -1, -1, -1, -1, -1, -1, -1],
[9, 5, 8, 8, 5, 7, 10, 1, 3, 10, 3, 11, -1, -1, -1, -1],
[5, 7, 0, 5, 0, 9, 7, 11, 0, 1, 0, 10, 11, 10, 0, -1],
[11, 10, 0, 11, 0, 3, 10, 5, 0, 8, 0, 7, 5, 7, 0, -1],
```

```
[11, 10, 5, 7, 11, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[10, 6, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 3, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[9, 0, 1, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 8, 3, 1, 9, 8, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1],
[1, 6, 5, 2, 6, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 6, 5, 1, 2, 6, 3, 0, 8, -1, -1, -1, -1, -1, -1, -1],
[9, 6, 5, 9, 0, 6, 0, 2, 6, -1, -1, -1, -1, -1, -1, -1],
[5, 9, 8, 5, 8, 2, 5, 2, 6, 3, 2, 8, -1, -1, -1, -1],
[2, 3, 11, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[11, 0, 8, 11, 2, 0, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1],
[0, 1, 9, 2, 3, 11, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1],
[5, 10, 6, 1, 9, 2, 9, 11, 2, 9, 8, 11, -1, -1, -1, -1],
[6, 3, 11, 6, 5, 3, 5, 1, 3, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 11, 0, 11, 5, 0, 5, 1, 5, 11, 6, -1, -1, -1, -1],
[3, 11, 6, 0, 3, 6, 0, 6, 5, 0, 5, 9, -1, -1, -1, -1],
[6, 5, 9, 6, 9, 11, 11, 9, 8, -1, -1, -1, -1, -1, -1, -1],
[5, 10, 6, 4, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 3, 0, 4, 7, 3, 6, 5, 10, -1, -1, -1, -1, -1, -1, -1],
[1, 9, 0, 5, 10, 6, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1],
[10, 6, 5, 1, 9, 7, 1, 7, 3, 7, 9, 4, -1, -1, -1, -1],
[6, 1, 2, 6, 5, 1, 4, 7, 8, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 5, 5, 2, 6, 3, 0, 4, 3, 4, 7, -1, -1, -1, -1],
[8, 4, 7, 9, 0, 5, 0, 6, 5, 0, 2, 6, -1, -1, -1, -1],
[7, 3, 9, 7, 9, 4, 3, 2, 9, 5, 9, 6, 2, 6, 9, -1],
[3, 11, 2, 7, 8, 4, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1],
[5, 10, 6, 4, 7, 2, 4, 2, 0, 2, 7, 11, -1, -1, -1, -1],
[0, 1, 9, 4, 7, 8, 2, 3, 11, 5, 10, 6, -1, -1, -1, -1],
[9, 2, 1, 9, 11, 2, 9, 4, 11, 7, 11, 4, 5, 10, 6, -1],
[8, 4, 7, 3, 11, 5, 3, 5, 1, 5, 11, 6, -1, -1, -1, -1],
[5, 1, 11, 5, 11, 6, 1, 0, 11, 7, 11, 4, 0, 4, 11, -1],
[0, 5, 9, 0, 6, 5, 0, 3, 6, 11, 6, 3, 8, 4, 7, -1],
[6, 5, 9, 6, 9, 11, 4, 7, 9, 7, 11, 9, -1, -1, -1, -1],
[10, 4, 9, 6, 4, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 10, 6, 4, 9, 10, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1],
[10, 0, 1, 10, 6, 0, 6, 4, 0, -1, -1, -1, -1, -1, -1, -1],
[8, 3, 1, 8, 1, 6, 8, 6, 4, 6, 1, 10, -1, -1, -1, -1],
[1, 4, 9, 1, 2, 4, 2, 6, 4, -1, -1, -1, -1, -1, -1, -1],
[3, 0, 8, 1, 2, 9, 2, 4, 9, 2, 6, 4, -1, -1, -1, -1],
[0, 2, 4, 4, 2, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[8, 3, 2, 8, 2, 4, 4, 2, 6, -1, -1, -1, -1, -1, -1, -1],
[10, 4, 9, 10, 6, 4, 11, 2, 3, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 2, 2, 8, 11, 4, 9, 10, 4, 10, 6, -1, -1, -1, -1],
[3, 11, 2, 0, 1, 6, 0, 6, 4, 6, 1, 10, -1, -1, -1, -1],
[6, 4, 1, 6, 1, 10, 4, 8, 1, 2, 1, 11, 8, 11, 1, -1],
[9, 6, 4, 9, 3, 6, 9, 1, 3, 11, 6, 3, -1, -1, -1, -1],
[8, 11, 1, 8, 1, 0, 11, 6, 1, 9, 1, 4, 6, 4, 1, -1],
[3, 11, 6, 3, 6, 0, 0, 6, 4, -1, -1, -1, -1, -1, -1, -1],
[6, 4, 8, 11, 6, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[7, 10, 6, 7, 8, 10, 8, 9, 10, -1, -1, -1, -1, -1, -1, -1],
[0, 7, 3, 0, 10, 7, 0, 9, 10, 6, 7, 10, -1, -1, -1, -1],
[10, 6, 7, 1, 10, 7, 1, 7, 8, 1, 8, 0, -1, -1, -1, -1],
[10, 6, 7, 10, 7, 1, 1, 7, 3, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 6, 1, 6, 8, 1, 8, 9, 8, 6, 7, -1, -1, -1, -1],
[2, 6, 9, 2, 9, 1, 6, 7, 9, 0, 9, 3, 7, 3, 9, -1],
```

```
[7, 8, 0, 7, 0, 6, 6, 0, 2, -1, -1, -1, -1, -1, -1, -1],
[7, 3, 2, 6, 7, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[2, 3, 11, 10, 6, 8, 10, 8, 9, 8, 6, 7, -1, -1, -1, -1],
[2, 0, 7, 2, 7, 11, 0, 9, 7, 6, 7, 10, 9, 10, 7, -1],
[1, 8, 0, 1, 7, 8, 1, 10, 7, 6, 7, 10, 2, 3, 11, -1],
[11, 2, 1, 11, 1, 7, 10, 6, 1, 6, 7, 1, -1, -1, -1, -1],
[8, 9, 6, 8, 6, 7, 9, 1, 6, 11, 6, 3, 1, 3, 6, -1],
[0, 9, 1, 11, 6, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[7, 8, 0, 7, 0, 6, 3, 11, 0, 11, 6, 0, -1, -1, -1, -1],
[7, 11, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[7, 6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[3, 0, 8, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 1, 9, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[8, 1, 9, 8, 3, 1, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1],
[10, 1, 2, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 10, 3, 0, 8, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1],
[2, 9, 0, 2, 10, 9, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1],
[6, 11, 7, 2, 10, 3, 10, 8, 3, 10, 9, 8, -1, -1, -1, -1],
[7, 2, 3, 6, 2, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[7, 0, 8, 7, 6, 0, 6, 2, 0, -1, -1, -1, -1, -1, -1, -1],
[2, 7, 6, 2, 3, 7, 0, 1, 9, -1, -1, -1, -1, -1, -1, -1],
[1, 6, 2, 1, 8, 6, 1, 9, 8, 8, 7, 6, -1, -1, -1, -1],
[10, 7, 6, 10, 1, 7, 1, 3, 7, -1, -1, -1, -1, -1, -1, -1],
[10, 7, 6, 1, 7, 10, 1, 8, 7, 1, 0, 8, -1, -1, -1, -1],
[0, 3, 7, 0, 7, 10, 0, 10, 9, 6, 10, 7, -1, -1, -1, -1],
[7, 6, 10, 7, 10, 8, 8, 10, 9, -1, -1, -1, -1, -1, -1, -1],
[6, 8, 4, 11, 8, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[3, 6, 11, 3, 0, 6, 0, 4, 6, -1, -1, -1, -1, -1, -1, -1],
[8, 6, 11, 8, 4, 6, 9, 0, 1, -1, -1, -1, -1, -1, -1, -1],
[9, 4, 6, 9, 6, 3, 9, 3, 1, 11, 3, 6, -1, -1, -1, -1],
[6, 8, 4, 6, 11, 8, 2, 10, 1, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 10, 3, 0, 11, 0, 6, 11, 0, 4, 6, -1, -1, -1, -1],
[4, 11, 8, 4, 6, 11, 0, 2, 9, 2, 10, 9, -1, -1, -1, -1],
[10, 9, 3, 10, 3, 2, 9, 4, 3, 11, 3, 6, 4, 6, 3, -1],
[8, 2, 3, 8, 4, 2, 4, 6, 2, -1, -1, -1, -1, -1, -1, -1],
[0, 4, 2, 4, 6, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 9, 0, 2, 3, 4, 2, 4, 6, 4, 3, 8, -1, -1, -1, -1],
[1, 9, 4, 1, 4, 2, 2, 4, 6, -1, -1, -1, -1, -1, -1, -1],
[8, 1, 3, 8, 6, 1, 8, 4, 6, 6, 10, 1, -1, -1, -1, -1],
[10, 1, 0, 10, 0, 6, 6, 0, 4, -1, -1, -1, -1, -1, -1, -1],
[4, 6, 3, 4, 3, 8, 6, 10, 3, 0, 3, 9, 10, 9, 3, -1],
[10, 9, 4, 6, 10, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 9, 5, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 3, 4, 9, 5, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1],
[5, 0, 1, 5, 4, 0, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1],
[11, 7, 6, 8, 3, 4, 3, 5, 4, 3, 1, 5, -1, -1, -1, -1],
[9, 5, 4, 10, 1, 2, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1],
[6, 11, 7, 1, 2, 10, 0, 8, 3, 4, 9, 5, -1, -1, -1, -1],
[7, 6, 11, 5, 4, 10, 4, 2, 10, 4, 0, 2, -1, -1, -1, -1],
[3, 4, 8, 3, 5, 4, 3, 2, 5, 10, 5, 2, 11, 7, 6, -1],
[7, 2, 3, 7, 6, 2, 5, 4, 9, -1, -1, -1, -1, -1, -1, -1],
[9, 5, 4, 0, 8, 6, 0, 6, 2, 6, 8, 7, -1, -1, -1, -1],
[3, 6, 2, 3, 7, 6, 1, 5, 0, 5, 4, 0, -1, -1, -1, -1],
[6, 2, 8, 6, 8, 7, 2, 1, 8, 4, 8, 5, 1, 5, 8, -1],
[9, 5, 4, 10, 1, 6, 1, 7, 6, 1, 3, 7, -1, -1, -1, -1],
```

```
[1, 6, 10, 1, 7, 6, 1, 0, 7, 8, 7, 0, 9, 5, 4, -1],
[4, 0, 10, 4, 10, 5, 0, 3, 10, 6, 10, 7, 3, 7, 10, -1],
[7, 6, 10, 7, 10, 8, 5, 4, 10, 4, 8, 10, -1, -1, -1, -1],
[6, 9, 5, 6, 11, 9, 11, 8, 9, -1, -1, -1, -1, -1, -1, -1],
[3, 6, 11, 0, 6, 3, 0, 5, 6, 0, 9, 5, -1, -1, -1, -1],
[0, 11, 8, 0, 5, 11, 0, 1, 5, 5, 6, 11, -1, -1, -1, -1],
[6, 11, 3, 6, 3, 5, 5, 3, 1, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 10, 9, 5, 11, 9, 11, 8, 11, 5, 6, -1, -1, -1, -1],
[0, 11, 3, 0, 6, 11, 0, 9, 6, 5, 6, 9, 1, 2, 10, -1],
[11, 8, 5, 11, 5, 6, 8, 0, 5, 10, 5, 2, 0, 2, 5, -1],
[6, 11, 3, 6, 3, 5, 2, 10, 3, 10, 5, 3, -1, -1, -1, -1],
[5, 8, 9, 5, 2, 8, 5, 6, 2, 3, 8, 2, -1, -1, -1, -1],
[9, 5, 6, 9, 6, 0, 0, 6, 2, -1, -1, -1, -1, -1, -1, -1],
[1, 5, 8, 1, 8, 0, 5, 6, 8, 3, 8, 2, 6, 2, 8, -1],
[1, 5, 6, 2, 1, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 3, 6, 1, 6, 10, 3, 8, 6, 5, 6, 9, 8, 9, 6, -1],
[10, 1, 0, 10, 0, 6, 9, 5, 0, 5, 6, 0, -1, -1, -1, -1],
[0, 3, 8, 5, 6, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[10, 5, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[11, 5, 10, 7, 5, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[11, 5, 10, 11, 7, 5, 8, 3, 0, -1, -1, -1, -1, -1, -1, -1],
[5, 11, 7, 5, 10, 11, 1, 9, 0, -1, -1, -1, -1, -1, -1, -1],
[10, 7, 5, 10, 11, 7, 9, 8, 1, 8, 3, 1, -1, -1, -1, -1],
[11, 1, 2, 11, 7, 1, 7, 5, 1, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 3, 1, 2, 7, 1, 7, 5, 7, 2, 11, -1, -1, -1, -1],
[9, 7, 5, 9, 2, 7, 9, 0, 2, 2, 11, 7, -1, -1, -1, -1],
[7, 5, 2, 7, 2, 11, 5, 9, 2, 3, 2, 8, 9, 8, 2, -1],
[2, 5, 10, 2, 3, 5, 3, 7, 5, -1, -1, -1, -1, -1, -1, -1],
[8, 2, 0, 8, 5, 2, 8, 7, 5, 10, 2, 5, -1, -1, -1, -1],
[9, 0, 1, 5, 10, 3, 5, 3, 7, 3, 10, 2, -1, -1, -1, -1],
[9, 8, 2, 9, 2, 1, 8, 7, 2, 10, 2, 5, 7, 5, 2, -1],
[1, 3, 5, 3, 7, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 7, 0, 7, 1, 1, 7, 5, -1, -1, -1, -1, -1, -1, -1],
[9, 0, 3, 9, 3, 5, 5, 3, 7, -1, -1, -1, -1, -1, -1, -1],
[9, 8, 7, 5, 9, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[5, 8, 4, 5, 10, 8, 10, 11, 8, -1, -1, -1, -1, -1, -1, -1],
[5, 0, 4, 5, 11, 0, 5, 10, 11, 11, 3, 0, -1, -1, -1, -1],
[0, 1, 9, 8, 4, 10, 8, 10, 11, 10, 4, 5, -1, -1, -1, -1],
[10, 11, 4, 10, 4, 5, 11, 3, 4, 9, 4, 1, 3, 1, 4, -1],
[2, 5, 1, 2, 8, 5, 2, 11, 8, 4, 5, 8, -1, -1, -1, -1],
[0, 4, 11, 0, 11, 3, 4, 5, 11, 2, 11, 1, 5, 1, 11, -1],
[0, 2, 5, 0, 5, 9, 2, 11, 5, 4, 5, 8, 11, 8, 5, -1],
[9, 4, 5, 2, 11, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[2, 5, 10, 3, 5, 2, 3, 4, 5, 3, 8, 4, -1, -1, -1, -1],
[5, 10, 2, 5, 2, 4, 4, 2, 0, -1, -1, -1, -1, -1, -1, -1],
[3, 10, 2, 3, 5, 10, 3, 8, 5, 4, 5, 8, 0, 1, 9, -1],
[5, 10, 2, 5, 2, 4, 1, 9, 2, 9, 4, 2, -1, -1, -1, -1],
[8, 4, 5, 8, 5, 3, 3, 5, 1, -1, -1, -1, -1, -1, -1, -1],
[0, 4, 5, 1, 0, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[8, 4, 5, 8, 5, 3, 9, 0, 5, 0, 3, 5, -1, -1, -1, -1],
[9, 4, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 11, 7, 4, 9, 11, 9, 10, 11, -1, -1, -1, -1, -1, -1, -1],
[0, 8, 3, 4, 9, 7, 9, 11, 7, 9, 10, 11, -1, -1, -1, -1],
[1, 10, 11, 1, 11, 4, 1, 4, 0, 7, 4, 11, -1, -1, -1, -1],
[3, 1, 4, 3, 4, 8, 1, 10, 4, 7, 4, 11, 10, 11, 4, -1],
```

```
[4, 11, 7, 9, 11, 4, 9, 2, 11, 9, 1, 2, -1, -1, -1, -1],
[9, 7, 4, 9, 11, 7, 9, 1, 11, 2, 11, 1, 0, 8, 3, -1],
[11, 7, 4, 11, 4, 2, 2, 4, 0, -1, -1, -1, -1, -1, -1, -1],
[11, 7, 4, 11, 4, 2, 8, 3, 4, 3, 2, 4, -1, -1, -1, -1],
[2, 9, 10, 2, 7, 9, 2, 3, 7, 7, 4, 9, -1, -1, -1, -1],
[9, 10, 7, 9, 7, 4, 10, 2, 7, 8, 7, 0, 2, 0, 7, -1],
[3, 7, 10, 3, 10, 2, 7, 4, 10, 1, 10, 0, 4, 0, 10, -1],
[1, 10, 2, 8, 7, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 9, 1, 4, 1, 7, 7, 1, 3, -1, -1, -1, -1, -1, -1, -1],
[4, 9, 1, 4, 1, 7, 0, 8, 1, 8, 7, 1, -1, -1, -1, -1],
[4, 0, 3, 7, 4, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[4, 8, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[9, 10, 8, 10, 11, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[3, 0, 9, 3, 9, 11, 11, 9, 10, -1, -1, -1, -1, -1, -1, -1],
[0, 1, 10, 0, 10, 8, 8, 10, 11, -1, -1, -1, -1, -1, -1, -1],
[3, 1, 10, 11, 3, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 2, 11, 1, 11, 9, 9, 11, 8, -1, -1, -1, -1, -1, -1, -1],
[3, 0, 9, 3, 9, 11, 1, 2, 9, 2, 11, 9, -1, -1, -1, -1],
[0, 2, 11, 8, 0, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[3, 2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[2, 3, 8, 2, 8, 10, 10, 8, 9, -1, -1, -1, -1, -1, -1, -1],
[9, 10, 2, 0, 9, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[2, 3, 8, 2, 8, 10, 0, 1, 8, 1, 10, 8, -1, -1, -1, -1],
[1, 10, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[1, 3, 8, 9, 1, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 9, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[0, 3, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]]

class Polygonise:
      class GridCell:
            def __init__(self,points,vals):
                  self.p = points
                  self.val = vals

      def __init__(self, field, isolevel):
            self.field = numpy.array(field, dtype=numpy.float)
            self.isolevel = isolevel
            self.triangles = []

      def isosurface(self,progressDialogCallback=None,**kwargs):
            '''
            Gets isosurface mesh return trimodel
            '''
            self.grid(progressDialogCallback=progressDialogCallback)
            triVertexList = numpy.array(range(len(self.triangles)))
            triVertexList.shape = (len(self.triangles)/3,3)
            if 'joint' in kwargs:
                  model = TriModel(self.triangles,
triVertexList,kwargs['joint'],referenceVolume=self.field)
            else:
                  model = TriModel(self.triangles,
triVertexList,referenceVolume=self.field)
            return model
```

```
	def _VertexInterp(self,p1,p2,valp1,valp2):
		'''
		Linearly interpolate the position where an isosurface cuts
		an edge between two vertices, each with their own scalar value
		'''
		p1 = numpy.array(p1)
		p2 = numpy.array(p2)
		if abs(self.isolevel-valp1) < 0.00001:
			return p1
		if abs(self.isolevel-valp2) < 0.00001:
			return p2
		if abs(valp1-valp2) < 0.00001:
			return p1
		mu = (self.isolevel - valp1) / (valp2 - valp1)
		p = p1 + mu * (p2 - p1)
		return p

	def grid(self,step=1,progressDialogCallback=None):
		ni = len(self.field)-1
		nj = len(self.field[0])-1
		nk = len(self.field[0][0])-1

		n=0
		start = time.time()
		for i in xrange(0,ni,step):
			for j in xrange(0,nj,step):
				for k in xrange(0,nk,step):
					(x,y,z)=(float(i),float(j),float(k))
					p = [(x,y,z), (x+step,y,z), (x+step,y+step,z),
						(x,y+step,z) ,(x,y,z+step),
(x+step,y,z+step),
						(x+step,y+step,z+step), (x,y+step,z+step)]
					values = numpy.array(
							(self.field[p[0]], self.field[p[1]],
							 self.field[p[2]], self.field[p[3]],
							 self.field[p[4]], self.field[p[5]],
							 self.field[p[6]], self.field[p[7]]))
					n += self.MarchingCubes(self.GridCell(p, values))
				percentCompleted = 100.0*(i*nj*nk+j*nk+k)/(ni*nj*nk)
				if progressDialogCallback is not None:
					progressDialogCallback(int(percentCompleted))
				else:
#					print '%0.3f%% %d triangles' % (percentCompleted,n)
					pass
		print "Marching cube time: %f" % (time.time()-start)
		return (ni,nj,nk)

	def MarchingCubes(self,grid):
		'''
		grid is gridCell class

		Modified for python from http://paulbourke.net/geometry/polygonise/

		Given a grid cell and an isolevel, calculate the triangular
		facets required to represent the isosurface through the cell.
```

```
        Return the number of triangular facets, the array "triangles"
        will be loaded up with the vertices at most 5 triangular facets.
        0 will be returned if the grid cell is either totally above
        of totally below the isolevel.
        '''
        vertlist = numpy.zeros((12,3))

        #Determine the index into the edge table which
        #tells us which vertices are inside of the surface
        cubeindex = 0
        if grid.val[0] < self.isolevel: cubeindex |= 1
        if grid.val[1] < self.isolevel: cubeindex |= 2
        if grid.val[2] < self.isolevel: cubeindex |= 4
        if grid.val[3] < self.isolevel: cubeindex |= 8
        if grid.val[4] < self.isolevel: cubeindex |= 16
        if grid.val[5] < self.isolevel: cubeindex |= 32
        if grid.val[6] < self.isolevel: cubeindex |= 64
        if grid.val[7] < self.isolevel: cubeindex |= 128

        #Cube is entirely in/out of the surface
        if edgeTable[cubeindex] == 0:
            return 0

        #Find the vertices where the surface intersects the cube
        if edgeTable[cubeindex] & 1:
            vertlist[0] =
self._VertexInterp(grid.p[0],grid.p[1],grid.val[0],grid.val[1])
        if edgeTable[cubeindex] & 2:
            vertlist[1] =
self._VertexInterp(grid.p[1],grid.p[2],grid.val[1],grid.val[2])
        if edgeTable[cubeindex] & 4:
            vertlist[2] =
self._VertexInterp(grid.p[2],grid.p[3],grid.val[2],grid.val[3])
        if edgeTable[cubeindex] & 8:
            vertlist[3] =
self._VertexInterp(grid.p[3],grid.p[0],grid.val[3],grid.val[0])
        if edgeTable[cubeindex] & 16:
            vertlist[4] =
self._VertexInterp(grid.p[4],grid.p[5],grid.val[4],grid.val[5])
        if edgeTable[cubeindex] & 32:
            vertlist[5] =
self._VertexInterp(grid.p[5],grid.p[6],grid.val[5],grid.val[6])
        if edgeTable[cubeindex] & 64:
            vertlist[6] =
self._VertexInterp(grid.p[6],grid.p[7],grid.val[6],grid.val[7])
        if edgeTable[cubeindex] & 128:
            vertlist[7] =
self._VertexInterp(grid.p[7],grid.p[4],grid.val[7],grid.val[4])
        if edgeTable[cubeindex] & 256:
            vertlist[8] =
self._VertexInterp(grid.p[0],grid.p[4],grid.val[0],grid.val[4])
        if edgeTable[cubeindex] & 512:
            vertlist[9] =
self._VertexInterp(grid.p[1],grid.p[5],grid.val[1],grid.val[5])
        if edgeTable[cubeindex] & 1024:
```

```
                    vertlist[10] =
self._VertexInterp(grid.p[2],grid.p[6],grid.val[2],grid.val[6])
            if edgeTable[cubeindex] & 2048:
                    vertlist[11] =
self._VertexInterp(grid.p[3],grid.p[7],grid.val[3],grid.val[7])

            #Create the triangle
            ntriangles = 0;
            i=0
            while triTable[cubeindex][i] != -1:
                    self.triangles.append(vertlist[triTable[cubeindex][i]])
                    self.triangles.append(vertlist[triTable[cubeindex][i+1]])
                    self.triangles.append(vertlist[triTable[cubeindex][i+2]])
                    i+=3
                    ntriangles+=1

            return ntriangles
```

openGLUtils.py

```
# -*- coding:utf-8 -*-
"""
Created on Aug 25, 2011

@author: grant
"""
import math, numpy
#TODO: try and remove cgkit, use numpy matrix instead
from cgkit.cgtypes import quat
from OpenGL import GL, GLU

class Camera:
	def __init__(self):
		self.cameraQuat = quat(1) #identity quaternion
		self.camLoc = numpy.array([0.0,0.0,2.0])
		self.camFocus = numpy.array([0.0,0.0,0.0])
		self.camUp = numpy.array([0.0,1.0,0.0])

	def rotateAroundAxis(self, angle, axis):
		#create rotation quat based on local axis
		q=quat(angle, self.cameraQuat.rotateVec(axis))
		#rotate camera around focus point
		self.camLoc = numpy.array(q.rotateVec(self.camLoc-self.camFocus)) +
self.camFocus
		#update camera orientation
		self.cameraQuat = q*self.cameraQuat
		#calculate up direction from new orientation
		self.camUp = numpy.array(self.cameraQuat.rotateVec([0.0,1.0,0.0]))

	def zoom(self, amount):
		'''amount is multiplied by distance, 0.5 is twice as close, 2.0 is twice
as far'''
		if amount <= 0.0:
			return
		self.camLoc = (self.camLoc-self.camFocus)*amount + self.camFocus

	def pan(self, offset):
		'''offset must be in form [x,y,z]'''
		#change coordinates off offset to local orientation
		offset = self.cameraQuat.rotateVec(offset)
		self.camFocus += offset
		self.camLoc += offset

	def setView(self, *args,**kwargs):
		'''
		function signatures:
		setView(self, vMin, vMax, view='ortho1')
		setView(self, orientation, position, focusPoint):
		'''
		if isinstance(args[0],quat) and len(args)>=3:
			self.cameraQuat = args[0]
			self.camLoc = numpy.array(args[1])
			self.camFocus = numpy.array(args[2])
```

```
                    self.camUp =
numpy.array(self.cameraQuat.rotateVec([0.0,1.0,0.0]))
              elif len(args) >= 2:
                    vMin = numpy.array(args[0])
                    vMax = numpy.array(args[1])
                    view = 'ortho1'
                    if 'view' in kwargs:
                          view = kwargs['view']

                    yAngle = None
                    xAngle = None
                    diameter = 2.0 * numpy.sqrt(numpy.sum((vMax-vMin)**2))
                    self.camFocus = (vMax+vMin)/2.0
                    if view=='top':
                          yAngle = 0.0
                          xAngle = -0.5*math.pi
                    elif view=='bottom':
                          yAngle = 0.0
                          xAngle = 0.5*math.pi
                    elif view=='right':
                          yAngle = 0.5*math.pi
                          xAngle = 0.0
                    elif view=='left':
                          yAngle = -0.5*math.pi
                          xAngle = 0.0
                    elif view=='front':
                          yAngle = 0.0
                          xAngle = 0.0
                    elif view=='back':
                          yAngle = math.pi
                          xAngle = 0.0
                    elif view=='ortho1':
                          yAngle = 0.25*math.pi
                          xAngle = -0.25*math.pi
                    elif view=='ortho2':
                          yAngle = 0.75*math.pi
                          xAngle = -0.25*math.pi
                    elif view=='ortho3':
                          yAngle = 1.25*math.pi
                          xAngle = -0.25*math.pi
                    elif view=='ortho4':
                          yAngle = 1.75*math.pi
                          xAngle = -0.25*math.pi
                    elif view=='ortho5':
                          yAngle = 0.75*math.pi
                          xAngle = 0.25*math.pi
                    elif view=='ortho6':
                          yAngle = 1.25*math.pi
                          xAngle = 0.25*math.pi
                    elif view=='ortho7':
                          yAngle = 1.75*math.pi
                          xAngle = 0.25*math.pi
                    elif view=='ortho8':
                          yAngle = 0.25*math.pi
                          xAngle = 0.25*math.pi
```

```
                  if yAngle is not None and xAngle is not None:
                        #create quat as rotation around y then around (local)x
                        startQuat = quat(yAngle, [0,1,0])
#                       startQuat = quat(numpy.pi,
startQuat.rotateVec([0,1,0]))*startQuat
                        self.cameraQuat = quat(xAngle,
startQuat.rotateVec([1,0,0]))*startQuat
                        self.camLoc =
self.cameraQuat.rotateVec([0.0,0.0,diameter])
                        self.camLoc += self.camFocus
                        self.camUp =
numpy.array(self.cameraQuat.rotateVec([0.0,1.0,0.0]))

      def openGLTransform(self):
            GLU.gluLookAt(self.camLoc[0], self.camLoc[1], self.camLoc[2],
                              self.camFocus[0], self.camFocus[1],
self.camFocus[2],
                              self.camUp[0], self.camUp[1], self.camUp[2])

class OpenGlLight:
      def __init__(self, lightNumber, enabled=False):
            self.enabled = enabled
            self.lightNumber = lightNumber
            self.specularColor = [0.0,0.0,0.0,0.0]
            self.diffuseColor = [0.0,0.0,0.0,1.0]
            self.ambientColor = [0.0,0.0,0.0,1.0]
            self.emissiveColor = [0.0,0.0,0.0,0.0]
            self.position = [0.0,0.0,1.0,0.0]
            self.directional = True

      def updateOpenGl(self):
            if self.enabled:
                  GL.glEnable(GL.GL_LIGHT0+self.lightNumber)

                  GL.glLightfv (GL.GL_LIGHT0+self.lightNumber, GL.GL_DIFFUSE,
self.diffuseColor)
                  GL.glLightfv (GL.GL_LIGHT0+self.lightNumber, GL.GL_AMBIENT,
self.ambientColor)
                  GL.glLightfv (GL.GL_LIGHT0+self.lightNumber, GL.GL_SPECULAR,
self.specularColor)
                  if self.directional:
                        self.position[3] = 0.0
                  else:
                        self.position[3] = 1.0
                  GL.glLightfv (GL.GL_LIGHT0+self.lightNumber, GL.GL_POSITION,
self.position)
            else:
                  GL.glDisable(GL.GL_LIGHT0+self.lightNumber)
```

MatlabFunctions.py

```
'''
Created on Feb 29, 2012

@author: Jeff
'''
import mlabraw

class MatlabFmincon():
	def __init__(self):
		self.mlab = mlabraw.open('')
		mlabraw.eval(self.mlab, "clear all")
		mlabraw.eval(self.mlab,
"cd('C:/Users/Jeff/Dropbox/Thesis/software/pyAtlasSegmentation/src')")
		mlabraw.eval(self.mlab, 'global modelPoints dataPoints x0 lb ub')

	def __del__(self):
		mlabraw.close(self.mlab)

	def minimize(self, modelPoints, dataPoints, x0, lb, ub, **kwargs):
		mlabraw.put(self.mlab, 'modelPoints', modelPoints.T)
		mlabraw.put(self.mlab, 'dataPoints', dataPoints.T)
		mlabraw.put(self.mlab, 'x0', x0)
		mlabraw.put(self.mlab, 'lb', lb)
		mlabraw.put(self.mlab, 'ub', ub)

		if 'scaleOnly' in kwargs and kwargs['scaleOnly']:
			mlabraw.eval(self.mlab, '[R, T, S] = mlabMinScale()')
		elif 'scaleOnlyIso' in kwargs and kwargs['scaleOnlyIso']:
			mlabraw.eval(self.mlab, '[R, T, S] = mlabMinScaleIso()')
		else:
			mlabraw.eval(self.mlab, '[R, T, S] = mlabMin()')

		R = mlabraw.get(self.mlab, 'R')
		T = mlabraw.get(self.mlab, 'T')
		S = mlabraw.get(self.mlab, 'S')

		return R, T, S
```

matlabcom.py

```
#!/usr/bin/env python

""" A python module for raw communication with Matlab(TM) using COM client
 under windows.

The module sends commands to the matlab process as a COM client. This is only
supported under windows.

Author: Dani Valevski <daniva@gmail.com>
Dependencies: pywin32, numpy
Tested Matlab Versions: 2011a
License: MIT
"""

import numpy as np
try:
	import win32com.client
except:
	print 'win32com in missing, please install it'
	raise

class MatlabError(Exception):
	"""Raised when a Matlab evaluation results in an error inside Matlab."""
	pass

class MatlabConnectionError(Exception):
	"""Raised for errors related to the Matlab connection."""
	pass

class MatlabCom(object):
	""" Manages a matlab COM client.

	The process can be opened and close with the open/close methods.
	To send a command to the matlab shell use 'eval'.
	To load numpy data to the matlab shell use 'put'.
	To retrieve numpy data from the matlab shell use 'get'.
	"""
	def __init__(self):
		self.client = None

	def open(self, visible=False):
		""" Dispatches the matlab COM client.

		Note: If this method fails, try running matlab with the -regserver flag.
		"""
		if self.client:
			raise MatlabConnectionError('Matlab(TM) COM client is still
active. Use close to '
                                                                'close it')
		self.client = win32com.client.Dispatch('matlab.application')
		self.client.visible = visible

	def close(self):
```

```
            """ Closes the matlab COM client.
            """
            self._check_open()
            try:
                  self.eval('quit();')
            except:
                  pass
            del self.client

      def eval(self, expression, identify_erros=True):
            """ Evaluates a matlab expression synchronously.

            If identify_erros is true, and the last output line after evaluating the
            expressions begins with '???' an excpetion is thrown with the matlab
error
            following the '???'.
            The return value of the function is the matlab output following the
call.
            """
            #print expression
            self._check_open()
            ret = self.client.Execute(expression)
            #print ret
            if identify_erros and ret.rfind('???') != -1:
                  begin = ret.rfind('???') + 4
                  end = ret.find('\n', begin)
                  raise MatlabError(ret[begin:end])
            return ret

      def get(self, names_to_get, convert_to_numpy=True):
            """ Loads the requested variables from the matlab com client.

            names_to_get can be either a variable name or a list of variable names.
            If it is a variable name, the values is returned.
            If it is a list, a dictionary of variable_name -> value is returned.

            If convert_to_numpy is true, the method will all array values to numpy
            arrays. Scalars are left as regular python objects.

            """
            self._check_open()
            single_itme = isinstance(names_to_get, (unicode, str))
            if single_itme:
                  names_to_get = [names_to_get]
            ret = {}
            for name in names_to_get:
                  ret[name] = self.client.GetWorkspaceData(name, 'base')
                  # TODO(daniv): Do we really want to reduce dimensions like that?
what if this a row vector?
                  while isinstance(ret[name], (tuple, list)) and len(ret[name]) ==
1:
                        ret[name] = ret[name][0]
                  if convert_to_numpy and isinstance(ret[name], (tuple, list)):
                        ret[name] = np.array(ret[name])
            if single_itme:
```

```
                return ret.values()[0]
            return ret

        def put(self, name_to_val):
            """ Loads a dictionary of variable names into the matlab com client.

            """
            self._check_open()
            for name, val in name_to_val.iteritems():
                # First try to put data as a matrix:
                try:
                    self.client.PutFullMatrix(name, 'base', val, None)
                except:
                    self.client.PutWorkspaceData(name, 'base', val)

        def _check_open(self):
            if not self.client:
                raise MatlabConnectionError('Matlab(TM) process is not active.')

if __name__ == '__main__':
        import unittest


        class TestMatlabCom(unittest.TestCase):
            def setUp(self):
                self.matlab = MatlabCom()
                self.matlab.open()

            def tearDown(self):
                self.matlab.close()

            def test_eval(self):
                for i in xrange(100):
                    ret = self.matlab.eval('disp \'hiush world%s\';' %
('b'*i))
                    self.assertTrue('hiush world' in ret)

            def test_put(self):
                self.matlab.put({'A' : [1, 2, 3]})
                ret = self.matlab.eval('A')
                self.assertTrue('A =' in ret)

            def test_1_element(self):
                self.matlab.put({'X': 'string'})
                ret = self.matlab.get('X')
                self.assertEquals(ret, 'string')

            def test_get(self):
                self.matlab.eval('A = [1 2 3];')
                ret = self.matlab.get('A')
                self.assertEquals(ret[0], 1)
                self.assertEquals(ret[1], 2)
                self.assertEquals(ret[2], 3)

            def test_error(self):
```

```
            self.assertRaises(MatlabError,
self.matlab.eval,
'no_such_function')
unittest.main()
```

ICP.py

```
'''
Created on Feb 15, 2012

@author: Jeff
'''
import numpy
import numpyTransform
from scipy.spatial import cKDTree as KDTree
#from scipy.spatial import Delaunay
from scipy.spatial.distance import cdist
import scipy.optimize
import time
from math import pi
from MatlabFunctions import MatlabFmincon
import nlopt
import sys

class ICP(object):
	'''
	classdocs
	'''


	def __init__(self, modelPointCloud, dataPointCloud, **kwargs):
		'''
		Supported Signatures
		modelPointCloud
			The model point cloud is the base to which the data point cloud
will be matched
		dataPointCloud
			The data point cloud is transformed so that it matches the model
point cloud

		Key Word Arguments:
		maxIterations
			maximum number of iterations to perform, default is 10
			TODO: in the future provide an option to also account for minimum
acceptable error
		matchingMethod
			'kdtree'		Use a KD-Tree for nearest neighbor search
{default}
			'bruteforce' Use brute force for nearest neighbor search
		minimizeMethod
			'point'			Use point to point minimization
{default}
			'plane'			Use point to plane minimization
		weightMethod
			function that takes indices into the modelPointCloud and returns
the weight of those indices
			By default all points are weighted equally
		modelDownsampleFactor
			integer that represents uniform sampling of model point cloud
```

```
                1 is no resampling, 2 is every other point, 3 is every third
point...
            dataDownsampleFactor
                integer that represents uniform sampling of model point cloud
                1 is no resampling, 2 is every other point, 3 is every third
point...


            ICP Process is five steps
                1: Input Filter
                2: Match
                3: Outlier Filter
                4: Error Minimization
                5: Check if error is less than limits
                    yes: we are don
                    no: go back to step 2 with new transformation function
            '''
            self.startTime = time.time()

            if 'modelDownsampleFactor' in kwargs and
int(kwargs['modelDownsampleFactor']) > 1:
                factor = int(kwargs['modelDownsampleFactor'])
                temp = numpy.zeros(factor, dtype=numpy.bool)
                temp[-1] = True
                modelDownSampleIndices = numpy.tile(temp,
(modelPointCloud.shape[0]/factor)+1)[:modelPointCloud.shape[0]]
            else:
                modelDownSampleIndices = numpy.ones(modelPointCloud.shape[0],
dtype=numpy.bool)
            if 'dataDownsampleFactor' in kwargs and
int(kwargs['dataDownsampleFactor']) > 1:
                factor = int(kwargs['dataDownsampleFactor'])
                temp = numpy.zeros(factor, dtype=numpy.bool)
                temp[-1] = True
                dataDownSampleIndices = numpy.tile(temp,
(dataPointCloud.shape[0]/factor)+1)[:dataPointCloud.shape[0]]
            else:
                dataDownSampleIndices = numpy.ones(dataPointCloud.shape[0],
dtype=numpy.bool)


            #TODO: uniform sampling of point clouds
            self.q = modelPointCloud[modelDownSampleIndices]
            self.p = dataPointCloud[dataDownSampleIndices]
            self.matlab = None

            #get kwargs
            if 'maxIterations' in kwargs:
                self.K = int(kwargs['maxIterations'])
            else:
                self.K = 10
            if 'matchingMethod' in kwargs:
                if kwargs['matchingMethod'] == 'bruteforce':
                    self.matching = self.matchingBruteForce
                else:
```

```
                        self.matching = self.matchingKDTree
                        self.qKDTree = KDTree(self.q)
                else:
                        self.matching = self.matchingKDTree
                        self.qKDTree = KDTree(self.q)

                if 'minimizeMethod' in kwargs:
                        if kwargs['minimizeMethod'] == 'plane':#point to plane
                                self.minimize = self.minimizePlane
                        elif kwargs['minimizeMethod'] == 'fmincon':
                                self.minimize = self.minimizeMatlab
                                self.matlab = MatlabFmincon()
                        elif kwargs['minimizeMethod'] == 'custom':
                                self.minimize = self.minimizeCustom
                        else:  #point to point
                                self.minimize = self.minimizePoint
                else:
                        self.minimize = self.minimizePoint

                if 'weightMethod' in kwargs:
                        self.weightMethod = kwargs['weightMethod']
                else:
                        self.weightMethod = self.weightEqual


                #initialize translation and rotation matrix
                self.transformMatrix = numpy.matrix(numpy.identity(4))
                #initialize list of translations and rotation matrix for each iteration
of ICP
                self.totalTransformMatrix = [numpy.matrix(numpy.identity(4))]

                self.pt = self.p.copy()   #transformed point cloud
                self.t = []  #array of times for each iteration of ICP
                self.err = []#error for each iteration of ICP
                self.Np = self.p.shape[0] #number of points in data cloud

                #preprocessing finish, log time
                self.t.append(time.time()-self.startTime)
                print 'Time for preprocessing:', self.t[-1]

        def __del__(self):
                if self.matlab is not None:
                        del self.matlab

        def runICP(self, **kwargs):
                tStart = time.time()

                #get 'global' tolerances
                if 'x0' in kwargs:
                        kwargs['initX0'] = kwargs['x0'].copy()
                if 'lb' in kwargs:
                        kwargs['initLB'] = kwargs['lb'].copy()
                if 'ub' in kwargs:
                        kwargs['initUB'] = kwargs['ub'].copy()
```

```
            #main ICP loop
            for k in xrange(self.K):
                  t1 = time.time()
                  minDistances, nearestNeighbor = self.matching(self.pt)

                  #get indices of the points we are interested in
                  p_idx = numpy.ones(self.p.shape[0], dtype=numpy.bool)      #since
there are no edges we are interested in all the points
                  q_idx = nearestNeighbor
                  print '\tTime to calc min distance:', time.time() - t1

                  #TODO: Input filtering
                  #reject some % of worst matches
                  #Multiresolution sampling

                  #add error for first iteration
                  if k == 0:
                        t1 = time.time()
                        self.err.append(
numpy.sqrt(numpy.sum(minDistances**2)/minDistances.shape[0]) )
                        print '\tInitial RMS error: %f, Time to calc: %f' %
(self.err[-1], time.time()-t1)

                  #generate rotation matrix and translation
                  t1 = time.time()
                  weights = self.weightMethod(nearestNeighbor)

                  #get current cumulative rotation/translation in independent
variable values, this way we can change the iteration bounds so that the global
bounds are not violated
                  cummulativeX0 = numpy.zeros(9)
                  rotMat, tx, ty, tz, sx, sy,sz =
numpyTransform.decomposeMatrix(self.totalTransformMatrix[-1])
                  rx, ry, rz = numpyTransform.rotationMat2Euler(rotMat)
                  cummulativeX0[0] = rx
                  cummulativeX0[1] = ry
                  cummulativeX0[2] = rz
                  cummulativeX0[3] = tx
                  cummulativeX0[4] = ty
                  cummulativeX0[5] = tz
                  cummulativeX0[6] = sx
                  cummulativeX0[7] = sy
                  cummulativeX0[8] = sz

                  R, T, S = self.minimize(self.q[q_idx], self.pt[p_idx],
weights=weights, cummulativeX0=cummulativeX0, **kwargs)
                  print '\tTime to calc new transformation:', time.time()-t1

                  #create combined transformation matrix, apply this relative
transformation to current transformation
                  transformMatrix = numpy.matrix(numpy.identity(4))
                  transformMatrix *= T
                  transformMatrix *= R
                  transformMatrix *= S
```

```
                    self.totalTransformMatrix.append( self.totalTransformMatrix[-1] *
transformMatrix)

                    #apply last transformation
                    t1 = time.time()
                    self.pt =
numpyTransform.transformPoints(self.totalTransformMatrix[-1], self.p)
                    print '\tTime to applying transform to all points:', time.time()-
t1

                    #root mean of objective function
                    t1=time.time()
                    self.err.append( self.rms_error(self.q[q_idx], self.pt[p_idx]))
                    print '\tIteration %d RMS error: %f, Time to calc: %f' % (k+1,
self.err[-1], time.time()-t1)

                    #TODO: add extrapolation

                    #store time to get to this iteration
                    self.t.append(time.time()-self.startTime)
                    print 'Iteration %d took %7.3f seconds' % (k+1, self.t[-1]-
self.t[-2])

             print 'Total ICP run time:', time.time() - tStart
             return self.totalTransformMatrix, self.err, self.t

      def matchingKDTree(self, points):
             minDistances,nearestNeighborIndex = self.qKDTree.query(points)
             return minDistances,nearestNeighborIndex

      def matchingBruteForce(self, points):
             nearestNeighborIndex = numpy.zeros(points.shape[0])
             distances = cdist(points, self.q)      #calculate all combination of
point distances
             minDistances = distances.min(axis=1)
             for i in xrange(points.shape[0]):
                    nearestNeighborIndex[i] = numpy.where(distances[i] ==
minDistances[i])[0][0]
             return minDistances, nearestNeighborIndex

      def minimizePoint(self, q, p, **kwargs):
             R = numpy.matrix(numpy.identity(4))
             T = numpy.matrix(numpy.identity(4))
             S = numpy.matrix(numpy.identity(4))

             if 'weights' in kwargs:
                    weights = kwargs['weights']
             else:
                    raise Warning('weights argument not supplied')
                    return R, T
#            function [R,T] = eq_point(q,p,weights)
             m = p.shape[0]
             n = q.shape[0]

             # normalize weights
```

```
            weights = weights / weights.sum()

            # find data centroid and deviations from centroid
            q_bar = (numpy.mat(q.T) *
numpy.mat(weights[:,numpy.newaxis])).getA().squeeze()
            q_mark = q - numpy.tile(q_bar, n).reshape( (n,3) )
            # Apply weights
            q_mark = q_mark * numpy.repeat(weights, 3).reshape((weights.shape[0],3))

            # find data centroid and deviations from centroid
            p_bar = (numpy.mat(p.T) *
numpy.mat(weights[:,numpy.newaxis])).getA().squeeze()
            p_mark = p - numpy.tile(p_bar, m).reshape( (m,3) )
            # Apply weights
            #p_mark = p_mark * numpy.repeat(weights,
3).reshape((weights.shape[0],3))

            N = (numpy.mat(p_mark).T * numpy.mat(q_mark)).getA() # taking points of
q in matched order

            [U,Ss,V] = numpy.linalg.svd(N); # singular value decomposition
            V = (numpy.mat(V).H).getA()

            RMattemp = numpy.mat(V)*numpy.mat(U).T

            Ttemp = (numpy.mat(q_bar).T -
RMattemp*numpy.mat(p_bar).T).getA().squeeze()

            R[:3,:3] = RMattemp.getA()
            T = numpyTransform.translation(Ttemp)

            return R, T, S

      def minimizeMatlab(self, modelPoints, dataPoints, **kwargs):
            if 'x0' in kwargs:
                  x0 = kwargs['x0']
            else:
                  raise Exception('There are no variables to solve for')

            #check for initial settings and bounds so that we can calculate current
settings and bounds
            if 'initX0' in kwargs:
                  initX0 = kwargs['initX0']
            if 'initLB' in kwargs:
                  initLB = kwargs['initLB']
            if 'initUB' in kwargs:
                  initUB = kwargs['initUB']
            if 'cummulativeX0' in kwargs:
                  cummulativeX0 = kwargs['cummulativeX0']

            #NOTE: I think this only works if x0/initX) is all zeros
            ub = initUB - (cummulativeX0-initX0)
            lb = initLB - (cummulativeX0-initX0)
            #rounding errors can cause Bounds to be incorrect
            i = ub < x0
```

```
            if numpy.any(i):
                  print 'upper bounds less than x0'
            ub[i]= x0[i] + 10*numpy.spacing (x0[i])
            i = lb > x0
            if numpy.any(i):
                  print 'lower bounds less than x0'
            lb[i]= x0[i] - 10*numpy.spacing (x0[i])


#           if x0.shape[0] > 6 or ('scaleOnly' in kwargs and kwargs['scaleOnly']):
#                 raise Exception('Scaling is not currently supported it will screw
things up. Need some way to control scaling bounds so that it stays in global scaling
bounds')
            try:
                  if 'scaleOnly' in kwargs:
                        R, T, S = self.matlab.minimize(modelPoints, dataPoints,
x0[-3:], lb[-3:], ub[-3:], scaleOnly=kwargs['scaleOnly'])
                  elif 'scaleOnlyIso' in kwargs:
                        R, T, S = self.matlab.minimize(modelPoints, dataPoints,
x0[-1:], lb[-1:], ub[-1:], scaleOnlyIso=kwargs['scaleOnlyIso'])
                  else:
                        R, T, S = self.matlab.minimize(modelPoints, dataPoints,
x0[:6], lb[:6], ub[:6])   #only rotation and translation
            except:
                  sys.stderr.write('ERROR: Problem with matlab, closing matlab\n')
                  del self.matlab
                  self.matlab = None

            return R, T, S

      def minimizeCustom(self,p,q,**kwargs):
            S = numpy.matrix(numpy.identity(4))
            #TODO: try using functions from the nlopt module
            def objectiveFunc(*args, **kwargs):
                  d = p
                  m = q
                  params = args[0]
                  if args[1].size > 0:      #gradient
                        args[1][:] = numpy.array([pi/100, pi/100, pi/100, 0.01,
0.01, 0.01]) #arbitrary gradient

#                 transform = numpy.matrix(numpy.identity(4))
                  translate = numpyTransform.translation(params[3:6])
                  rotx = numpyTransform.rotation(params[0], [1,0,0], N=4)
                  roty = numpyTransform.rotation(params[1], [0,1,0], N=4)
                  rotz = numpyTransform.rotation(params[2], [0,0,1], N=4)
                  transform = translate * rotx * roty * rotz

                  Dicp = numpyTransform.transformPoints(transform, d)

#                 err = self.rms_error(m, Dicp)
                  err = numpy.mean(numpy.sqrt(numpy.sum((m - Dicp) ** 2, axis=1)))
#                 err = numpy.sqrt(numpy.sum((m - Dicp) ** 2, axis=1))
                  return err
```

```
            x0 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
            if 'optAlg' in kwargs:
                  opt = nlopt.opt(kwargs['optAlg'], 6)
            else:
                  opt = nlopt.opt(nlopt.GN_CRS2_LM, 6)

            opt.set_min_objective(objectiveFunc)
            opt.set_lower_bounds([-pi, -pi, -pi, -3.0, -3.0, -3.0])
            opt.set_upper_bounds([pi, pi, pi, 3.0, 3.0, 3.0])
            opt.set_maxeval(1500)
            params = opt.optimize(x0)

#           output = scipy.optimize.leastsq(objectiveFunc, x0, args=funcArgs)
#           params = output[0]

#           params = scipy.optimize.fmin(objectiveFunc, x0, args=funcArgs)

#           constraints = []
#           varBounds = [(-pi, pi), (-pi, pi), (-pi, pi), (-3.0, 3.0), (-3.0, 3.0),
(-3.0, 3.0)]
#           params = scipy.optimize.fmin_slsqp(objectiveFunc, x0,
eqcons=constraints, bounds=varBounds, args=funcArgs)

#           output = scipy.optimize.fmin_l_bfgs_b(objectiveFunc, x0,
bounds=varBounds, args=funcArgs, approx_grad=True)
#           params = output[0]
#           print  'Min error:', output[1]

#           params = scipy.optimize.fmin_tnc(objectiveFunc, x0, bounds=varBounds,
args=funcArgs, approx_grad=True)
#           params = scipy.optimize.fmin_slsqp(objectiveFunc, x0,
eqcons=constraints, bounds=varBounds, args=funcArgs)
#           params = scipy.optimize.fmin_slsqp(objectiveFunc, x0,
eqcons=constraints, bounds=varBounds, args=funcArgs)

            translate = numpyTransform.translation(params[3:6])
            rotx = numpyTransform.rotation(params[0], [1,0,0], N=4)
            roty = numpyTransform.rotation(params[1], [0,1,0], N=4)
            rotz = numpyTransform.rotation(params[2], [0,0,1], N=4)
            transform = translate * rotx * roty * rotz
            return rotx * roty * rotz, S

      def minimizePlane(self, p, q, **kwargs):
            #TODO: Actually fill out
            R = numpy.matrix(numpy.identity(4))
            T = numpy.matrix(numpy.identity(4))
            S = numpy.matrix(numpy.identity(4))

#           function [R,T] = eq_plane(q,p,n,weights)
#           n = n .* repmat(weights,3,1);
#
#           c = cross(p,n);
#
#           cn = vertcat(c,n);
#
```

```
#             C = cn*transpose(cn);
#
#             b = - [sum(sum((p-q).*repmat(cn(1,:),3,1).*n));
#                    sum(sum((p-q).*repmat(cn(2,:),3,1).*n));
#                    sum(sum((p-q).*repmat(cn(3,:),3,1).*n));
#                    sum(sum((p-q).*repmat(cn(4,:),3,1).*n));
#                    sum(sum((p-q).*repmat(cn(5,:),3,1).*n));
#                    sum(sum((p-q).*repmat(cn(6,:),3,1).*n))];
#
#             X = C\b;
#
#             cx = cos(X(1)); cy = cos(X(2)); cz = cos(X(3));
#             sx = sin(X(1)); sy = sin(X(2)); sz = sin(X(3));
#
#             R = [cy*cz cz*sx*sy-cx*sz cx*cz*sy+sx*sz;
#                    cy*sz cx*cz+sx*sy*sz cx*sy*sz-cz*sx;
#                    -sy cy*sx cx*cy];
#
#             T = X(4:6);

              return R, T, S

       def weightEqual(self, qIndices):
              return numpy.ones(qIndices.shape[0])

       def rms_error(self, a, b):
              '''
              Determine the RMS error between two point equally sized point clouds
with point correspondence.
              NOTE: a and b need to have equal number of points
              '''
              if a.shape[0] != b.shape[0]:
                     raise Exception('Input Point clouds a and b do not have the same
number of points')

              distSq = numpy.sum((a-b)**2, axis=1)
              err = numpy.sqrt(numpy.mean(distSq))
              return err

def demo(*args, **kwargs):
       import math
       m = 80 # width of grid
       n = m**2 # number of points

       minVal = -2.0
       maxVal = 2.0
       delta = (maxVal-minVal)/(m-1)
       X,Y = numpy.mgrid[minVal:maxVal+delta:delta, minVal:maxVal+delta:delta]

       X = X.flatten()
       Y = Y.flatten()

       Z = numpy.sin(X) * numpy.cos(Y)

       # Create the data point-matrix
```

```
        M = numpy.array([X, Y, Z]).T

        # Translation values (a.u.):
        Tx = 0.5
        Ty = -0.3
        Tz = 0.2

        # Translation vector
        T = numpyTransform.translation(Tx, Ty, Tz)

        S = numpyTransform.scaling(1.0, N=4)

        # Rotation values (rad.):
        rx = 0.3
        ry = -0.2
        rz = 0.05

        Rx = numpy.matrix([[1, 0, 0,0],
                                   [0, math.cos(rx), -math.sin(rx),0],
                                   [0, math.sin(rx), math.cos(rx),0],
                                   [0,0,0,1]])

        Ry = numpy.matrix([[math.cos(ry), 0, math.sin(ry),0],
                                         [0, 1, 0,0],
                                         [-math.sin(ry), 0, math.cos(ry),0],
                                         [0,0,0,1]])

        Rz = numpy.matrix([[math.cos(rz), -math.sin(rz), 0,0],
                                         [math.sin(rz), math.cos(rz), 0,0],
                                         [0, 0, 1,0],
                                         [0,0,0,1]])

        # Rotation matrix
        R = Rx*Ry*Rz

        transformMat = numpy.matrix(numpy.identity(4))
        transformMat *= T
        transformMat *= R
        transformMat *= S

        # Transform data-matrix plus noise into model-matrix
        D = numpyTransform.transformPoints(transformMat, M)

        # Add noise to model and data
        M = M + 0.01*numpy.random.randn(n,3)
        D = D + 0.01*numpy.random.randn(n,3)

        # Run ICP (standard settings)
        initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
        lowerBounds = numpy.array([-pi, -pi, -pi, -100.0, -100.0, -100.0])
        upperBounds = numpy.array([pi, pi, pi, 100.0, 100.0, 100.0])
        icp = ICP(M, D, maxIterations=15, dataDownsampleFactor=1,
minimizeMethod='fmincon', **kwargs)
#       icp = ICP(M, D, maxIterations=15, dataDownsampleFactor=1,
minimizeMethod='point', **kwargs)
```

```
transform, err, t = icp.runICP(x0=initialGuess, lb=lowerBounds,ub=upperBounds)

# Transform data-matrix using ICP result
Dicp = numpyTransform.transformPoints(transform[-1], D)

# Plot model points blue and transformed points red
if False:
	import matplotlib.pyplot as plt
	from mpl_toolkits.mplot3d import Axes3D
	fig = plt.figure()
	ax = fig.add_subplot(2,2,1, projection='3d')
	ax.scatter(M[:,0],M[:,1],M[:,2], c='r', marker= 'o')
	ax.scatter(D[:,0],D[:,1],D[:,2], c='b', marker= '^')
	ax.set_xlabel('X Label')
	ax.set_ylabel('Y Label')
	ax.set_zlabel('Z Label')

	ax = fig.add_subplot(2,2,2, projection='3d')
	ax.scatter(M[:,0],M[:,1],M[:,2], c='r', marker= 'o')
	ax.scatter(Dicp[:,0],Dicp[:,1],Dicp[:,2], c='b', marker= '^')
	ax.set_xlabel('X Label')
	ax.set_ylabel('Y Label')
	ax.set_zlabel('Z Label')

	ax = fig.add_subplot(2,2,3)
	ax.plot(t,err,'x--')
	ax.set_xlabel('X Label')
	ax.set_ylabel('Y Label')

	plt.show()
else:
	import visvis as vv
	app=vv.use()
	vv.figure()
	vv.subplot(2,2,1)
	vv.plot(M[:,0],M[:,1],M[:,2],lc='b', ls='', ms='o')
	vv.plot(D[:,0],D[:,1],D[:,2],lc='r', ls='', ms='x')
	vv.xlabel('[0,0,1] axis')
	vv.ylabel('[0,1,0] axis')
	vv.zlabel('[1,0,0] axis')
	vv.title('Red: z=sin(x)*cos(y), blue: transformed point cloud')

	# Plot the results
	vv.subplot(2,2,2)
	vv.plot(M[:,0],M[:,1],M[:,2],lc='b', ls='', ms='o')
	vv.plot(Dicp[:,0],Dicp[:,1],Dicp[:,2],lc='r', ls='', ms='x')
	vv.xlabel('[0,0,1] axis')
	vv.ylabel('[0,1,0] axis')
	vv.zlabel('[1,0,0] axis')
	vv.title('ICP result')

	# Plot RMS curve
	vv.subplot(2,2,3)
	vv.plot(t,err,ls='--', ms='x')
	vv.xlabel('time [s]')
```

```
            vv.ylabel('d_{RMS}')
            vv.title('KD-Tree matching')
            if 'optAlg' in kwargs:
                  opt2=nlopt.opt(kwargs['optAlg'],2)
                  vv.title(opt2.get_algorithm_name())
                  del opt2
            else:
                  vv.title('KD-Tree matching')
            app.Run()

if __name__ == '__main__':
      demo()
#     demo2()
```

Joint.py

```
# -*- coding:utf-8 -*-
"""
Created on Aug 8, 2011

@author: grant
"""
import math
import numpy
from OpenGL import GL, GLU
import TriModel
#TODO: try and remove cgkit, use numpy matrix instead
from cgkit.cgtypes import quat
import numpyTransform

class Joint:
	def __init__(self, *args, **kwargs):
		'''
		Function Signatures:
		Joint()
		Joint(location, ...)

		Arguments {default value}:

		location
			array like object of form [x,y,z] containing x, y, and z
coordinates
			of this Joint. { [0,0,0] }

		Keywords:
		parentJoint
			Joint object of which this Joint is a child. {None}
		models
			TriModel object of list of TriModel objects that represent bones
			that are attached to this joint. { [] }
		name
			Name of joint
		initialOrientation
			quaternion (type cgkit.cgtypes.quat) representing the initial
			orientation. { quat(1,0,0,0) }
		showAxis
			Boolean that determines whether or not a 3D representation of the
			Joint is visible. { False }
		axisScale
			Number that determines the size of the drawn axis. Must be
greater
			than 0. { 1.0 }
		'''
#		self.translateMat = cgtypes.mat4.identity()
		self.scaleMat = numpy.matrix(numpy.identity(4))
		self.rotateMat = numpy.matrix(numpy.identity(4))

		if len(args) > 0:
			self.location = numpy.array(args[0], dtype=float)
```

```
		else:
			self.location = numpy.array([0.0,0.0,0.0], dtype=float)
		self.initialLocationMat = numpy.matrix([[1.0,0.0,0.0,self.location[0]],

	[0.0,1.0,0.0,self.location[1]],

	[0.0,0.0,1.0,self.location[2]],

	[0.0,0.0,0.0,1.0]])
		self.translateMat = numpy.matrix([	[1.0,0.0,0.0,self.location[0]],

	[0.0,1.0,0.0,self.location[1]],

	[0.0,0.0,1.0,self.location[2]],

	[0.0,0.0,0.0,1.0]])
		self.transformICP = numpy.matrix(numpy.identity(4))


		self.childJoints = []
#		self.scale = 1.0
		self.locationUnityScale = self.location.copy()
		self.type = 'ball'
		self.length = 10.0
		self.proximodistalVec = numpy.array([1.0,0.0,0.0])
		self.secondaryVec = numpy.array([0.0,1.0,0.0])
		self.tertiaryVec = numpy.array([0.0,0.0,1.0])
		self.proximodistalVecTransformed = numpy.array([1.0,0.0,0.0])
		self.secondaryVecTransformed = numpy.array([0.0,1.0,0.0])
		self.tertiaryVecTransformed = numpy.array([0.0,0.0,1.0])
		self.proximodistalVecScaled = self.length * self.proximodistalVec
		self.proximodistalVecTransformedScaled = self.length *
self.proximodistalVecTransformed
		self.DOFvec = numpy.array([0,0,10.0])
		self.DOFangle = math.radians(45.0)
		self.DOFtrans = 5.0

		if 'parentJoint' in kwargs and isinstance(kwargs['parentJoint'],Joint):
			self.parentJoint = kwargs['parentJoint']
			self.parentJoint.childJoints.append(self)
		else:
			self.parentJoint = None

		self.models = []
		if 'models' in kwargs:
			try:
				for model in kwargs['models']:
					if isinstance(model, TriModel):
						self.models.append(model)
						self.models[-1].setJoint(self)
			except TypeError:
				if isinstance(kwargs['models'], TriModel):
					self.models.append(kwargs['models'])
					self.models[-1].setJoint(self)
```

```
            if 'initialOrientation' in kwargs and
isinstance(kwargs['initialOrientation'], quat):
                  self.orientation = kwargs['initialOrientation']
            else:
                  self.orientation = quat(1,0,0,0)
            self.xAngle = 0.0
            self.yAngle = 0.0
            self.zAngle = 0.0

            if 'showAxis' in kwargs and isinstance(kwargs['showAxis'],bool):
                  self.showAxis = kwargs['showAxis']
            else:
                  self.showAxis = False
            if 'axisScale' in kwargs and kwargs['axisScale'] > 0.0:
                  self.axisScale = kwargs['axisScale']
            else:
                  self.axisScale = 1.0
            if 'name' in kwargs:
                  self.name = kwargs['name']
            else:
                  self.name = 'Joint'

            for model in self.models:
                  model.initialRotationCenter = self.location.copy()

            if self.parentJoint == None:
                  self.initalLocationRelativeToParentJoint = self.location.copy()
                  self.initialRelativeOrientationFromParent =
self.orientation*quat(1,0,0,0).inverse()
            else:
                  self.initalLocationRelativeToParentJoint = self.location -
self.parentJoint.location
                  self.initialRelativeOrientationFromParent =
self.orientation*self.parentJoint.orientation.inverse()
            self.relativeOrientationFromParent =
quat(self.initialRelativeOrientationFromParent)

            self.createAxis(self.showAxis)

      def translate(self,coord,absolute=True):
            '''
            Arguements {Default value}
            coord
                  array like object with dimensions 1x3 in format [x,y,z]
            absolute
                  boolean that tells whether coord is the point to set joint to,
                  or the step by with to move the joint from its current location
            '''
            coord = numpy.array(coord)
            if coord.shape != (3,):
                  raise Exception("Incorrect input parameters")
            if absolute:
                  self.translateMat = numpyTransform.translation(coord)
            else:
                  self.translateMat *= numpyTransform.translation(coord)
```

```
	def rotate(self,*args, **kwargs):
		'''
		Function Signatures:
		rotate(q, ...)
		rotate(mat, ...)
		rotate(angle, axis, ...)
		rotate(xAngle, yAngle, zAngle, ...)
		rotate(elevation, azimuth, spin, sphericalCoord=True, ...)

		Arguments {default value}:
		q
			quaternion (cgkit.cgtypes.quat) that defines joint orientation
			(relative or absolute)
		mat
			4x4 rotation matrix (cgkit.cgtypes.mat4) that defines joint
orientation
		angle
			angle (degrees or radians, radians default) which to rotate
around
			axis
		axis
			vector [i,j,k] which defines axis to rotate around
		xAngle
			angle (degrees or radians, radians default) which to rotate
around
			x axis
		yAngle
			angle (degrees or radians, radians default) which to rotate
around
			y axis
		zAngle
			angle (degrees or radians, radians default) which to rotate
around
			z axis
		elevation
			elevation angle (spherical coordinate system)
		azimuth
			azimuth angle (spherical coordinate system)
		spin
			spin angle around vector defined by elevation and azimuth

		Keywords:
		relative
			defines whether the change in orientation is relative or absolute
			{False}
		updateModels
			flag that determines if child models should be updated {True}
		angleOrder
			string that determines the order that Euler angle rotations are
			applied. {'xyz'}
		unitsDegrees
			flag that indicates what units the passed angles are in, degrees
or
			radians. {False}
```

```
            sphericalCoord
                  flag that indicates passed angles are spherical coordinates. In
the
                  spherical coordinate system, elevation rotates around the x axis
                  first, then azimuth rotates around the y axis, finally spin
rotates
                  around the vector created by elevation and azimuth {False}
            '''

            if 'relative' in kwargs:
                  relative = kwargs['relative']
            else:
                  relative = False
            if 'updateModels' in kwargs:
                  updateModels = kwargs['updateModels']
            else:
                  updateModels = True
            if 'sphericalCoord' in kwargs:
                  sphericalCoord = kwargs['sphericalCoord']
            else:
                  sphericalCoord = False

            #the baseOrientation of the joint is either its current orientation,
relativeAngle=True
            #or the baseOrientation is the initial relative orientation from the
parent joint, relativeAngle=False
            if relative:
                  baseOrientation = quat(self.orientation)
            #change by original orientation difference, to regain original
orientation, relative to parent
            elif self.parentJoint == None:
                  baseOrientation =
self.initialRelativeOrientationFromParent*quat(1,0,0,0)
            else:
                  baseOrientation =
self.initialRelativeOrientationFromParent*self.parentJoint.orientation

            if len(args) == 1:  #Quaternion rotation
                  if isinstance(args[0], quat):
                        rotateMat = args[0].toMat4()
                  else:
                        rotateMat = args[0]
            elif len(args) == 2:      #angle and axis rotation
                  angle = args[0]
                  axis = args[1]

                  #convert angle units to radians if required
                  if 'unitsDegrees' in kwargs and kwargs['unitsDegrees']:
                        angle = math.radians(angle)
                  angle = NormalizeAngleRad(angle)

                  #create rotate matrix
                  rotateMat = numpyTransform.rotation(angle, axis, N=4)

            elif len(args) == 3:      #Euler angle rotation
```

```
                        xAngle = args[0]
                        yAngle = args[1]
                        zAngle = args[2]
                        if 'angleOrder' in kwargs and not sphericalCoord:
                                angleOrder = kwargs['angleOrder'].lower()
                                if len(angleOrder) != 3 or angleOrder.find('x') < 0 or
angleOrder.find('y') < 0 or angleOrder.find('z') < 0:
                                        raise Exception('invalid angle order string')
                        else:
                                angleOrder = 'xyz'
                        if 'unitsDegrees' in kwargs and kwargs['unitsDegrees']:
                                xAngle = math.radians(xAngle)
                                yAngle = math.radians(yAngle)
                                zAngle = math.radians(zAngle)
                        if 'relative' in kwargs and kwargs['relative']:
                                self.xAngle += xAngle
                                self.yAngle += yAngle
                                self.zAngle += zAngle
                        else:
                                self.xAngle = xAngle
                                self.yAngle = yAngle
                                self.zAngle = zAngle
                        if sphericalCoord:
                                self.xAngle = NormalizeAngleRad(self.xAngle,-math.pi/2,
math.pi/2, math.pi)
                                self.yAngle = NormalizeAngleRad(self.yAngle)
                                self.zAngle = NormalizeAngleRad(self.zAngle)
                        else:
                                self.xAngle = NormalizeAngleRad(self.xAngle)
                                self.yAngle = NormalizeAngleRad(self.yAngle)
                                self.zAngle = NormalizeAngleRad(self.zAngle)

                        rotateMat = numpy.matrix(numpy.identity(4))
                        #create orientation quaternion by multiplying rotations around
local
                        #x,y, and z axis
                        #TODO: maybe flip the order of this rotation application?
                        #FIXME: Spherical rotation not working
                        for i in xrange(3):
                                if angleOrder[i] == 'x':
                                        rotateMat *=
numpyTransform.rotation(self.xAngle,[1.0,0.0,0.0], N=4)
                                if angleOrder[i] == 'y':
                                        rotateMat *=
numpyTransform.rotation(self.yAngle,[0.0,1.0,0.0], N=4)
                                if angleOrder[i] == 'z':
                                        rotateMat *=
numpyTransform.rotation(self.zAngle,[0.0,0.0,1.0], N=4)

                else:  #invalid signature
                        raise Exception("Invalid Function Signature")

                if relative:
                        self.rotateMat *= rotateMat
                else:
```

```
                        self.rotateMat = rotateMat

#       def setScale(self,*args,**kwargs):
#               #TODO:remove after mat4 transformation is done, also rename
scaleTempname to scale, remote scale float value
#               self.scaleTempname(*args,**kwargs)

        def scale(self,*args,**kwargs):
                '''
                Arguments {Default value}
                scale(scale, ...)
                scale(scaleX,scaleY,scaleZ, ...)
                scale([scaleX,scaleY,scaleZ], ...)

                scale
                        scale in the X,Y, and Z direction
                scaleX
                        scale in the X dimension
                scaleY
                        scale in the Y dimension
                scaleZ
                        scale in the Z dimension

                keyword arguments
                absolute        {True}
                        boolean that tells whether scale is the new scale (True)
                        or an amount to adjust current scale by (False)
                '''
                if len(args)==1:
                        scale = numpy.array(args[0],dtype=float)
                        if scale.shape != (3,):
                                if scale.shape == ():
                                        scale = numpy.repeat(scale, 3)
                                else:
                                        scale = numpy.repeat(scale[0], 3)
                elif len(args)==3:
                        scale = numpy.array([args[0],args[1],args[2]],dtype=float)
                if 'absolute' in kwargs and kwargs['absolute']==False:
                        self.scaleMat *= numpyTransform.scaling(scale, N=4)
                else:
                        self.scaleMat = numpyTransform.scaling(scale, N=4)

        def createAxis(self, axisVisible):
                self.xAxis = TriModel.createCone(self.axisScale/4, self.axisScale,
self.location, name='axis_X', joint=self,
axis='x',updateOnlyFromGrandparentJoints=True, visible=axisVisible,
color=[1.0,0.0,0.0,1.0])
                self.yAxis = TriModel.createCone(self.axisScale/4, self.axisScale,
self.location, name='axis_Y', joint=self,
axis='y',updateOnlyFromGrandparentJoints=True, visible=axisVisible,
color=[0.0,1.0,0.0,1.0])
                self.zAxis = TriModel.createCone(self.axisScale/4, self.axisScale,
self.location, name='axis_Z', joint=self,
axis='z',updateOnlyFromGrandparentJoints=True, visible=axisVisible,
color=[0.0,0.0,1.0,1.0])
```

```
	def OpenGLPaint(self, colorDrivenMaterial=None, useCallLists =
True,parentTransform=numpy.matrix(numpy.identity(4))):
		#push matrix
		GL.glPushMatrix()

		#matrix transformations steps: (applied in reverse order)
		#1: move model initial rotation center to origin
		#2: scale model
		#3: rotate model to new orientation
		#4: move model to parent joint position
		if self.name == 'Neck':
			pass

	GL.glTranslatef(self.translateMat[0,3],self.translateMat[1,3],self.translateMa
t[2,3])	#aka GL.glMultMatrixf(numpy.array(self.translateMat))

		GL.glMultMatrixf(numpy.array(self.rotateMat).T)	#need to transpose
this because numpy matrices are row-major but OpenGl is expecting column-major matrix
#		axis, angle = numpyTransform.axisAngleFromMatrix(self.rotateMat,
angleInDegrees=True)
#		GL.glRotated(angle, axis[0], axis[1], axis[2])
		GL.glScalef(self.scaleMat[0,0],self.scaleMat[1,1],self.scaleMat[2,2])
		GL.glTranslatef(-self.initialLocationMat[0,3],-
self.initialLocationMat[1,3],-self.initialLocationMat[2,3])

#		if self.name == 'Neck':
#			print 'Neck Draw Transform'
#			print 'Rotation:'
#			print self.rotateMat
#			print 'Translation:'
#			print self.translateMat
#			print 'Scale'
#			print self.scaleMat
#			print 'Original location'
#			print self.initialLocationMat
#			print 'Transform'
#			tform = self.translateMat * self.rotateMat * self.scaleMat *
self.initialLocationMat.I
#			print tform

		#draw models
		for model in self.models:
			model.OpenGLPaint(colorDrivenMaterial, useCallLists)
		#recursivly paint child joints
		for childJoint in self.childJoints:
			childJoint.OpenGLPaint(colorDrivenMaterial, useCallLists)
		#pop matrix
		GL.glPopMatrix()

	def transformVertices(self, baseTransform = numpy.matrix(numpy.identity(4)),
modelID=None):
		#create transform matrix
		transform = numpy.matrix(baseTransform)
		transform *= self.translateMat
```

```
            transform  *= self.rotateMat
            transform *= self.scaleMat
            transform *= self.initialLocationMat.I #            transform *=
numpyTransform.translation( (-self.initialLocationMat[0,3], -
self.initialLocationMat[1,3], -self.initialLocationMat[2,3]) )

#            self.location = (transform * numpy.matrix([[self.locationUnityScale[0]],
[self.locationUnityScale[1]], [self.locationUnityScale[2]],
[1.0]])).getA().squeeze()[:3]
            self.location = numpyTransform.transformPoints(transform,
self.locationUnityScale)

            transformRotScaleOnly = numpy.matrix(numpy.identity(4))
            transformRotScaleOnly[:3,:3] = transform[:3,:3]
            self.proximodistalVecTransformed =
numpyTransform.transformPoints(transformRotScaleOnly,
self.proximodistalVec[numpy.newaxis,:]).squeeze()
            self.proximodistalVecTransformedScaled =
numpyTransform.transformPoints(transformRotScaleOnly, self.length *
self.proximodistalVec[numpy.newaxis,:]).squeeze()
            self.secondaryVecTransformed =
numpyTransform.transformPoints(transformRotScaleOnly,
self.secondaryVec[numpy.newaxis,:]).squeeze()
            self.tertiaryVecTransformed =
numpyTransform.transformPoints(transformRotScaleOnly,
self.tertiaryVec[numpy.newaxis,:]).squeeze()

            for model in self.models:
                  model.transformVertices(transform, modelID)

            for childJoint in self.childJoints:
                  childJoint.transformVertices(transform, modelID)

      def getCummulativeTransform(self, jointID, baseTransform =
numpy.matrix(numpy.identity(4))):
            transform = numpy.matrix(baseTransform)
            transform *= self.translateMat
            transform  *= self.rotateMat
            transform *= self.scaleMat
            transform *= self.initialLocationMat.I

            if id(self) == jointID:
                  return transform
            else:
                  retTform = None
                  for childJoint in self.childJoints:
                        tempTform = childJoint.getCummulativeTransform(jointID,
transform)
                        if tempTform is not None:
                              retTform = tempTform
                              break
                  return retTform

      def createKDTrees(self):
            for model in self.models:
```

```
                  if model.visible and model.name[:5] != 'axis_':     #ignore
models that are not visible and models that illustrate the axis
                        model.createKDTrees()
            for childJoint in self.childJoints:
                  childJoint.createKDTrees()

      def getBoundingBox(self, baseTransform = numpy.matrix(numpy.identity(4))):
            #TODO: change this to use lists and then use numpy to search for max and
min along axis=0 instead of constantly comparing values
            points = []
            minPoint = None
            maxPoint = None

            #create transform matrix
            transform = numpy.matrix(baseTransform)
            transform *= self.translateMat
            transform  *= self.rotateMat
            transform *= self.scaleMat
            transform *= self.initialLocationMat.I #            transform *=
numpyTransform.translation( (-self.initialLocationMat[0,3], -
self.initialLocationMat[1,3], -self.initialLocationMat[2,3]) )

            for model in self.models:
                  if model.visible:
                        vMin2, vMax2 = model.getBoundingBox(transform)
                        if vMin2 is not None and vMax2 is not None:
                              points.append(vMin2)
                              points.append(vMax2)

            for childJoint in self.childJoints:
                  vMin2, vMax2 = childJoint.getBoundingBox(transform)
                  if vMin2 is not None and vMax2 is not None:
                        points.append(vMin2)
                        points.append(vMax2)

            if len(points) > 1:
                  points = numpy.array(points)
                  minPoint = numpy.min(points,axis=0)
                  maxPoint = numpy.max(points,axis=0)

            return minPoint, maxPoint

      def compareToTransformedPoints(self, point, currentClosestSqDistance=None,
currentModelID=None, modelName=''):
            for model in self.models:
                  if model.visible and model.name[:5] != 'axis_':     #ignore
models that are not visible and models that illustrate the axis
                        currentClosestSqDistance, currentModelID, modelName =
model.compareToTransformedPoints(point, currentClosestSqDistance, currentModelID,
modelName)
            for childJoint in self.childJoints:
                  currentClosestSqDistance, currentModelID, modelName =
childJoint.compareToTransformedPoints(point, currentClosestSqDistance,
currentModelID, modelName)
            return currentClosestSqDistance, currentModelID, modelName
```

```
	def compareToTransformedPointsKDTrees(self, point,
currentClosestSqDistance=None, currentModelID=None, modelName=''):
		for model in self.models:
			if model.visible and model.name[:5] != 'axis_':		#ignore
models that are not visible and models that illustrate the axis
				currentClosestSqDistance, currentModelID, modelName =
model.compareToTransformedPointsKDTrees(point, currentClosestSqDistance,
currentModelID, modelName)
		for childJoint in self.childJoints:
			currentClosestSqDistance, currentModelID, modelName =
childJoint.compareToTransformedPointsKDTrees(point, currentClosestSqDistance,
currentModelID, modelName)
		return currentClosestSqDistance, currentModelID, modelName

def NormalizeAngleRad(angle, minimum=0.0, maximum = 2*math.pi,delta=2*math.pi):
	while angle > maximum:
		angle -= delta
	while angle < minimum:
		angle += delta
	return angle

if __name__ == '__main__':
	pass
```

InputDataTypes.py

```
'''
Created on Jan 29, 2012

@author: Jeff
'''
import os.path
import numpy, scipy
from scipy.ndimage import gaussian_filter, map_coordinates
from PySide import QtGui, QtCore
import dicom
import TriModel
from Joint import Joint
from polygonise import Polygonise
from STL import readSTLfile, saveSTLFile
from OpenGL import GL
import time
from lxml import etree
from scipy.ndimage.measurements import label
from scipy.spatial import cKDTree as KDTree

class UserAtlas():
	'''
	classdocs
	'''
	def __init__(self, filepath=None, joint=None, **kwargs):
		'''
		filepath
			filepath to xml file that describes the atlas

		joint
			joint

		Key Words:
		joint
		'''
		if filepath is None or not isinstance(filepath, str) or not
os.path.exists(filepath):
			filepath, flter =
QtGui.QFileDialog.getOpenFileName(caption="Select an Atlas xml file to load",
filter="XML (*.xml);;Any File (*.*)") #@UnusedVariable
		if len(filepath) == 0:
			return

		if joint is None or not isinstance(joint, Joint):
			return

		#set up variables
		self.atlasAxes = numpy.identity(3)
		self.atlasModels = {}

		#Start loading atlas
		xmldoc = etree.parse(filepath)
		root = xmldoc.getroot()
```

```
            numBones = 0
            for axesElement in root.iter(tag=etree.Element):
                  if axesElement.tag == 'axes':
                        for vectorElement in axesElement:
                              if vectorElement.tag == 'anteriorVector':
                                    v = vectorElement.text[1:-1].split(',')
                                    self.atlasAxes[0] =
[float(v[0]),float(v[1]),float(v[2])]
                              elif vectorElement.tag == 'dorsalVector':
                                    v = vectorElement.text[1:-1].split(',')
                                    self.atlasAxes[1] =
[float(v[0]),float(v[1]),float(v[2])]
                              elif vectorElement.tag == 'rightVector':
                                    v = vectorElement.text[1:-1].split(',')
                                    self.atlasAxes[2] =
[float(v[0]),float(v[1]),float(v[2])]
                        break  #should only be one 'axes' element

            for element in root.iter('bone'): #@UnusedVariable
                  numBones += 1
            progress = QtGui.QProgressDialog("Loading Atlas...", "Abort", 0,
numBones, None)
            progress.setWindowModality(QtCore.Qt.WindowModal)
            atlasBasePosition = [0.0,0.0,0.0]
            for posElement in root.iter(tag=etree.Element):
                  if posElement.tag == 'position':
                        p=posElement.text[1:-1].split(',')
                        atlasBasePosition=[float(p[0]),float(p[1]),float(p[2])]
                        break

            self.atlasJoint = Joint(atlasBasePosition,name='Atlas
Base',parentJoint=joint)
            self._loadAtlasJointInfo(self.atlasJoint, root)
            self._loadAtlas(root, self.atlasJoint, filepath, progress)
            progress.setValue(numBones)

      def _loadAtlasJointInfo(self, joint, jointElement):
            for e in jointElement.iter(tag=etree.Element):
                  if e.tag == 'jointType':
                        joint.type = e.text.strip()
                        break
            for e in jointElement.iter(tag=etree.Element):
                  if e.tag == 'proximodistalVec':
                        v=e.text.strip()[1:-1].split(',')
                        joint.proximodistalVec =
numpy.array([float(v[0]),float(v[1]),float(v[2])])
                        joint.proximodistalVec /=
numpy.linalg.norm(joint.proximodistalVec)     #normalize
                        joint.proximodistalVecTransformed =
joint.proximodistalVec.copy()
                        break
            for e in jointElement.iter(tag=etree.Element):
                  if e.tag == 'secondaryVec':
                        v=e.text.strip()[1:-1].split(',')
```

```
                        joint.secondaryVec =
numpy.array([float(v[0]),float(v[1]),float(v[2])])
                        joint.secondaryVec /=
numpy.linalg.norm(joint.secondaryVec)  #normalize
                        joint.secondaryVecTransformed = joint.secondaryVec.copy()
                        break
            for e in jointElement.iter(tag=etree.Element):
                  if e.tag == 'tertiaryVec':
                        v=e.text.strip()[1:-1].split(',')
                        joint.tertiaryVec =
numpy.array([float(v[0]),float(v[1]),float(v[2])])
                        joint.tertiaryVec /= numpy.linalg.norm(joint.tertiaryVec)
      #normalize
                        joint.tertiaryVecTransformed = joint.tertiaryVec.copy()
                        break
            for e in jointElement.iter(tag=etree.Element):
                  if e.tag == 'length':
                        joint.length = float(e.text.strip())
                        break
            for e in jointElement.iter(tag=etree.Element):
                  if e.tag == 'DOFvec':
                        v=e.text.strip()[1:-1].split(',')
                        joint.DOFvec =
numpy.array([float(v[0]),float(v[1]),float(v[2])])
                        break
            for e in jointElement.iter(tag=etree.Element):
                  if e.tag == 'DOFangle':
                        joint.DOFangle = numpy.radians(float(e.text.strip()))
                        break
            for e in jointElement.iter(tag=etree.Element):
                  if e.tag == 'DOFtrans':
                        joint.DOFtrans = float(e.text.strip())
                        break

            joint.proximodistalVecScaled = joint.length * joint.proximodistalVec

      def _loadAtlas(self,parentElement, parentJoint, origfilepath=None,
progressDialog=None):
            for element in parentElement:
                  if progressDialog is not None and progressDialog.wasCanceled():
                        break
                  #attach bones to joint
                  if element.tag == 'bone':
                        #find bone model filepath
                        filepath=None
                        for pathElement in element:
                              if pathElement.tag == 'filepath':
                                    filepath = pathElement.text
                                    break
                        if os.path.exists(filepath) is False and origfilepath is
not None:
                              #convert relative filepath (from xml
location) to absolute filepaths
                              head, tail = os.path.split(origfilepath)
#@UnusedVariable
```

```
                                    filepath = os.path.join(head, filepath)
                              if os.path.exists(filepath) is True:
                                    for colorElement in element:
                                          color=None
                                          if colorElement.tag == 'color':
                                                c=colorElement.text[1:-1].split(',')
                                                if len(c) >= 3:
                                                      color =
[float(c[0])/255.0,float(c[1])/255.0,float(c[2])/255.0]
                                                      if len(c) >= 4:

      color.append(float(c[3])/255.0)
                                    visible = True
                                    for visibleElement in element:
                                          if visibleElement.tag == 'visible':
                                                if visibleElement.text == 'False':
                                                      visible = False
                                    model = readSTLfile(filepath,
True,color=color,visible=visible)
                                    model.name = element.text.strip()
                                    model.setJoint(parentJoint)
                                    self.atlasModels[model.name] = model
                                    print 'Model ID %d is %s' % (id(model), model.name)

                              #update progress dialog
                              if progressDialog is not None:
                                    progressDialog.setValue(progressDialog.value()+1)

                        if element.tag == 'joint':
                              pos = None
                              #search for position
                              for posElement in element:
                                    if posElement.tag == 'position':
                                          p=posElement.text[1:-1].split(',')
                                          pos=[float(p[0]),float(p[1]),float(p[2])]
                                          break
                              if pos is None:
                                    continue
                              #create joint
                              joint = Joint(pos, parentJoint=parentJoint, showAxis =
True, axisScale=0.5,name=element.text.strip())
                              self._loadAtlasJointInfo(joint, element)
                              #recursively call load atlas to add children joints
                              self._loadAtlas(element, joint,
origfilepath,progressDialog)

      def resetAtlas(self):
            #TODO: create resetAtlas function
            pass

      def createSTL(self, filepath=''):
#             transformedVertexList =numpy.array([])
#             TriangleVertexIndexList = numpy.array([])
#             NormVectors = numpy.array([])
            self._createSTL(self.atlasJoint)
```

```
#            transformedVertexList, TriangleVertexIndexList, NormVectors =
self._getModelData(self.atlasJoint, transformedVertexList, TriangleVertexIndexList,
NormVectors)
#            #TODO: make trimodel from data
#            model = TriModel.TriModel(transformedVertexList,
TriangleVertexIndexList, joint=self.atlasJoint, normalVectors=NormVectors)
#            saveSTLFile(model, filepath, binary=False)

      def _createSTL(self, joint):
            for model in joint.models:
                  if model.visible and model.name[:5] != 'axis_':     #ignore
models that are not visible and models that illustrate the axis
                        newModel =
TriModel.TriModel(model.transformedVertexList.copy(),
model.TriangleVertexIndexList.copy(), normalVectors=model.NormVectors.copy())
                        filepath = 'Atlas Bone %d %s.stl' % (id(model),
model.name)
                        saveSTLFile(newModel, filepath, binary=False)
                        print 'Saved STL file ',filepath
            for childJoint in joint.childJoints:
                  self._createSTL(childJoint)

      def _getModelData(self, joint, transformedVertexList, TriangleVertexIndexList,
NormVectors):
            for model in joint.models:
                  if model.visible and model.name[:5] != 'axis_':     #ignore
models that are not visible and models that illustrate the axis
                        #get new transformedVertexList
                        n1 = transformedVertexList.shape[0]
                        n2 = model.transformedVertexList.shape[0]
                        transformedVertexList =
numpy.append(transformedVertexList, model.transformedVertexList)
                        transformedVertexList.shape = (n1+n2, 3)
                        #get new VertexList
                        n1 = TriangleVertexIndexList.shape[0]
                        n2 = model.TriangleVertexIndexList.shape[0]
                        TriangleVertexIndexList =
numpy.append(TriangleVertexIndexList, model.TriangleVertexIndexList)
                        TriangleVertexIndexList.shape = (n1+n2, 3)
                        #get new NormVectors
                        n1 = NormVectors.shape[0]
                        n2 = model.NormVectors.shape[0]
                        NormVectors = numpy.append(NormVectors, model.NormVectors)
                        NormVectors.shape = (n1+n2, 3)
            for childJoint in joint.childJoints:
                  transformedVertexListJ, TriangleVertexIndexListJ, NormVectorsJ =
self._getModelData(childJoint, transformedVertexList, TriangleVertexIndexList,
NormVectors)
                  #get new transformedVertexList
                  n1 = transformedVertexList.shape[0]
                  n2 = transformedVertexListJ.shape[0]
                  transformedVertexList = numpy.append(transformedVertexList,
transformedVertexListJ)
                  transformedVertexList.shape = (n1+n2, 3)
                  #get new VertexList
```

```
                    n1 = TriangleVertexIndexList.shape[0]
                    n2 = TriangleVertexIndexListJ.shape[0]
                    TriangleVertexIndexList = numpy.append(TriangleVertexIndexList,
TriangleVertexIndexListJ)
                    TriangleVertexIndexList.shape = (n1+n2, 3)
                    #get new NormVectors
                    n1 = NormVectors.shape[0]
                    n2 = NormVectorsJ.shape[0]
                    NormVectors = numpy.append(NormVectors, NormVectorsJ)
                    NormVectors.shape = (n1+n2, 3)

             return transformedVertexList, TriangleVertexIndexList, NormVectors

class UserDicom():
      '''
      classdocs
      '''

      def __init__(self, ct=None, nm=None, **kwargs):
            '''
            ct
                  filepath or dicom object of uCT data

            nm
                  filepath or dicom object of uSPECT or uPET data, this data is
fused with uCT data

            Key Words:
            joint
            sigma
            resampleFactor
            isolevel
            '''

            if ct is None:
                  ct, flter = QtGui.QFileDialog.getOpenFileName(caption="Select a
uCT dicom file to load", filter="DICOM (*.dcm);;Any File (*.*)") #@UnusedVariable
                  if len(ct) == 0:
                        return
            if nm is None:
                  nm, flter = QtGui.QFileDialog.getOpenFileName(caption="Select a
uSPECT or uPET dicom file to load", filter="DICOM (*.dcm);;Any File (*.*)")
#@UnusedVariable
                  if len(nm) == 0:
                        return
            if isinstance(ct, (str, unicode)) and os.path.exists(ct):
                  ct = dicom.read_file(ct)
            if isinstance(nm, (str, unicode)) and os.path.exists(nm):
                  nm = dicom.read_file(nm)
            if not isinstance(ct, dicom.dataset.FileDataset):
                  Exception('CT dataset Incorrect data type')
            if not isinstance(nm, dicom.dataset.FileDataset):
                  Exception('Nuclear Medicine dataset Incorrect data type')

            #set up variables
```

```
                self.ctDCM = ct
                self.nmDCM = nm
                self.slice100Model = None
                self.slice010Model = None
                self.slice001Model = None
                self.CTaxis100TextureIDs = None
                self.CTaxis010TextureIDs = None
                self.CTaxis001TextureIDs = None
                self.isosurfaceModel = None
                self.currentAxis100TextureIDs = None
                self.currentAxis010TextureIDs = None
                self.currentAxis001TextureIDs = None
                self.CTaxis100TextureIDs = None
                self.CTaxis010TextureIDs = None
                self.CTaxis001TextureIDs = None
                self.LabeledVolumeaxis100TextureIDs = None
                self.LabeledVolumeaxis010TextureIDs = None
                self.LabeledVolumeaxis001TextureIDs = None
                self.alignmentAxis = None

                if 'joint' in kwargs and isinstance(kwargs['joint'], Joint):
                        self.joint = kwargs['joint']
                        self.isosurfaceJoint = Joint(parentJoint=self.joint,
name='Isosurface')
                        self.slice100Joint = Joint(parentJoint=self.joint, name='Slice
100 Joint')
                        self.slice010Joint = Joint(parentJoint=self.joint, name='Slice
010 Joint')
                        self.slice001Joint = Joint(parentJoint=self.joint, name='Slice
001 Joint')
                else:
                        self.joint = None

                self.isolevel = 350.0
                if 'isolevel' in kwargs:
                        self.isolevel = kwargs['isolevel']
                else:
                        dialogIsolevel, ok = QtGui.QInputDialog.getDouble(None, 'Choose
Isolevel','Enter the Hounsfield value to use as the isolevel to create the skeleton
isosurface from the CT data', maxValue=1000.0,minValue=-1000.0,value=self.isolevel)
                        if ok:
                                self.isolevel = dialogIsolevel

                #Get uCT field in hounsfield units
                self.ctField = numpy.array(self.ctDCM.pixel_array, dtype=numpy.float)
                self.ctField = self.ctField * self.ctDCM.RescaleSlope +
self.ctDCM.RescaleIntercept
                #TODO: read in pixel res
                try:
                        self.sliceThickness = self.ctDCM.SliceThickness
                except:
                        raise Exception("Can't find slice thickness in CT dicom")

                #get nm field
                self.nmField = numpy.array(self.nmDCM.pixel_array, dtype=numpy.float)
```

```
            #Create field to do computations on
            self.ctFieldPreprocessed = self.ctField.copy()

            if 'sigma' in kwargs:
                  self.sigma = kwargs['sigma']
            else:
                  self.sigma, ok = QtGui.QInputDialog.getInt(None, 'Gaussian Filter
Sigma','Enter Sigma for Gaussian Filter, 0 means no filter',
maxValue=10,minValue=0,value=2)
                  if not ok:
                        self.sigma = 0
            if self.sigma != 0:
                  self.ctFieldPreprocessed =
gaussian_filter(self.ctFieldPreprocessed, self.sigma)

            if 'resampleFactor' in kwargs:
                  self.resampleFactor = kwargs['resampleFactor']
            else:
                  self.resampleFactor, ok = QtGui.QInputDialog.getInt(None,
'Resample',"Enter Resample factor. 1 means don't resample, 2 means 1/2 points, 3
means 1/3 points,...", value=3, minValue=1, maxValue=min(self.ctField.shape)/10)
                  if not ok:
                        self.resampleFactor=1
            if self.resampleFactor>1:
                  #resample at a fraction of the original size to make future
processing much faster
#                   coord =
numpy.mgrid[0:len(self.ctFieldPreprocessed):self.resampleFactor,0:len(self.ctFieldPre
processed[0]):self.resampleFactor,0:len(self.ctFieldPreprocessed[0,0]):self.resampleF
actor]
                  coord =
numpy.mgrid[0:self.ctFieldPreprocessed.shape[0]:self.resampleFactor,

      0:self.ctFieldPreprocessed.shape[1]:self.resampleFactor,

      0:self.ctFieldPreprocessed.shape[2]:self.resampleFactor]
                  self.ctFieldPreprocessed =
map_coordinates(self.ctFieldPreprocessed, coord)

#           #setup slice view
#           if self.joint is not None:
#                 t1 = time.time()
#                 self.CTaxis100TextureIDs, self.CTaxis010TextureIDs,
self.CTaxis001TextureIDs = self.createTextures(self.ctField)
#                 self.setTextureToCT()
#                 self.createSlicesView()
#                 print 'Took %f seconds to create slice models & textures' %
(time.time()-t1)

            #set up meshs that represent forelimbs and the rest of the body
            #Figure out labels
            labels = []
            ctFieldIso = numpy.zeros(self.ctFieldPreprocessed.shape,
dtype=numpy.int)
```

```
            ctFieldIso[self.ctFieldPreprocessed >= self.isolevel] = 1
            volLabeled, numLabels = label(ctFieldIso)
            for i in xrange(numLabels):
                  size = numpy.sum(volLabeled==i)
                  labels.append((i, size))
#                 print 'Label %d Size %d' % (i, size)
            labels = numpy.array(labels, dtype = [('label',int), ('size', int)])
            labels=numpy.sort(labels, order='size')
#           volDisp = numpy.zeros(volLabeled.shape)
#           print labels
#           print 'Largest Labels', labels[-2][0], labels[-3][0], labels[-4][0]
#           volDisp[volLabeled == (labels[-2][0])]=1.0
#           volDisp[volLabeled == (labels[-3][0])]=2.0
#           volDisp[volLabeled == (labels[-4][0])]=3.0

            self.ctVolume3 = self.ctFieldPreprocessed.copy()
            self.ctVolume3[numpy.logical_not(volLabeled == (labels[-4][0]))] = 0.0
            self.ctVolume2 = self.ctFieldPreprocessed.copy()
            self.ctVolume2[numpy.logical_not(volLabeled == (labels[-3][0]))] = 0.0
            self.ctVolume1 = self.ctFieldPreprocessed.copy()
            self.ctVolume1[numpy.logical_not(volLabeled == (labels[-2][0]))] = 0.0

            if 'subsection3Shortcut' in kwargs:

      self.isosurfaceModel3=readSTLfile(kwargs['subsection3Shortcut'],joint=self.iso
surfaceJoint)
                  self.isosurfaceModel3.name = 'Isosurface 3'
            else:
                  pd =  QtGui.QProgressDialog("Polygonising SubSection file",
"Cancel", 0, 100)
                  pd.setCancelButton(None)
                  self.isosurfaceModel3 = Polygonise(self.ctVolume3,
350.0).isosurface(progressDialogCallback=pd.setValue,joint=self.isosurfaceJoint)
                  self.isosurfaceModel3.invertNormals()
                  self.isosurfaceModel3.name = 'Isosurface 3'
                  pd.setValue(100)
#                 saveSTLFile(self.isosurfaceModel3, 'Isosurface3.stl',
binary=False)

            if 'subsection2Shortcut' in kwargs:

      self.isosurfaceModel2=readSTLfile(kwargs['subsection2Shortcut'],joint=self.iso
surfaceJoint)
                  self.isosurfaceModel2.name = 'Isosurface 2'
            else:
                  pd =  QtGui.QProgressDialog("Polygonising SubSection file",
"Cancel", 0, 100)
                  pd.setCancelButton(None)
                  self.isosurfaceModel2 = Polygonise(self.ctVolume2,
350.0).isosurface(progressDialogCallback=pd.setValue,joint=self.isosurfaceJoint)
                  self.isosurfaceModel2.invertNormals()
                  self.isosurfaceModel2.name = 'Isosurface 2'
                  pd.setValue(100)
#                 saveSTLFile(self.isosurfaceModel2, 'Isosurface2.stl',
binary=False)
```

```
            if 'subsection1Shortcut' in kwargs:

      self.isosurfaceModel1=readSTLfile(kwargs['subsection1Shortcut'],joint=self.iso
surfaceJoint)
                  self.isosurfaceModel1.name = 'Isosurface 1'
            else:
                  pd =  QtGui.QProgressDialog("Polygonising SubSection file",
"Cancel", 0, 100)
                  pd.setCancelButton(None)
                  self.isosurfaceModel1 = Polygonise(self.ctVolume1,
350.0).isosurface(progressDialogCallback=pd.setValue,joint=self.isosurfaceJoint)
                  self.isosurfaceModel1.invertNormals()
                  self.isosurfaceModel1.name = 'Isosurface 1'
                  pd.setValue(100)
#                   saveSTLFile(self.isosurfaceModel1, 'Isosurface1.stl',
binary=False)

            #setup isosurface
            if self.joint is not None:
                  if 'isosurfaceModelShortcut' in kwargs:

      self.createIsosurfaceView(stlShortcutFilepath=kwargs['isosurfaceModelShortcut'
])
                  else:
                        self.createIsosurfaceView()
#                         v = []
#                         v = numpy.append(v,
self.isosurfaceModel1.OriginalVertexList.flatten())
#                         v = numpy.append(v,
self.isosurfaceModel2.OriginalVertexList.flatten())
#                         v = numpy.append(v,
self.isosurfaceModel3.OriginalVertexList.flatten())
#                         v = v.reshape((-1,3))
#                         tri = numpy.arange(v.shape[0])
#                         tri = tri.reshape((-1,3))
#                         self.isosurfaceModel = TriModel.TriModel(v, tri,
name='Whole Body IsoSurface', joint=self.isosurfaceJoint)

            self.createCTMask()

            self.bonesVol1 = ['Skull Outside','Skull Inside','Pelvis Right','Upper
Hindlimb Right','Lower Hindlimb Right','HindPaw Right','Pelvis Left','Upper Hindlimb
Left','Lower Hindlimb Left','HindPaw Left']

      def setTextureToCT(self):
                  self.currentAxis100TextureIDs = self.CTaxis100TextureIDs
                  self.currentAxis010TextureIDs = self.CTaxis010TextureIDs
                  self.currentAxis001TextureIDs = self.CTaxis001TextureIDs

      def setTextureToLabeledVolume(self):
                  self.currentAxis100TextureIDs =
self.LabeledVolumeaxis100TextureIDs
                  self.currentAxis010TextureIDs =
self.LabeledVolumeaxis010TextureIDs
```

```
                  self.currentAxis001TextureIDs =
self.LabeledVolumeaxis001TextureIDs

      def createLabelVolumeTextures(self, labeledVolume):
            self.LabeledVolumeaxis100TextureIDs,
self.LabeledVolumeaxis010TextureIDs, self.LabeledVolumeaxis001TextureIDs =
self.createTextures(labeledVolume)

      def createSlicesView(self):
            #choose initial index
            self.sliceIndex = numpy.array([100,90,90],dtype=numpy.int)

            #setup axis [1,0,0]
            vertexList = [      [0,                                     0,
                        0],
                                    [0,
      self.ctField.shape[1],    0],
                                    [0,
      self.ctField.shape[1],    self.ctField.shape[2]],
                                    [0,                                     0,
                                    self.ctField.shape[2]]]
            sqrList = [[0,1,2,3]]
            self.slice100Model = TriModel.TriModel(numpy.array(vertexList),
numpy.array(sqrList), self.slice100Joint, name='Slice
[1,0,0]',textureID=self.CTaxis100TextureIDs[self.sliceIndex[0]])

            #setup axis [0,1,0]
            vertexList = [      [0,                                     0,
                        0],
                                    [self.ctField.shape[0],   0,
                  0],
                                    [self.ctField.shape[0],   0,
                  self.ctField.shape[2]],
                                    [0,                                     0,
                                    self.ctField.shape[2]]]
            sqrList = [[0,1,2,3]]
            self.slice010Model = TriModel.TriModel(numpy.array(vertexList),
numpy.array(sqrList), self.slice010Joint, name='Slice
[0,1,0]',textureID=self.CTaxis010TextureIDs[self.sliceIndex[1]])

            #setup axis [0,0,1]
            vertexList = [      [0,                                     0,
                        0],
                                    [self.ctField.shape[0],   0,
                  0],
                                    [self.ctField.shape[0],
      self.ctField.shape[1],    0],
                                    [0,
      self.ctField.shape[1],    0]]
            sqrList = [[0,1,2,3]]
            self.slice001Model = TriModel.TriModel(numpy.array(vertexList),
numpy.array(sqrList), self.slice001Joint, name='Slice
[0,0,1]',textureID=self.CTaxis001TextureIDs[self.sliceIndex[2]])

      def setSlice100Index(self, index):
```

```
            if self.slice100Model is not None and index >= 0 and index <
self.ctField.shape[0]:    #check for proper index
                  self.slice100Model.textureID =
self.currentAxis100TextureIDs[index]
                  self.slice100Joint.translate([index,0,0], absolute=True)

      def setSlice010Index(self, index):
            if self.slice010Model is not None and index >= 0 and index <
self.ctField.shape[1]:    #check for proper index
                  self.slice010Model.textureID =
self.currentAxis010TextureIDs[index]
                  self.slice010Joint.translate([0,index,0], absolute=True)

      def setSlice001Index(self, index):
            if self.slice001Model is not None and index >= 0 and index <
self.ctField.shape[2]:    #check for proper index
                  self.slice001Model.textureID =
self.currentAxis001TextureIDs[index]
                  self.slice001Joint.translate([0,0,index], absolute=True)

      def createTextures(self, volume, colormap=None):
            refVol = numpy.array(volume)
            if refVol.ndim == 4:
                  volRGBA = numpy.array(refVol, dtype=numpy.ubyte)
                  if refVol.shape[3] == 4:
                        pixelFormat = GL.GL_RGBA
                  elif refVol.shape[3] == 3:
                        pixelFormat = GL.GL_RGB
                  else:
                        return None, None, None
            elif refVol.ndim == 3:
                  #normalize to range 0-1
                  refVol -= refVol.min()
                  refVol /= refVol.max()
                  refVol *= 255
                  refVol = numpy.array(refVol, dtype=numpy.ubyte)
                  if colormap is None:      #grayscale
                        volRGBA = refVol
                        pixelFormat = GL.GL_LUMINANCE
                  else:  #assign colormap to grayscale values
                        colormap = numpy.array(colormap)
                        if colormap.shape[1] == 3:
                              pixelFormat = GL.GL_RGB
                        elif colormap.shape[1] == 4:
                              pixelFormat = GL.GL_RGBA
                        else:
                              return None, None, None
                        volRGBA = numpy.empty( (refVol.shape[0], refVol.shape[1],
refVol.shape[2], colormap.shape[1]) , dtype=numpy.ubyte)
                        for i in xrange(colormap.shape[0]):
                              volRGBA[refVol==i] = colormap[i]
            else:
                  return None, None, None

            #create openGL texture ID array
```

```
            axis100TextureIDs = GL.glGenTextures(volRGBA.shape[0])
            for i in xrange(volRGBA.shape[0]):
                  image = volRGBA[i,:,:]
                  data = image.tostring()
                  height = image.shape[0]
                  width = image.shape[1]
                  GL.glBindTexture( GL.GL_TEXTURE_2D, axis100TextureIDs[i] )
                  GL.glTexEnvf( GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE,
GL.GL_MODULATE )
                  GL.glTexParameterf( GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER,
GL.GL_LINEAR )
                  GL.glTexParameterf( GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER,
GL.GL_LINEAR )
                  GL.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGBA, width, height,
0, pixelFormat, GL.GL_UNSIGNED_BYTE, data)

            axis010TextureIDs = GL.glGenTextures(volRGBA.shape[1])
            for i in xrange(volRGBA.shape[1]):
                  image = volRGBA[:,i,:]
                  data = image.tostring()
                  height = image.shape[0]
                  width = image.shape[1]
                  GL.glBindTexture( GL.GL_TEXTURE_2D, axis010TextureIDs[i] )
                  GL.glTexEnvf( GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE,
GL.GL_MODULATE )
                  GL.glTexParameterf( GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER,
GL.GL_LINEAR )
                  GL.glTexParameterf( GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER,
GL.GL_LINEAR )
                  GL.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGBA, width, height,
0, pixelFormat, GL.GL_UNSIGNED_BYTE, data)

            axis001TextureIDs = GL.glGenTextures(volRGBA.shape[2])
            for i in xrange(volRGBA.shape[2]):
                  image = volRGBA[:,:,i]
                  data = image.tostring()
                  height = image.shape[0]
                  width = image.shape[1]
                  GL.glBindTexture( GL.GL_TEXTURE_2D, axis001TextureIDs[i] )
                  GL.glTexEnvf( GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE,
GL.GL_MODULATE )
                  GL.glTexParameterf( GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER,
GL.GL_LINEAR )
                  GL.glTexParameterf( GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER,
GL.GL_LINEAR )
                  GL.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGBA, width, height,
0, pixelFormat, GL.GL_UNSIGNED_BYTE, data)

            return axis100TextureIDs, axis010TextureIDs, axis001TextureIDs

      def createIsosurfaceView(self, stlShortcutFilepath=''):
            if isinstance(stlShortcutFilepath, str) and
os.path.exists(stlShortcutFilepath):
                  self.isosurfaceModel =
readSTLfile(stlShortcutFilepath,True,joint=self.isosurfaceJoint)
```

```
            else:
                  pd =  QtGui.QProgressDialog("Polygonising DICOM file", "Cancel",
0, 100)
                  pd.setCancelButton(None)
                  self.isosurfaceModel = Polygonise(self.ctFieldPreprocessed,
self.isolevel).isosurface(progressDialogCallback=pd.setValue,joint=self.isosurfaceJoi
nt)
                  self.isosurfaceModel.invertNormals()
                  pd.setValue(100)
            self.isosurfaceModel.name = 'Isosurface from CT'
            self.isosurfaceJoint.scale(self.resampleFactor)

      def createCTMask(self):
            #assume centers of CT and SPECT images are aligned
            ctdim =
numpy.array(self.ctVolume1.shape)*self.resampleFactor*self.ctDCM.SliceThickness
      #ct volume dimension in mm
            spectdim = numpy.array(self.nmField.shape)*self.nmDCM.SliceThickness
      #nm dimension in mm

            minval = (ctdim-spectdim)/2.0
            maxval = ctdim - (ctdim-spectdim)/2.0

            #get slicing coordinates in mm, subtracting spacing() makes it exclusive
on the stop value
            coord = numpy.mgrid[minval[0]:maxval[0]-
numpy.spacing(maxval[0]):self.nmDCM.SliceThickness,
                                          minval[1]:maxval[1]-
numpy.spacing(maxval[0]):self.nmDCM.SliceThickness,
                                          minval[2]:maxval[2]-
numpy.spacing(maxval[0]):self.nmDCM.SliceThickness]

            #convert mm slicing coordinates to ct slice units
            coord /= self.resampleFactor*self.ctDCM.SliceThickness

            ctVolume1NMSize = map_coordinates(self.ctVolume1, coord)
            print ctVolume1NMSize.shape
            if ctVolume1NMSize.shape != self.nmField.shape:
                  raise Exception('created CT mask volume is not the same size as
NM volume')
            self.ctVolume1NMmask =
numpy.zeros(ctVolume1NMSize.shape,dtype=numpy.int)
            self.ctVolume1NMmask[ctVolume1NMSize>=self.isolevel] = 1
            self.ctVolumeNMmaskOriginLocation = minval
            print 'CT to NM offset is', self.ctVolumeNMmaskOriginLocation

      def createNM_Mask(self, boneVertexLists):
            '''
            boneVertexLists are lists of transformed vertices, each list contains
all the vertices from a single bone group
            '''
            dataPoints = []
            boneIndx = []
            for boneVertexList in boneVertexLists:
                  dataPoints = numpy.append(dataPoints, boneVertexList)
```

```
                        if len(boneIndx) == 0:
                                boneIndx.append(len(boneVertexList))
                        else:
                                boneIndx.append(len(boneVertexList)+boneIndx[-1])
                dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
                bonesKDTree = KDTree(dataPoints)
                scanBonePoints = numpy.array(numpy.where(self.ctVolume1NMmask == 1)).T
                d, NN = bonesKDTree.query(scanBonePoints)

                mask = numpy.zeros(self.ctVolume1NMmask.shape, dtype=numpy.int)
                for i in xrange(NN.shape[0]):
                        for j in xrange(len(boneIndx)):
                                if NN[i] < boneIndx[j]:   #this nearest neighbor is in
bone j, mark it as such in the label mask
                                        mask[scanBonePoints[i,0], scanBonePoints[i,1],
scanBonePoints[i,2]] = j+1
                                        break

                return mask



if __name__ == '__main__':
        UserDicom('..\dicom\Myeloma MDP Scans\Trial\JVO_MyelomaMDPTrial_DM533-1-
10_CT.dcm', '..\dicom\Myeloma MDP Scans\Trial\JVO_MyelomaMDPTrial_DM533-1-
10_SPECT.dcm')
```

AlignAtlas.py

```
'''
Created on Jan 4, 2012

@author: Jeff
'''
import numpy, time
from math import pi
from scipy.ndimage.interpolation import affine_transform
from scipy.ndimage.measurements import center_of_mass
from scipy.ndimage import gaussian_filter
import scipy.ndimage
from TriModel import TriModel
from scipy.io import loadmat, savemat
from scipy.ndimage.measurements import label
import scipy.signal
import numpyTransform
from ICP import ICP
import matplotlib.pyplot as plt
from scipy.spatial import KDTree
from polygonise import Polygonise
import STL
import os.path
from numpy.ma.core import absolute

def getLargestVolumeLabel(binaryVol):
	labeledVol, num_features = label(binaryVol)
	volumeSize = numpy.zeros(num_features+1)
	for i in xrange(1,num_features+1):
		volumeSize[i] = numpy.sum(binaryVol[labeledVol==i])
	maxVolume = volumeSize[1:].max()
	maxVolumeLabel = numpy.nonzero(volumeSize==maxVolume)
	labeledVol[labeledVol!=maxVolumeLabel] = 0
	labeledVol[labeledVol==maxVolumeLabel] = 1
	return labeledVol

def filterSpineData(x,data,headTop):
	dataRaw = data.copy()
	dataDiv1 = numpy.zeros(data.shape)
	dataDiv2 = numpy.zeros(data.shape)
	window = numpy.ones(data.shape[0]/10)
	window /= len(window)
	#smooth the signal a lot
	dataSmoothed = scipy.signal.convolve(data, window, mode='same')
	dataSmoothed = scipy.signal.convolve(dataSmoothed, window, mode='same')
	dataSmoothed = scipy.signal.convolve(dataSmoothed, window, mode='same')
	dataSmoothed = scipy.signal.convolve(dataSmoothed, window, mode='same')

	dataDiv1[:-1] = dataSmoothed[1:] - dataSmoothed[:-1]
	dataDiv1[-1] = dataDiv1[-2]
	dataDiv2[:-1] = dataDiv1[1:] - dataDiv1[:-1]
	dataDiv2[-1] = dataDiv2[-2]

	#cutting 1/7 of front and back of mouse will remove nose and tail
```

```
	#this removes some of the noise that occurs at the nose
	startSlice = data.shape[0]/7
	endSlice = data.shape[0]-startSlice

	dataSmoothed = dataSmoothed[startSlice:endSlice]
	dataDiv1 = dataDiv1[startSlice:endSlice]
	dataDiv2 = dataDiv2[startSlice:endSlice]
	x = x[startSlice:endSlice]
	dataRaw = dataRaw[startSlice:endSlice]

	#look for the first zero crossing of the second derivative, on an increasing
slope
	#this point should be the inflection point between the skull and the spine,
this is pretty close to the neck position
	if headTop:
		for i in xrange(len(dataDiv2),0,-1):
			if dataDiv2[i-1] > 0 and dataDiv2[i] <= 0:
				break
	else:
		for i in xrange(1,len(dataDiv2)):
			if dataDiv2[i-1] < 0 and dataDiv2[i] >= 0:
				break
	neckPos = x[i]
	return dataRaw, dataSmoothed, dataDiv1, dataDiv2, neckPos, x

def getSpine(filteredVol, neckLoc, headTop, **kwargs):
	'''
	This function assumes that the posterior vector is [1,0,0], ie headTop==False

	function returns volume of spine data as well as hip joint location
	'''
	neckLoc = numpy.array(neckLoc).squeeze()
	hipLoc = None
	if headTop:  #head is in high indices, flip array and neckLoc
		filteredVol = filteredVol[::-1,:,:]
		neckLoc[0] = filteredVol.shape[0]-neckLoc[0]
	neckLocInt = neckLoc.round().astype(numpy.int)
	filteredVol[neckLocInt[0], neckLocInt[1], neckLocInt[2]] = 3.0
	spineVol = numpy.zeros(filteredVol.shape)
	spineVolFinal = numpy.zeros(spineVol.shape)
	tailVolFinal = numpy.zeros(spineVol.shape)
	numBlobDetachFromSpine = 0
	taillabel = None

	for taSliceIndx in xrange(neckLocInt[0],filteredVol.shape[0]):
		taSlice = filteredVol[taSliceIndx]
		taSlice = scipy.ndimage.binary_fill_holes(taSlice)
		taSlicei, numFeaturesi = label(taSlice)
		taSliceLabeled = taSlice.copy()
		taSlicej= spineVol[taSliceIndx-1]	#previous slice
		taSlice = numpy.zeros(taSlice.shape)	#taSlice will be rebuild using
labels
		if 1 not in numpy.unique(taSlicej) and taSliceIndx != neckLocInt[0] and
hipLoc is None:
			print 'Spine label is gone!'
```

```
            #first time through find blobs, if more that one, find which blob center
of mass is closest to neck location, this is the spine blob, ignore other blobs
            if taSliceIndx == neckLocInt[0]:
                  if numFeaturesi > 1:
                        CoMs = numpy.array(center_of_mass(taSliceLabeled,
taSlicei, numpy.unique(taSlicei)[1:]))
                        if CoMs.ndim == 1:
                              raise Exception('There are suppose to be multiple
labels but yet there is only one center of mass? something is wrong')
                        CoMDist = []
                        for CoM in CoMs:
                              CoMDist.append( numpy.sqrt(numpy.sum((CoM-
neckLoc[1:])**2)) )
                        CoMDist = numpy.array(CoMDist)
                        spineLabel = numpy.where(CoMDist == CoMDist.min())[0][0]+1
      #label of closest blob
                        taSlice[taSlicei==spineLabel] = 1
                        spineVol[taSliceIndx] = taSlice
                  else:  #one blob in first slice, must be good
                        spineVol[taSliceIndx] = taSlicei
                  continue     #move on to next slice

            for labeli in numpy.unique(taSlicei):  #look at each label in this
slice to see if it corresponds to a label in the previous slice
                  if labeli == 0:
                        continue
                  maski = taSlicei==labeli
                  for labelj in numpy.unique(taSlicej):  #look at each label in
previous slice
                        if labelj == 0:
                              continue
                        maskj = taSlicej==labelj
                        if numpy.any(numpy.logical_and(maski, maskj)):      #if
masks have something in common then they are part of the same thing
                              taSlice[maski] = labelj
                              break

            if hipLoc is None and 1 not in numpy.unique(taSlice):      #we haven't
found the hip yet and for some reason there isn't a 1 in our slice, make nearest
label to previous slice spine label, the spine label for this slice
                  CoMj = numpy.array(center_of_mass(taSlicej, taSlicej, 1))
      #previous slice spine location
                  CoMs = numpy.array(center_of_mass(taSlicei, taSlicei,
numpy.unique(taSlicei)[1:]))
                  CoMDist = []
                  for CoM in CoMs:
                        CoMDist.append( numpy.sqrt(numpy.sum((CoM-CoMj)**2)) )
                  CoMDist = numpy.array(CoMDist)
                  spineLabel = numpy.where(CoMDist == CoMDist.min())[0][0]+1 #label
of closest blob
                  taSlice[taSlicei==spineLabel] = 1

            for labeli in numpy.unique(taSlicei):  #look for labels which are new
and weren't in previous slice
                  if labeli == 0:
```

```
                        continue
                    for labelNum in xrange(taSlice.size):  #look for next available
label number
                        if labelNum not in taSlice:
                            break
                    maski = taSlicei==labeli
                    if numpy.all(taSlice[maski]==0): #no overlap must be unique label
                        taSlice[maski] = labelNum

            #check to see if a label merged into the spine label
            #spine label will always be 1
            #hip can't be closer than 15 slices from neck position
            if hipLoc is None and taSliceIndx > neckLocInt[0]+25:
                spineMask = taSlice==1
                for labelj in numpy.unique(taSlicej):  #look at each label in
previous slice
                    if labelj == 0 or labelj == 1:
                        continue
                    maskj = taSlicej==labelj
                    if numpy.any(numpy.logical_and(spineMask, maskj)):  #if
masks have something in common then they are part of the same thing, blob merge has
been found
                        CoM = center_of_mass(taSlicej, taSlicej, [1])
                        CoM = numpy.array(CoM).squeeze()
                        hipLoc = numpy.array([taSliceIndx, CoM[0], CoM[1]])
                        spineVolFinal[spineVol == 1] = 1
                        print 'Hip joint start location', hipLoc
                        break

            if hipLoc is not None: #search for end of hipjoint
                taSliceii = numpy.zeros(taSlice.shape)
                taSliceii[taSlice==1] = 1
                labelii, numFeaturesii = label(taSliceii)
                if numBlobDetachFromSpine == 0 and numFeaturesii > 2:      #spine
has split into at least 3 things, this means both hips are separated in the space of
one slice, measure distances from CoM of each blob to the hipLoc, closest blob CoM is
the spine/tail blob
                    numBlobDetachFromSpine += numFeaturesii-1
                    #Get distance between each blob of
                    CoMs = numpy.array(center_of_mass(labelii, labelii,
numpy.unique(labelii)[1:]))
                    if CoMs.ndim == 1:
                        raise Exception('There are suppose to be multiple
labels but yet there is only one center of mass? something is wrong')
                    CoMDist = []
                    for CoM in CoMs:
                        CoMDist.append( numpy.sqrt(numpy.sum((CoM-
hipLoc[1:])**2)) )
                    CoMDist = numpy.array(CoMDist)
                    spineLabel = numpy.where(CoMDist == CoMDist.min())[0][0]+1
      #label of closest blob to hiploc
                    for lab in numpy.unique(labelii):
                        if lab== 0 or lab == spineLabel:
                            continue
```

```
                                    for labelnum in xrange(taSlice.size):  #look for
next available label number
                                          if labelnum not in taSlice:
                                                break
                                    taSlice[labelii==lab] = labelnum

                        elif numBlobDetachFromSpine == 0 and numFeaturesii > 1:    #only
one hip has detached from the spine, use the larger of the two blobs as the
spine\tail blob
                              numBlobDetachFromSpine += numFeaturesii-1
                              labSize=[]
                              for lab in numpy.unique(labelii):
                                    if lab== 0:
                                          continue
                                    labSize.append(numpy.sum(labelii==lab))
                              labSize = numpy.array(labSize)
                              spineLabelIndex =
numpy.where(labSize==labSize.max())[0][0]+1   #index in unique() that gets label with
largest area
                              spineLabel = numpy.unique(labelii)[spineLabelIndex]
                              for lab in numpy.unique(labelii):
                                    if lab== 0 or lab == spineLabel:
                                          continue
                                    for labelnum in xrange(taSlice.size):  #look for
next available label number
                                          if labelnum not in taSlice:
                                                break
                                    taSlice[labelii==lab] = labelnum

                        elif numBlobDetachFromSpine == 1 and numFeaturesii > 1:
            #second hip has detached from the spine, measure the distance from CoM of
blobs to hiploc, uses closest blob CoM as spine/tail blob
                              numBlobDetachFromSpine += numFeaturesii-1
                              #Get distance between each blob of
                              CoMs = numpy.array(center_of_mass(labelii, labelii,
numpy.unique(labelii)[1:]))
                              if CoMs.ndim == 1:
                                    raise Exception('There are suppose to be multiple
labels but yet there is only one center of mass? something is wrong')
                              CoMDist = []
                              for CoM in CoMs:
                                    CoMDist.append( numpy.sqrt(numpy.sum((CoM-
hipLoc[1:])**2)) )
                              CoMDist = numpy.array(CoMDist)
                              spineLabel = numpy.where(CoMDist == CoMDist.min())[0][0]+1
            #label of closest blob to hiploc
                              for lab in numpy.unique(labelii):
                                    if lab== 0 or lab == spineLabel:
                                          continue
                                    for labelnum in xrange(taSlice.size):  #look for
next available label number
                                          if labelnum not in taSlice:
                                                break
                                    taSlice[labelii==lab] = labelnum
```

```
                    if numBlobDetachFromSpine >= 2 and taillabel is None:      #both
hips have detached from the spine, mark this point as end of spine joint, modify
hiploc,
                            #modify hipLoc position to be center of hip joint
                            CoM = center_of_mass(taSlice, taSlice, [1])
                            CoM = numpy.array(CoM).squeeze()
                            print 'Hip joint end location', numpy.array([taSliceIndx,
CoM[0], CoM[1]])
                            hipLoc = (hipLoc + numpy.array([taSliceIndx, CoM[0],
CoM[1]])) / 2.0     #get mean between start and end location

                            #overwrite spine label with tail
                            taillabel = taSlice.size+1
                            taSlice[taSlice==1] = taillabel

                spineVol[taSliceIndx] = taSlice

        if hipLoc is None:
                raise Exception('Could not find hip')

        #create tail volume
        tailVolFinal[spineVol==taillabel] = 1

        if headTop:
                print 'accounting for aligned volume being flipped'
                hipLoc[0] = filteredVol.shape[0]-hipLoc[0]    #flip first dimension of
hipLoc to compensate for original flipping of filteredVol
                spineVolFinal = spineVolFinal[::-1,:,:]#flip first dimension of
spineVolFinal to compensate for original flipping of filteredVol
                tailVolFinal = tailVolFinal[::-1,:,:]  #flip first dimension of
tailVolFinal to compensate for original flipping of filteredVol
        print 'Hip location:', hipLoc

        if 'spineModelShortcut' in kwargs and
os.path.exists(kwargs['spineModelShortcut']):
                spineModel = STL.readSTLfile(kwargs['spineModelShortcut'],
verbose=False)
        else:
                #create mesh of spine
                spineModel = Polygonise(spineVolFinal, 0.5).isosurface()
                if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                        STL.saveSTLFile(spineModel,
kwargs['savePrefix']+'SpineModel.stl', binary=False)

        if 'tailModelShortcut' in kwargs and
os.path.exists(kwargs['tailModelShortcut']):
                tailModel = STL.readSTLfile(kwargs['tailModelShortcut'], verbose=False)
        else:
                #create mesh of tail
                tailModel = Polygonise(tailVolFinal, 0.5).isosurface()
                if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                        STL.saveSTLFile(tailModel, kwargs['savePrefix']+'TailModel.stl',
binary=False)

        if 'visualize' in kwargs and kwargs['visualize']:
```

```
            import visvis as vv
            for taillabelDisp in xrange(taSlice.size):    #look for next available
label number
                  if taillabelDisp not in spineVol:
                        break
            spineVol[spineVol==taillabel] = taillabelDisp
            app = vv.use()
            vv.figure()
            vv.volshow2(spineVol)
            vv.title('Spine segmentation')
            vv.xlabel('[0,0,1] axis')
            vv.ylabel('[0,1,0] axis')
            vv.zlabel('[1,0,0] axis')
            app.Run()

      return hipLoc, spineModel, tailModel

def roughAlign(scanData, atlasData, **kwargs):
      '''
      roughtAlign() takes the CT volume, CTmesh, and isolevel and returns the
      rotation matrix, scale factor, and neck joint location

      This rough align function is only designed to work for mice, it might work
      for other rodents, but almost certainly would not work for people.

      Rough Align Steps:
      1.    PCA analysis of Mesh points to figure out the axes
                  This should results in the axes being defined in a certain order
                  [anteroposterior, dorsoventral, left-right]

      2.    Align CTVolume to the standard [x, y, z] cartesian coordinate system
                  This can be done by using the PCA axis as a rotation matrix from
                  step 1. The resulting alignedCTVolume dimensions are
                  [transverse, coronal, sagittal]. For example
alignedCTVolume[:,:,1]
                  would be a sagittal plane slice.

      3.    Determine the vector that points to the anterior end
                  Since we know that the anteroposterior axis is along the [1,0,0]
                  line, the vector, as taken from the center point on the
                  anteroposterior axis of the alignedCTVolume, that points to the
                  anterior end must be [-1,0,0] or [1,0,0]. Measure the amount of
bone
                  mass in each half of the alignedCTVolume. Halves are created by
                  splitting the alignedCTVolume by the [N/2,:,:] plane, where N is
                  length of the alignedCTVolume in the first dimension. Bone mass
is
                  determined as the number of pixels above the isolevel threshold.
The
                  half with the most bone mass will be the half with the skull. So
if
                  the alignedCTVolume[0:N/2] half has more bone mass than the
                  alignedCTVolume[N/2:N] half, the vector pointing in the direction
of
                  the anterior end would be [-1,0,0]
```

```
	4.	Determine the vector that points to the dorsal side
		We know that the dorsoventral axis is the [0,1,0] axis of the
		alignedCTVolume. So the vector, as taken from the center point on
		the dorsoventral axis of the alignedCTVolume, must be either
[0,1,0]
		or [0,-1,0]. The center of mass (CoM) can can be calculated for
each
		binary transverse slice. normal transverse slices are transformed
		into binary by thresholding on the isolevel. The line that
connects
		these CoM points closely follows the spine along the midsection
of
		the animal. The spine curvature in the midsection of a mouse is
		typically a high arch with its apex pointing towards the dorsal
		side. This high arch can be detected because on average the CoM
line
		on the arc side is further away from the median in the second
		dimension that the CoM values on the other side of the median
line.
		First flatten the 3D curve onto the [:,:,0] plane. Determine the
		mean coordinate, curveMean. Define the second dimension of
curveMean
		value B, i.e. B=curveMean[1]. Determine mean point of all points
		whose second dimension value is greater than curveMean[1], call
this
		value A. Determine mean point of all points whose second
dimension
		value is less than curveMean[1], call this value C. If A > C than
		the vector that points to the dorsal end is [0,1,0]

12345678911234567892123456789312345678941234567895123456789612345678971234567898
	5.	Determine Neck Location
		asdf

12345678911234567892123456789312345678941234567895123456789612345678971234567898
	6. Determine Scale factor
		asdf
	'''
	#preprocessing
	if 'verbose' in kwargs:
		verbose = kwargs['verbose']
	else:
		verbose=False
	if 'visualize' in kwargs:
		visualize = kwargs['visualize']
	else:
		visualize=False
	if 'pcaAxis' in kwargs:
		pcaAxis = kwargs['pcaAxis']
	else:
		pcaAxis=None

	#Step 1.	PCA analysis of Mesh points to figure out the axes.
	if pcaAxis is None:
```

```
            t1 = time.time()
            pcaAxis = numpy.matrix(scanData.isosurfaceModel.PCA())
            if verbose:
                  print 'Time to perform Principle Component Analysis:',
time.time()-t1

      #Step 2.     Align CTvolume to the standard [x, y, z] cartesian coordinate
system
      rotMat = numpyTransform.coordinateSystemConversionMatrix(pcaAxis,
numpy.identity(3), N=3)
      print 'Alignment Rotation Matrix:'
      print rotMat
      alignedCTVolume, rotMat, offset, affineTrans =
alignVolume(scanData.ctFieldPreprocessed, rotMat, verbose=verbose)

      #Step 3.     Determine the vector that points to the anterior end
      #make alignedCTVolume binary based on isolevel threshold
      t1 = time.time()
      zeroIndex = alignedCTVolume<scanData.isolevel
      oneIndex = alignedCTVolume>=scanData.isolevel
      alignedCTVolume[zeroIndex]=0.0
      alignedCTVolume[oneIndex]=1.0
      if verbose:
            print 'Time to create binary alignCTVolume:', time.time()-t1

      #Determine is anterior diction based on bone mass and translate it back to PCA
axis
      t1 = time.time()
      lowerBoneMass = numpy.sum(alignedCTVolume[0:alignedCTVolume.shape[0]/2,:,:])
      upperBoneMass = numpy.sum(alignedCTVolume[alignedCTVolume.shape[0]/2:,:,:])
      if upperBoneMass > lowerBoneMass:
            anteriorVector = numpy.asarray(numpy.matrix([1.0,0.0,0.0]) *
rotMat.I).squeeze()
            headTop=True
      else:
            anteriorVector = numpy.asarray(numpy.matrix([-1.0,0.0,0.0]) *
rotMat.I).squeeze()
            headTop=False
      if verbose:
            print 'Larger of the two following numbers indicates the anterior
direction'
            print 'Lower bone mass:',lowerBoneMass
            print 'Upper bone mass:',upperBoneMass
            print 'Anterior Vector is (aligned to PCA axes):', anteriorVector
            print 'Time to calculate anterior vector:', time.time()-t1

      #Step 4.     Determine the vector that points to the dorsal side
      #Create 3D curve of center of mass points on transverse slices
      #also calculate the standard deviation of the distance from the center of mass
to all points above the isolevel in each transverse slice
      #this will be used to estimate the neck location
      t1 = time.time()
      largestVolLabel = getLargestVolumeLabel(alignedCTVolume)   #largestVolLabel
used to be alignedCTVolume, using largestVolLabel has the benefit of removing the
forelimbs because they aren't directly attached to the rest of the skeleton
```

```
	CoMList = []
	coronalData = []
	for i in xrange(largestVolLabel.shape[0]):
		transverseSlice = largestVolLabel[i,:,:]
		indicies = numpy.array(numpy.where(transverseSlice==1.0))
		if indicies.shape[1] == 0:
			continue
		CoM = center_of_mass(transverseSlice)
		if numpy.any(numpy.isnan(CoM)):
			continue
		CoMList.append([i, CoM[0], CoM[1]])
		coronalData.append(numpy.std(indicies[1]-CoM[1]))
	CoMList = numpy.array(CoMList)
	coronalData = numpy.array(coronalData)
	if verbose:
		print 'Time to calculate 3D Center of Mass Curve and Stats:',
time.time()-t1

	#Pick dorsal vector and transform it to Mesh PCA axis
	t1 = time.time()
	meanVal = CoMList.mean(axis=0)
	meanHigh = numpy.mean(CoMList[CoMList[:,1] > meanVal[1]], axis = 0)
	meanLow = numpy.mean(CoMList[CoMList[:,1] < meanVal[1]], axis = 0)
	if abs((meanHigh-meanVal)[1]) > abs((meanLow-meanVal)[1]):
		dorsalVector = numpy.asarray(numpy.matrix([0.0,1.0,0.0]) *
rotMat.I).squeeze()
	else:
		dorsalVector = numpy.asarray(numpy.matrix([0.0,-1.0,0.0]) *
rotMat.I).squeeze()
	if verbose:
		print 'Greater of following two values indicates that is the dorsal
direction'
		print 'Mean of values above mean:', abs((meanHigh-meanVal)[1])
		print 'Mean of values below mean:', abs((meanLow-meanVal)[1])
		print 'Dorsal Vector is (aligned to PCA axes):', dorsalVector
		print 'Time to calculate dorsal vector:', time.time()-t1

	#Step 5: Determine neck location
	#TODO: Document
	t1=time.time()
	dataRaw, dataSmoothed, dataDiv1, dataDiv2, neckPosAligned, x =
filterSpineData(CoMList[:,0], coronalData, headTop=headTop)
	neckPosAligned = CoMList[CoMList[:,0]==neckPosAligned][0]
	#transform neck position back to original coordinate system
	neckPosResample = numpyTransform.transformPoints(affineTrans.I,
neckPosAligned)

	#scale Neck position by the resampleFactor and slice to appropriate size
	neckPos = neckPosResample * scanData.resampleFactor
	if verbose:
		print 'Neck Position', neckPos
		print 'Time to calculate Neck Position:',time.time()-t1

	#get spine and hip information
	filteredBinVol = numpy.zeros(scanData.ctFieldPreprocessed.shape)
```

```
	filteredBinVol[scanData.ctFieldPreprocessed>=scanData.isolevel]=1.0
	hipLocAligned, spineModel, tailModel = getSpine(largestVolLabel,
neckPosAligned, headTop,**kwargs)

	vertices = numpyTransform.transformPoints(affineTrans.I,
spineModel.OriginalVertexList) * scanData.resampleFactor
	spineModel = TriModel(vertices, spineModel.TriangleVertexIndexList,
name='Spine Model', color=[ 0.83921569,0. ,  0.51764706])

	vertices = numpyTransform.transformPoints(affineTrans.I,
tailModel.OriginalVertexList) * scanData.resampleFactor
	tailModel = TriModel(vertices, tailModel.TriangleVertexIndexList, name='Tail
Model', color=[ 1,0,0])

	#transform the hip position back to non aligned coordinate system
	hipLocResample = numpyTransform.transformPoints(affineTrans.I, hipLocAligned)
	hipLoc = hipLocResample * scanData.resampleFactor
	print 'Hip location dicom', hipLoc

	#Step 6: Determine scaling factor
	#TODO: add documentation
	t1=time.time()
	atlasSize = atlasData.atlasJoint.getBoundingBox()
	dicomSize = scanData.isosurfaceJoint.getBoundingBox()
	if verbose:
		print 'Atlas Bounding Box:', atlasSize
		print 'DICOM Bounding Box:', dicomSize
	atlasSize=numpy.sqrt(numpy.sum((atlasSize[1]-atlasSize[0])**2))
	dicomSize=numpy.sqrt(numpy.sum((dicomSize[1]-dicomSize[0])**2))
	if verbose:
		print 'Atlas Size:', atlasSize
		print 'DICOM Size:', dicomSize
	scaleFactor = dicomSize/atlasSize
	if verbose:
		print 'Scale Factor:',scaleFactor
		print 'Time to calculate scale factor:',time.time()-t1

	if visualize:
		import visvis as vv #visvis axis are (z,y,x)
		app = vv.use()

		#for visualization purposes set values so CoM line is visible in volume
		for CoM in CoMList:
			if numpy.all(CoM == neckPosAligned):
				CoM = CoM.round().astype(numpy.int)
				alignedCTVolume[CoM[0], CoM[1], CoM[2]] = 3.0
			else:
				CoM = CoM.round().astype(numpy.int)
				alignedCTVolume[CoM[0], CoM[1], CoM[2]] = 2.0
		#highlight hip joint
		hipindx = hipLocAligned.round().astype(numpy.int)
		alignedCTVolume[hipindx[0], hipindx[1], hipindx[2]] = 3.0

		vv.figure()
		vv.volshow(scanData.ctFieldPreprocessed, renderStyle='mip')
```

```
            vv.ColormapEditor(vv.gca())
            vv.title('Start Volume')
            vv.xlabel('[0,0,1] axis')
            vv.ylabel('[0,1,0] axis')
            vv.zlabel('[1,0,0] axis')

            vv.figure()
            vv.volshow(alignedCTVolume, renderStyle='mip')
            vv.ColormapEditor(vv.gca())
            vv.title('Aligned Volume')
            vv.xlabel('[0,0,1] axis')
            vv.ylabel('[0,1,0] axis')
            vv.zlabel('[1,0,0] axis')

            vv.figure()
            vv.plot(CoMList[:,2],CoMList[:,1],CoMList[:,0],lc='k')
            PCAaxis1 = numpy.zeros((3,3))
            PCAaxis1[2,:] = numpy.asarray(rotMat)[0]*70
            PCAaxis1 += CoMList.mean(axis=0)
            vv.plot(PCAaxis1[:,2], PCAaxis1[:,1],PCAaxis1[:,0],lc='r')
            PCAaxis2 = numpy.zeros((3,3))
            PCAaxis2[2,:] = numpy.asarray(rotMat)[1]*70
            PCAaxis2 += CoMList.mean(axis=0)
            vv.plot(PCAaxis2[:,2], PCAaxis2[:,1],PCAaxis2[:,0],lc='g')
            PCAaxis3 = numpy.zeros((3,3))
            PCAaxis3[2,:] = numpy.asarray(rotMat)[2]*70
            PCAaxis3 += CoMList.mean(axis=0)
            vv.plot(PCAaxis3[:,2], PCAaxis3[:,1],PCAaxis3[:,0],lc='b')
            vv.plot([0,dorsalVector[2]*30], [0,dorsalVector[1]*30],
[0,dorsalVector[0]*30],lc='c')
            vv.plot([0,anteriorVector[2]*30], [0,anteriorVector[1]*30],
[0,anteriorVector[0]*30],lc='m')
            vv.title('Spine Curve')
            vv.xlabel('[0,0,1] axis')
            vv.ylabel('[0,1,0] axis')
            vv.zlabel('[1,0,0] axis')
            vv.gca().SetLimits(rangeX=(0, alignedCTVolume.shape[2]), rangeY=(0,
alignedCTVolume.shape[1]), rangeZ=(0, alignedCTVolume.shape[0]))
            vv.gca().legend = 'CoM line aligned to normal axis', 'PCA[0]
(anteroposterior) axis', 'PCA[1] axis', 'PCA[2] axis', 'Mouse back direction', 'Mouse
head direction'

            vv.figure()
            vv.subplot(4,1,1)
            vv.plot(x, dataRaw,lc='k')
            vv.title('Standard Deviation of bone in Coronal Plane')
            vv.subplot(4,1,2)
            vv.plot(x, dataSmoothed,lc='r')
            vv.title('Smoothed')
            vv.subplot(4,1,3)
            vv.plot(x, dataDiv1,lc='g')
            vv.title('First Derivative')
            vv.subplot(4,1,4)
            vv.plot(x, dataDiv2,lc='b')
            vv.title('Second Derivative')
```

```
            app.Run()

        #create rotation matrix that will align atlas to CTVolume
        alignmentAxis = numpy.empty((3,3))
        alignmentAxis[0,:] = anteriorVector
        alignmentAxis[1,:] = dorsalVector
        alignmentAxis[2,:] = numpy.cross(anteriorVector, dorsalVector)

        scanData.alignmentAxis = alignmentAxis


        #Perform PCA of spine model, second vector will be dorsoventral axis
        spinePCA = spineModel.PCA()
        spinePCA[0] = numpy.asarray(numpy.matrix(spinePCA[0]) * rotMat.I).squeeze()
        spinePCA[1] = numpy.asarray(numpy.matrix(spinePCA[1]) * rotMat.I).squeeze()
        spinePCA[2] = numpy.asarray(numpy.matrix(spinePCA[2]) * rotMat.I).squeeze()
#       ang = numpy.arccos(numpy.dot(spinePCA[0], alignmentAxis[0]))
#       rot = numpyTransform.rotation(ang, numpy.cross(spinePCA[0], alignmentAxis[0]),
N=4)
#       numpyTransform.transformPoints(rot, ang)

        spinePCA[1] = numpy.cross(alignmentAxis[0],spinePCA[2])
        spinePCA[2] = numpy.cross(alignmentAxis[0],spinePCA[1])

        #determine angles between model PCA and Spine PCA axis, they should be similar
(<20 degrees), if not we should to redo the rough align with spine axis
        dif = numpy.zeros(3)
        for i in xrange(spinePCA.shape[0]):
            dif[i] = numpy.min(numpy.degrees(numpy.arccos([numpy.dot(spinePCA[i],
alignmentAxis[i]), numpy.dot(-spinePCA[i], alignmentAxis[i])])))
        print 'Anteroposterior axis, spine & model %f degrees apart' % (dif[0])
        print 'Dorsoventral axis, spine & model %f degrees apart' % (dif[1])
        print 'Left-Right axis, spine & model %f degrees apart' % (dif[2])

        return alignmentAxis, neckPos, scaleFactor, hipLoc, spineModel, tailModel,
spinePCA


def roughAlignProfile():
        import os,cProfile
        '''
        run
            python -m cProfile -o roughAlign.TimeProfile pyAtlasSegmentation.py
        then run
            python runsnake.py roughAlign.TimeProfile
        '''
        if os.path.exists('roughAlign.TimeProfile'):
            os.remove('roughAlign.TimeProfile')
        cProfile.runctx( 'RoughAlignTest()', globals(), locals(),
filename='roughAlign.TimeProfile')
        os.system('runsnake roughAlign.TimeProfile')

def alignVolume(CTVolume, rotMat, verbose=False):
```

```
	#TODO: change function, this is all very confusing because affine_transform()
actually does rotMat.I instead of rotMat
	#all point * mat equations should be mat*point equations, they currently work
because mat is rotmat, not rotmat.I
	rotMat = numpy.matrix(rotMat)

	#list all vertices defining the bounding box formed by volume
	bbPoints = numpy.array([[0.0,                        0.0,
	0.0],
                                        [0.0,                        0.0,
		CTVolume.shape[2]],
                                        [0.0,
	CTVolume.shape[1],  0.0],
                                        [0.0,
	CTVolume.shape[1],  CTVolume.shape[2]],
                                        [CTVolume.shape[0], 0.0,
	0.0],
                                        [CTVolume.shape[0], 0.0,
	CTVolume.shape[2]],
                                        [CTVolume.shape[0], CTVolume.shape[1],
	0.0],
                                        [CTVolume.shape[0], CTVolume.shape[1],
	CTVolume.shape[2]]])

	#recalculate limit
	bbPointsRotated = numpy.empty_like(bbPoints)
	for i in xrange(bbPoints.shape[0]):
		bbPointsRotated[i] = numpy.asarray( numpy.matrix(bbPoints[i]) * rotMat
).squeeze()
	#parts of volume that are rotated into negative areas
	#Take the minimum values of the rotated bounding box, do the inverse rotation
and use the result as the offset to subtract
	offset = bbPointsRotated.min(axis=0)
	offset[offset>0] = 0
	offset = numpy.matrix(offset) * rotMat.I	#TODO: should this be a
transpose instead of invert? does it matter?
	offset = numpy.asarray(offset).squeeze()
	if verbose:
		print 'Offset:', offset
	#calculate shape, for each bounding box vertex subtract offset, then rotate
these points, the max coordinate will tell the shape
	points = numpy.empty_like(bbPoints)
	for i in xrange(bbPoints.shape[0]):
		points[i] = numpy.asarray( numpy.matrix(bbPoints[i]-offset) * rotMat
).squeeze()
	shape = points.max(axis=0)
#	if verbose:
#		print 'Shape:', shape

	#Transform CTvolume to alignedCTVolume
	if verbose:
		t1=time.time()

	#affine_transform subtracts offset then rotates
```

```
	alignedCTVolume = affine_transform(CTVolume, rotMat, offset=offset,
output_shape=shape, cval=-1000)

	#affineTrans is the transform that discribes the transformation done by
affine_transform()
	affineTrans = numpy.matrix(numpy.identity(4))
	affineTrans[:3,:3] = rotMat.I
	affineTrans = affineTrans * numpyTransform.translation(offset).I

	if verbose:
		print 'Time to align CT volume: %f seconds' % (time.time()-t1)
		print 'Reference Volume Shape:', CTVolume.shape
		print 'Aligned Volume Shape:  ', alignedCTVolume.shape

	return alignedCTVolume, rotMat, offset, affineTrans

def createLabelMask(scanData, atlasData):
	#TODO: probably don't want to base mask on unprocessed data, probably smooth
it, don't need to resample
#	ctVolume = scanData.ctField
	ctVolume = gaussian_filter(scanData.ctField, scanData.sigma)

	labelVolume = numpy.zeros(ctVolume.shape, dtype=numpy.int64)

	t1=time.time()
	#calculate transformed models
	atlasData.atlasJoint.transformVertices()
	print 'Time to transform all model vertices:', time.time()-t1

	#see what vertex each point in
	#TODO: maybe there is a smarter way of doing this where we first look at the
bounding boxes to rule out options
	#faster way, check to see distance to bounding box planes. If these are not
better than currentClosestDistance, don't need to check every vertex of this model
	t1=time.time()
	boneIndices = numpy.where(ctVolume >= scanData.isolevel)
	print 'Number of bone voxels to solve for:', boneIndices[0].shape[0]
	for i in xrange(boneIndices[0].shape[0]):
		t2=time.time()
		boneIndex = numpy.array([boneIndices[0][i], boneIndices[1][i],
boneIndices[2][i]])
		closestSqDistance, modelID, modelName =
atlasData.atlasJoint.compareToTransformedPoints(boneIndex)
		labelVolume[boneIndex[0],boneIndex[1],boneIndex[2]] = modelID
#		print '%10.6f%%: Index %s closest to bone ID %d %s. Time to figure this
out %f' %(100.0*i/boneIndices[0].shape[0], str(boneIndex), modelID, modelName,
time.time()-t2)
	print 'Time to create label volume:', time.time()-t1
	return labelVolume

def createLabelMaskKDTree(scanData, atlasData):
	#TODO: probably don't want to base mask on unprocessed data, probably smooth
it, don't need to resample
	#TODO: this is really slow because its a pure python KD tree implementation
#	ctVolume = scanData.ctField
```

```
	ctVolume = gaussian_filter(scanData.ctField, scanData.sigma)

	labelVolume = numpy.zeros(ctVolume.shape, dtype=numpy.int64)

	t1=time.time()
	#calculate transformed models
	atlasData.atlasJoint.transformVertices()
	print 'Time to transform all model vertices:', time.time()-t1

	t1=time.time()
	boneIndices = numpy.where(ctVolume >= scanData.isolevel)
	print 'Number of bone voxels to solve for:', boneIndices[0].shape[0]
	for i in xrange(boneIndices[0].shape[0]):
		t2=time.time()
		boneIndex = numpy.array([boneIndices[0][i], boneIndices[1][i],
boneIndices[2][i]])
		closestSqDistance, modelID, modelName =
atlasData.atlasJoint.compareToTransformedPointsKDTrees(boneIndex)
		labelVolume[boneIndex[0],boneIndex[1],boneIndex[2]] = modelID
#		print '%10.6f%%: Index %s closest to bone ID %d %s. Time to figure this
out %f' %(100.0*i/boneIndices[0].shape[0], str(boneIndex), modelID, modelName,
time.time()-t2)
	print 'Time to create label volume:', time.time()-t1
	return labelVolume


def RoughAlignTest(**kwargs):
	#Start with saved data
	savedData=loadmat('Saved Data.mat')

#	returnValues  = roughAlign(savedData['referenceVolume'], None, 350,
pcaAxis=savedData['axes'],alignedCTVolume=savedData['alignedCTVolume'],**kwargs)
	returnValues  = roughAlign(savedData['referenceVolume'], None, 350,
pcaAxis=savedData['axes'],**kwargs)
	return returnValues

def FineAlign(scanData, atlasData, **kwargs):
	visualize = True
	#apply transform to CT dataset, 1 time thing, this is to account for scaling
	scanData.isosurfaceJoint.transformVertices()
	iso1KDTree = KDTree(scanData.isosurfaceModel1.transformedVertexList)
	tailKDTree = KDTree(kwargs['tailVertices'])
	spineKDTree = KDTree(kwargs['spineVertices'])

	if 'fineAlignTransformsShortcut' in kwargs:
		fineAlignTransforms = loadmat(kwargs['fineAlignTransformsShortcut'])
	else:
		fineAlignTransforms = {}

	if 'spineVertexShortcut' in kwargs:
		vertexDict = loadmat(kwargs['spineVertexShortcut'])
		spineindx = vertexDict['spineindx'].astype(numpy.bool).squeeze()
		tailindx = vertexDict['tailindx'].astype(numpy.bool).squeeze()
	else:
		#find spine indices
```

```
            t1 = time.time()
            spineindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
            spineNN = spineKDTree.query_ball_tree(iso1KDTree,
1.0/scanData.sliceThickness)
            for spinevindx in spineNN:
                  spineindx[spinevindx] = True
            print 'Took %f seconds to calculate spine vertices' % (time.time() - t1)

            #find tail indices
            t1 = time.time()
            tailindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
            tailNN = tailKDTree.query_ball_tree(iso1KDTree,
1.0/scanData.sliceThickness)
            for tailvindx in tailNN:
                  tailindx[tailvindx] = True
            print 'Took %f seconds to calculate tail vertices' % (time.time() - t1)
            if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                  savemat(kwargs['savePrefix']+'spineIndices.mat',
{'spineindx':spineindx, 'tailindx':tailindx})

      ##########################################################################
#########################################################
      #Align Skull
      jointOfInterest = atlasData.atlasJoint
      cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
      bones =['Skull Outside', 'Skull Inside']
      if 'Atlas Base' not in fineAlignTransforms:   #Do ICP
            #step 1: get data vertex list. These vertexes are based on the current
best transform
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

            #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
#           modelIndx =
numpy.ones(scanData.isosurfaceModel.transformedVertexList.shape[0], dtype=numpy.bool)
            modelIndx =
numpy.ones(scanData.isosurfaceModel1.transformedVertexList.shape[0],
dtype=numpy.bool)

#           #remove model data that is behind the neck
#           normV = scanData.alignmentAxis[0]
#           planePoint = jointOfInterest.location
#           pointVecs = scanData.isosurfaceModel.transformedVertexList - planePoint
#           distance = numpy.dot(pointVecs, normV)
#           modelIndx[distance < 0] = False

            #create filtered model point cloud
```

```
#             modelPoints = scanData.isosurfaceModel.transformedVertexList.copy()
              modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()

              #Step 3: Reorient Point Clouds
              #transform points back so that joint of interest is in original
position/orientation
              #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
              #ICP must be done on original points not points midway through a
transformation
              modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
              dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)

              #move point clouds so that joint is at origin
              jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
              modelPoints -= jointLocation
              dataPoints -= jointLocation

              #limit points to only those reachable given the joint DOF
              mindx = numpyTransform.pointsInToleranceRange(modelPoints,
jointOfInterest.DOFvec, jointOfInterest.DOFangle, [jointOfInterest.DOFtrans,
jointOfInterest.DOFtrans, jointOfInterest.DOFtrans])
              modelPoints = modelPoints[numpy.logical_and(mindx, modelIndx)]

              #TODO: Convert the model and data points so that the DOFvec is aligned
with the z? axis this might allow better control of angles (rotz * rotx * rotz)

              #Step 4: Perform actual ICP
              icp = ICP(modelPoints, dataPoints, maxIterations=15,
modelDownsampleFactor=1, dataDownsampleFactor=1, minimizeMethod='fmincon')
              #+- 10mm translation
#             transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor *
scanData.sliceThickness)
              transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
              print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
              initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0,
1.0])
              lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -transBound, -
transBound, 0.8, 0.8, 0.8])
              upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
              #run ICP
              transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
              transform = transform[-1]
              print 'ICP Generated Transform for %s Joint' % (jointOfInterest.name)
              print transform

              if visualize:#display err plot of ICP
                    fig = plt.figure()
                    ax = fig.add_subplot(1,1,1)
```

```
                    ax.plot(t,err,'x--')
                    ax.set_xlabel('Time')
                    ax.set_ylabel('RMS error')
                    ax.set_title('Result of ICP on %s Joint' %(jointOfInterest.name))
                    plt.show()

            if visualize and False:
                    #entire model
                    allModelPoints =
scanData.isosurfaceModel1.transformedVertexList.copy()
                    allModelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,allModelPoints)
                    allModelPoints -= jointLocation
                    tri = numpy.array(range(allModelPoints.shape[0]-
allModelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(allModelPoints, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Model Point Cloud for %s Joint' %
(jointOfInterest.name), displayAsPoints=True, color=[0.0,0.0,1.0])

                    #Display initial filtered points clouds that will be passed to
ICP
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                    #display point clouds at final position
                    modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
```

```
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

      else:  #skip ICP, just use predetermined transform
            transform = numpy.matrix(fineAlignTransforms['Atlas Base'])
#           transform = numpy.matrix(
#                 [[  9.99422678e-01,   1.63207412e-03,   3.39359326e-02,  -
8.70562491e-01],
#                  [ -2.48435746e-03,   9.99682172e-01,   2.50875024e-02,  -
8.82143072e+00],
#                  [ -3.38842021e-02,  -2.51573278e-02,   9.99109088e-01,
1.04611786e+01],
#                  [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
1.00000000e+00]])

      #store transform
      fineAlignTransforms[jointOfInterest.name] = transform

      #Step 5: Apply ICP transform to joint
      R = numpy.matrix(numpy.identity(4))
      R[:3,:3] = transform[:3,:3]
      jointOfInterest.rotate(R, relative=True)
      cummulativeJointRotation = numpy.matrix(numpy.identity(4))
      cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
      modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
      #translation has to include rotation effects of cumulative transformations
      jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)

      if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
```

```
                kwargs['updateSceneFunc']()
                time.sleep(3)

        #print resulting joint location
        if 'getResults' in kwargs and kwargs['getResults']:
                atlasData.atlasJoint.transformVertices()
                print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
                meanDist = []
                for bone in bones:
                        d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                        meanDist.append(numpy.sum(d)/len(i))
                print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
                dist = 0.0
                numPoints = 0.0
                for bone in scanData.bonesVol1:
                        d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                        dist += numpy.sum(d)
                        numPoints += len(i)
                print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)



        if False:    #getting the skull indices might not be necessary if we are
blindly removing front half anyway.
                #get indices of model vertices within 1mm of aligned atlas skull
vertices
                if 'skullVertexShortcut' in kwargs and
os.path.exists(kwargs['skullVertexShortcut']):
                        vertexDict = loadmat(kwargs['skullVertexShortcut'])
                        skullindx = vertexDict['indx'].astype(numpy.bool).squeeze()
                else:
                        #create KDTree of newly aligned bone
                        dataPoints = []
                        for bone in bones:

        atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                                dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
                        dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
                        boneKDTree = KDTree(dataPoints)
                        #find spine indices
                        t1 = time.time()
                        skullindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
                        NN = boneKDTree.query_ball_tree(iso1KDTree,
1.0/scanData.sliceThickness)
                        for vindx in NN:
                                skullindx[vindx] = True
```

```
                    print 'Took %f seconds to calculate %s vertices' % (time.time() -
t1, bone)
                    if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                          savemat(kwargs['savePrefix']+'skullvertices.mat',
{'indx':skullindx})
      else:
            skullindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)

      ##########################################################################
#############################################################
      #align neck and all subjoints
      for joint in atlasData.atlasJoint.childJoints:
            if joint.name == 'Neck':
                  neckJoint = joint
                  break
      else:
            raise Exception('Correct Joint not found')

      ##########################################################################
#############################################################
      #translate hip complex to the previously determined hip location
      for joint in neckJoint.childJoints:
            if joint.name == 'Hip Complex':
                  hipComplexJoint = joint
                  break
      else:
            raise Exception('Correct Joint not found')
      if 'hipLocation' in kwargs:      #hip joint needs to be defined already
            jointOfInterest = hipComplexJoint
            cummulativeParentJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest.parentJoint))

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels['Pelvi
s Right']))
            hipStartLocation = atlasData.atlasModels['Pelvis
Right'].joint.parentJoint.location.squeeze()
            hipLocation = kwargs['hipLocation']
            hipoffset = hipLocation-hipStartLocation
            #need to compensate for the effects of parent joint cumulative transform
            tform = numpy.matrix(numpy.identity(4))
            tform[:3,:3] = cummulativeParentJointTransform[:3,:3]
            hipoffset = numpyTransform.transformPoints(tform.I, hipoffset)
            jointOfInterest.translate(hipoffset, absolute=False)

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels['Pelvi
s Right']))

      ##########################################################################
#############################################################
      #align Left Hip
      for joint in hipComplexJoint.childJoints:
            if joint.name == 'Hip Left':
                  hipLeft = joint
                  break
      else:
```

```
            raise Exception('Correct Joint not found')
      jointOfInterest = hipLeft
      cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
      bones =['Pelvis Left']
      if 'Hip Left' not in fineAlignTransforms:     #Do ICP
            #step 1: get data vertex list. These vertexes are based on the current
best transform
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

            #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
            #create filtered model point cloud
            modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
            modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #remove front half of model data
            normV = scanData.alignmentAxis[0]
            planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
            pointVecs = modelPoints - planePoint
            distance = numpy.dot(pointVecs, normV)
            modelIndx[distance > 0] = False

#           #remove right half of model data
#           normV = scanData.alignmentAxis[2]
#           planePoint = jointOfInterest.location
#           pointVecs = modelPoints - planePoint
#           distance = numpy.dot(pointVecs, normV)
#           modelIndx[distance > 0] = False

            #remove previously aligned vertecies
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))

            #Step 3: Reorient Point Clouds
            #transform points back so that joint of interest is in original
position/orientation
            #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
            #ICP must be done on original points not points midway through a
transformation
            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)
```

```
            #move point clouds so that joint is at origin
            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
            modelPoints -= jointLocation
            dataPoints -= jointLocation

            #Remove any points now that model is reoriented
            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #Get just points of interest
            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]

            #limit translation
#           transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor *
scanData.sliceThickness)
            transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
            print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
            #Step 4: Perform actual ICP
            icp = ICP(modelPoints, dataPoints, maxIterations=15,
modelDownsampleFactor=1, dataDownsampleFactor=1, minimizeMethod='fmincon')
            initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0,
1.0])
            lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -transBound, -
transBound, 0.8, 0.8, 0.8])
            upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
            transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
            transform = transform[-1]
            print 'ICP Generated Transform for %s Joint' % (jointOfInterest.name)
            print transform

            if visualize:#display err plot of ICP
                  fig = plt.figure()
                  ax = fig.add_subplot(1,1,1)
                  ax.plot(t,err,'x--')
                  ax.set_xlabel('Time')
                  ax.set_ylabel('RMS error')
                  ax.set_title('Result of ICP on %s Joint' %(jointOfInterest.name))
                  plt.show()

            if visualize and False:
                  #Display initial filtered points clouds that will be passed to
ICP
                  tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                  tri = tri.reshape((tri.shape[0]/3,3))
                  TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                  tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                  tri = tri.reshape((tri.shape[0]/3,3))
```

```
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                    #display point clouds at final position
                    modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

      else:  #skip ICP, just use predetermined transform
            transform = numpy.matrix(fineAlignTransforms['Hip Left'])
#           transform = numpy.matrix(
#                 [[ 0.94610378, -0.28824402, -0.14765847, -2.10207738],
#                  [ 0.19292585,  0.86781307, -0.45790839, -3.71208068],
#                  [ 0.2601293,   0.40474173,  0.87665095,  7.22883351],
```

```
#                   [ 0.,          0.,          0.,          1.        ]])

        #store transform
        fineAlignTransforms[jointOfInterest.name] = transform

        #Step 5: Apply ICP transform to joint
        R = numpy.matrix(numpy.identity(4))
        R[:3,:3] = transform[:3,:3]
        jointOfInterest.rotate(R, relative=True)
        cummulativeJointRotation = numpy.matrix(numpy.identity(4))
        cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
        modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
        #translation has to include rotation effects of cumulative transformations
#       jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
        jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

        if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
                kwargs['updateSceneFunc']()
                time.sleep(3)

        #print resulting joint location
        if 'getResults' in kwargs and kwargs['getResults']:
                atlasData.atlasJoint.transformVertices()
                print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
                meanDist = []
                for bone in bones:
                        d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                        meanDist.append(numpy.sum(d)/len(i))
                print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
                dist = 0.0
                numPoints = 0.0
                for bone in scanData.bonesVol1:
                        d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                        dist += numpy.sum(d)
                        numPoints += len(i)
                print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)


        #get indices of model vertices within 1mm of aligned atlas hip left vertices
        if 'hipLeftVertexShortcut' in kwargs and
os.path.exists(kwargs['hipLeftVertexShortcut']):
                vertexDict = loadmat(kwargs['hipLeftVertexShortcut'])
                hipLeftindx = vertexDict['indx'].astype(numpy.bool).squeeze()
        else:
                #create KDTree of newly aligned bone
                dataPoints = []
```

```
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
            boneKDTree = KDTree(dataPoints)
            #find spine indices
            t1 = time.time()
            hipLeftindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
            NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
            for vindx in NN:
                  hipLeftindx[vindx] = True
            print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
            if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                  savemat(kwargs['savePrefix']+'hipLeftvertices.mat',
{'indx':hipLeftindx})

      ##########################################################################
#########################################################
      #align Right Hip
      for joint in hipComplexJoint.childJoints:
            if joint.name == 'Hip Right':
                  hipRight = joint
                  break
      else:
            raise Exception('Correct Joint not found')
      jointOfInterest = hipRight
      cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
      bones =['Pelvis Right']
      if 'Hip Right' not in fineAlignTransforms:    #Do ICP
            #step 1: get data vertex list. These vertexes are based on the current
best transform
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

            #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
            #create filtered model point cloud
            modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
            modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #remove front half of model data
            normV = scanData.alignmentAxis[0]
            planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
```

```
            pointVecs = modelPoints - planePoint
            distance = numpy.dot(pointVecs, normV)
            modelIndx[distance > 0] = False

#           #remove Left half of model data
#           normV = scanData.alignmentAxis[2]
#           planePoint = jointOfInterest.location
#           pointVecs = modelPoints - planePoint
#           distance = numpy.dot(pointVecs, normV)
#           modelIndx[distance < 0] = False

            #remove previously aligned vertices
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(hipLeftindx))

            #Step 3: Reorient Point Clouds
            #transform points back so that joint of interest is in original
position/orientation
            #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
            #ICP must be done on original points not points midway through a
transformation
            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)

            #move point clouds so that joint is at origin
            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
            modelPoints -= jointLocation
            dataPoints -= jointLocation

            #Remove any points now that model is reoriented
            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #Get just points of interest
            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]

            #limit translation
#           transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor *
scanData.sliceThickness)
            transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
            print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
            #Step 4: Perform actual ICP
            icp = ICP(modelPoints, dataPoints, maxIterations=15,
modelDownsampleFactor=1, dataDownsampleFactor=1, minimizeMethod='fmincon')
            initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0,
1.0])
            lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -transBound, -
transBound, 0.8, 0.8, 0.8])
```

```
            upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
            transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
            transform = transform[-1]
            print 'ICP Generated Transform for %s Joint' % (jointOfInterest.name)
            print transform

            if visualize:#display err plot of ICP
                  fig = plt.figure()
                  ax = fig.add_subplot(1,1,1)
                  ax.plot(t,err,'x--')
                  ax.set_xlabel('Time')
                  ax.set_ylabel('RMS error')
                  ax.set_title('Result of ICP on %s Joint' %(jointOfInterest.name))
                  plt.show()

            if visualize and False:
                  #Display initial filtered points clouds that will be passed to
ICP
                  tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                  tri = tri.reshape((tri.shape[0]/3,3))
                  TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                  tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                  tri = tri.reshape((tri.shape[0]/3,3))
                  TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                  dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                  tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                  tri = tri.reshape((tri.shape[0]/3,3))
                  TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                  #display point clouds at final position
                  modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                  modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                  tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                  tri = tri.reshape((tri.shape[0]/3,3))
                  TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])
```

```
                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

      else:  #skip ICP, just use predetermined transform
            transform = numpy.matrix(fineAlignTransforms['Hip Right'])
#           transform = numpy.matrix(
#                 [[ 0.96487629, -0.07852993, -0.25069264,  5.96225278],
#                  [-0.03107695,  0.91345488, -0.40575166,  4.20753782],
#                  [ 0.26086007,  0.39929092,  0.87893048,  4.29578447],
#                  [ 0.,          0.,          0.,          1.        ]])

      #store transform
      fineAlignTransforms[jointOfInterest.name] = transform

      #Step 5: Apply ICP transform to joint
      R = numpy.matrix(numpy.identity(4))
      R[:3,:3] = transform[:3,:3]
      jointOfInterest.rotate(R, relative=True)
      cummulativeJointRotation = numpy.matrix(numpy.identity(4))
      cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
      modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
      #translation has to include rotation effects of cumulative transformations
#     jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
      jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

      if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
            kwargs['updateSceneFunc']()
            time.sleep(3)

      #print resulting joint location
      if 'getResults' in kwargs and kwargs['getResults']:
            atlasData.atlasJoint.transformVertices()
```

```
            print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
            meanDist = []
            for bone in bones:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  meanDist.append(numpy.sum(d)/len(i))
            print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
            dist = 0.0
            numPoints = 0.0
            for bone in scanData.bonesVol1:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  dist += numpy.sum(d)
                  numPoints += len(i)
            print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)


      #get indices of model vertices within 1mm of aligned atlas hip left vertices
      if 'hipRightVertexShortcut' in kwargs and
os.path.exists(kwargs['hipRightVertexShortcut']):
            vertexDict = loadmat(kwargs['hipRightVertexShortcut'])
            hipRightindx = vertexDict['indx'].astype(numpy.bool).squeeze()
      else:
            #create KDTree of newly aligned bone
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
            boneKDTree = KDTree(dataPoints)
            #find spine indices
            t1 = time.time()
            hipRightindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
            NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
            for vindx in NN:
                  hipRightindx[vindx] = True
            print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
            if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                  savemat(kwargs['savePrefix']+'hipRightvertices.mat',
{'indx':hipRightindx})


      ##########################################################################
#########################################################
      #align Left Femur
      for joint in hipLeft.childJoints:
```

```
            if joint.name == 'Pelvis Hindlimb Left':
                  femurLeft = joint
                  break
      else:
            raise Exception('Correct Joint not found')
      jointOfInterest = femurLeft
      cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
      bones =['Upper Hindlimb Left']
      if 'Pelvis Hindlimb Left' not in fineAlignTransforms:      #Do ICP
#     if True:     #Do ICP
            #step 1: get data vertex list. These vertexes are based on the current
best transform
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

            #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
            #create filtered model point cloud
            modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
            modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #remove front half of model data
            normV = scanData.alignmentAxis[0]
            planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
            pointVecs = modelPoints - planePoint
            distance = numpy.dot(pointVecs, normV)
            modelIndx[distance > 0] = False

            #remove previously aligned vertecies
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(hipLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(hipRightindx))

            #Step 3: Reorient Point Clouds
            #transform points back so that joint of interest is in original
position/orientation
            #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
            #ICP must be done on original points not points midway through a
transformation
            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)
```

```
            #move point clouds so that joint is at origin
            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
            modelPoints -= jointLocation
            dataPoints -= jointLocation

            #Remove any points now that model is reoriented
            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #Get just points of interest
            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]

            #create different starting transformations
            cummJointTransferRotOnly = numpy.matrix(numpy.identity(4))
            cummJointTransferRotOnly[:3,:3] = cummulativeJointTransform[:3,:3]
            proximodistalVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.proximodist
alVecTransformed)
            secondaryVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.secondaryVe
cTransformed)
            initialTransforms = []
            initialTransformsError = []
            icpTransformsErr = []
            icpTransforms = []
            icpT = []

            for spin in numpy.arange(0, 2*numpy.pi, 3*numpy.pi):
                  for elevation in numpy.arange(0, numpy.pi, numpy.pi/2):
                        for aximuth in numpy.arange(0, 2*numpy.pi, numpy.pi/2):
                              initialTransforms.append(
numpyTransform.rotation(aximuth, proximodistalVecTransformed, N=4) *
numpyTransform.rotation(elevation, secondaryVecTransformed, N=4) *
numpyTransform.rotation(spin, proximodistalVecTransformed, N=4) )

            for i in xrange(len(initialTransforms)):
                  initialTransform = initialTransforms[i]
                  dataPointsInitialTrandform =
numpyTransform.transformPoints(initialTransform, dataPoints)

                  #limit translation
#                 transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor
* scanData.sliceThickness)
                  transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
                  print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
                  #Step 4: Perform actual ICP
                  print 'ITERATION',i
                  icp = ICP(modelPoints, dataPointsInitialTrandform,
maxIterations=15, modelDownsampleFactor=1, dataDownsampleFactor=1,
minimizeMethod='fmincon')
                  initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
1.0, 1.0])
```

```
                    lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -
transBound, -transBound, 0.8, 0.8, 0.8])
                    upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
                    transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
                    del icp
                    transform = transform[-1]
                    print 'ICP Generated Transform for %s Joint' %
(jointOfInterest.name)
                    print transform

                    icpT.append(t)
                    icpTransformsErr.append(err)
                    icpTransforms.append(transform)

                    if visualize and False:   #display err plot of ICP
                          fig = plt.figure()
                          ax = fig.add_subplot(1,1,1)
                          ax.plot(t,err,'x--')
                          ax.set_xlabel('Time')
                          ax.set_ylabel('RMS error')
                          ax.set_title('Result of ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, len(initialTransformsError)))
                          plt.show()

             icpTransformsErr = numpy.array(icpTransformsErr)

             tindx = numpy.where(icpTransformsErr==icpTransformsErr.min())[0][0]
             transform = icpTransforms[tindx] * initialTransforms[tindx]
             print 'Best iteration was %d with transform:' % (tindx)
             print transform

             if visualize:#display err plot of ICP
                    fig = plt.figure()
                    ax = fig.add_subplot(1,1,1)
                    ax.plot(icpT[tindx],icpTransformsErr[tindx],'x--')
                    ax.set_xlabel('Time')
                    ax.set_ylabel('RMS error')
                    ax.set_title('Result of Scale ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, tindx))
                    plt.show()

             if visualize and False:
                    #Display initial filtered points clouds that will be passed to
ICP
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
```

```
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                    #display point clouds at final position
                    modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

            else:  #skip ICP, just use predetermined transform
                    transform = numpy.matrix(fineAlignTransforms['Pelvis Hindlimb Left'])

            #store transform
            fineAlignTransforms[jointOfInterest.name] = transform
```

```
	#Step 5: Apply ICP transform to joint
	R = numpy.matrix(numpy.identity(4))
	R[:3,:3] = transform[:3,:3]
	jointOfInterest.rotate(R, relative=True)
	cummulativeJointRotation = numpy.matrix(numpy.identity(4))
	cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
	modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
	#translation has to include rotation effects of cumulative transformations
#	jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
	jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

#	if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
#		kwargs['updateSceneFunc']()
#		time.sleep(3)
#
#	#print resulting joint location
#	if 'getResults' in kwargs and kwargs['getResults']:
#		atlasData.atlasJoint.transformVertices()
#		print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
#		meanDist = []
#		for bone in bones:
#			d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
#			meanDist.append(numpy.sum(d)/len(i))
#		print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
#		dist = 0.0
#		numPoints = 0.0
#		for bone in scanData.bonesVol1:
#			d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
#			dist += numpy.sum(d)
#			numPoints += len(i)
#		print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)
#
#	#get indices of model vertices within 1mm of aligned atlas hip left vertices
#	if 'femurLeftVertexShortcut' in kwargs and
os.path.exists(kwargs['femurLeftVertexShortcut']):
#		vertexDict = loadmat(kwargs['femurLeftVertexShortcut'])
#		femurLeftindx = vertexDict['indx'].astype(numpy.bool).squeeze()
#	else:
#		#create KDTree of newly aligned bone
#		dataPoints = []
#		for bone in bones:
#
	atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
```

```
#                   dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
#             dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
#             boneKDTree = KDTree(dataPoints)
#             #find spine indices
#             t1 = time.time()
#             femurLeftindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
#             NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
#             for vindx in NN:
#                   femurLeftindx[vindx] = True
#             print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
#             if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
#                   savemat(kwargs['savePrefix']+'femurLeftvertices.mat',
{'indx':femurLeftindx})

#      #Scale Left Femur
#      cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
#      bones =['Upper Hindlimb Left']
##     if 'Pelvis Hindlimb Left' not in fineAlignTransforms:      #Do ICP
#      if True:     #Do ICP
#             #step 1: get data vertex list. These vertexes are based on the current
best transform
#             dataPoints = []
#             for bone in bones:
#
       atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
#                   dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
#             dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
#
#             #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
#             #create filtered model point cloud
#             modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
#             modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)
#
#             #remove front half of model data
#             normV = scanData.alignmentAxis[0]
#             planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
#             pointVecs = modelPoints - planePoint
#             distance = numpy.dot(pointVecs, normV)
#             modelIndx[distance > 0] = False
#
#             #remove previously aligned vertecies
#             modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
#             modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
#             modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))
#             modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(hipLeftindx))
#             modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(hipRightindx))
#
```

```
#            #Step 3: Reorient Point Clouds
#            #transform points back so that joint of interest is in original
position/orientation
#            #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
#            #ICP must be done on original points not points midway through a
transformation
#            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
#            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)
#
#            #move point clouds so that joint is at origin
#            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
#            modelPoints -= jointLocation
#            dataPoints -= jointLocation
#
#            #Remove any points now that model is reoriented
#            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)
#
#            #Get just points of interest
#            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]
#
#            #create different starting transformations
#            cummJointTransferRotOnly = numpy.matrix(numpy.identity(4))
#            cummJointTransferRotOnly[:3,:3] = cummulativeJointTransform[:3,:3]
#            proximodistalVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.proximodist
alVecTransformed)
#            secondaryVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.secondaryVe
cTransformed)
#            initialTransforms = []
#            initialTransformsError = []
#            icpTransformsErr = []
#            icpTransforms = []
#            icpT = []
#
#            initialTransforms = [numpy.matrix(numpy.identity(4))]
#            #scaling transform
#            for initialTransform in initialTransforms:
#                  dataPointsInitialTrandform =
numpyTransform.transformPoints(initialTransform, dataPoints)
#
#                  #Step 4: Perform actual ICP
#                  transBound = jointOfInterest.DOFtrans / scanData.sliceThickness

#                  print 'Scaling ICP'
#                  icp = ICP(modelPoints, dataPointsInitialTrandform,
maxIterations=100, modelDownsampleFactor=1, dataDownsampleFactor=1,
minimizeMethod='fmincon')
#                  initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
1.0, 1.0])
```

```
#                   lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -
transBound, -transBound, 0.8, 0.8, 0.8])
#                   upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
#                   transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds,scaleOnlyIso=True)
#                   del icp
#                   transform = transform[-1]
#                   print 'ICP Generated Transform for %s Joint' %
(jointOfInterest.name)
#                   print transform
#
#                   icpT.append(t)
#                   icpTransformsErr.append(err)
#                   icpTransforms.append(transform)
#
#                   if visualize and False:   #display err plot of ICP
#                         fig = plt.figure()
#                         ax = fig.add_subplot(1,1,1)
#                         ax.plot(t,err,'x--')
#                         ax.set_xlabel('Time')
#                         ax.set_ylabel('RMS error')
#                         ax.set_title('Result of ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, len(initialTransformsError)))
#                         plt.show()
#
#             icpTransformsErr = numpy.array(icpTransformsErr)
#
#             tindx = numpy.where(icpTransformsErr==icpTransformsErr.min())[0][0]
#             transform = icpTransforms[tindx] * initialTransforms[tindx]
#             print 'Best Scale iteration was %d with transform:' % (tindx)
#             print transform
#
#             if visualize:#display err plot of ICP
#                   fig = plt.figure()
#                   ax = fig.add_subplot(1,1,1)
#                   ax.plot(icpT[tindx],icpTransformsErr[tindx],'x--')
#                   ax.set_xlabel('Time')
#                   ax.set_ylabel('RMS error')
#                   ax.set_title('Result of Scale ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, tindx))
#                   plt.show()
#
#             if visualize and False:
#                   #Display initial filtered points clouds that will be passed to
ICP
#                   tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
#                   tri = tri.reshape((tri.shape[0]/3,3))
#                   TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])
#
#                   tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
```

```
#                   tri = tri.reshape((tri.shape[0]/3,3))
#                   TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])
#
#                   dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
#                   tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
#                   tri = tri.reshape((tri.shape[0]/3,3))
#                   TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])
#
#                   #display point clouds at final position
#                   modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
#                   modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
#                   tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
#                   tri = tri.reshape((tri.shape[0]/3,3))
#                   TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])
#
#                   dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
#                   dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
#                   tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
#                   tri = tri.reshape((tri.shape[0]/3,3))
#                   TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])
#
#                   dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
#                   dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
#                   tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
#                   tri = tri.reshape((tri.shape[0]/3,3))
#                   TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])
#
#     else:  #skip ICP, just use predetermined transform
#            transform = numpy.matrix(fineAlignTransforms['Pelvis Hindlimb Left'])
#
#     #store transform
#     fineAlignTransforms[jointOfInterest.name] = transform
```

```
#
#	#Step 5: Apply ICP transform to joint
#	R = numpy.matrix(numpy.identity(4))
#	R[:3,:3] = transform[:3,:3]
#	jointOfInterest.rotate(R, relative=True)
#	cummulativeJointRotation = numpy.matrix(numpy.identity(4))
#	cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
#	modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
	#translation has to include rotation effects of cumulative transformations
##	jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
#	jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

	if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
		kwargs['updateSceneFunc']()
		time.sleep(3)

	#print resulting joint location
	if 'getResults' in kwargs and kwargs['getResults']:
		atlasData.atlasJoint.transformVertices()
		print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
		meanDist = []
		for bone in bones:
			d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
			meanDist.append(numpy.sum(d)/len(i))
		print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
		dist = 0.0
		numPoints = 0.0
		for bone in scanData.bonesVol1:
			d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
			dist += numpy.sum(d)
			numPoints += len(i)
		print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)

	#get indices of model vertices within 1mm of aligned atlas hip left vertices
	if 'femurLeftVertexShortcut' in kwargs and
os.path.exists(kwargs['femurLeftVertexShortcut']):
		vertexDict = loadmat(kwargs['femurLeftVertexShortcut'])
		femurLeftindx = vertexDict['indx'].astype(numpy.bool).squeeze()
	else:
		#create KDTree of newly aligned bone
		dataPoints = []
		for bone in bones:

	atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
```

```
                        dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
                dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
                boneKDTree = KDTree(dataPoints)
                #find spine indices
                t1 = time.time()
                femurLeftindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
                NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
                for vindx in NN:
                        femurLeftindx[vindx] = True
                print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
                if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                        savemat(kwargs['savePrefix']+'femurLeftvertices.mat',
{'indx':femurLeftindx})

        ##########################################################################
##########################################################
        #align Right Femur
        for joint in hipRight.childJoints:
                if joint.name == 'Pelvis Hindlimb Right':
                        femurRight = joint
                        break
        else:
                raise Exception('Correct Joint not found')
        jointOfInterest = femurRight
        cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
        bones =['Upper Hindlimb Right']
        if 'Pelvis Hindlimb Right' not in fineAlignTransforms:     #Do ICP
                #step 1: get data vertex list. These vertexes are based on the current
best transform
                dataPoints = []
                for bone in bones:

        atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                        dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
                dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

                #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
                #create filtered model point cloud
                modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
                modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

                #remove front half of model data
                normV = scanData.alignmentAxis[0]
                planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
                pointVecs = modelPoints - planePoint
                distance = numpy.dot(pointVecs, normV)
                modelIndx[distance > 0] = False
```

```
            #remove previously aligned vertecies
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(hipLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(hipRightindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurLeftindx))

            #Step 3: Reorient Point Clouds
            #transform points back so that joint of interest is in original
position/orientation
            #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
            #ICP must be done on original points not points midway through a
transformation
            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)

            #move point clouds so that joint is at origin
            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
            modelPoints -= jointLocation
            dataPoints -= jointLocation

            #Remove any points now that model is reoriented
            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #Get just points of interest
            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]

            #create different starting transformations
            cummJointTransferRotOnly = numpy.matrix(numpy.identity(4))
            cummJointTransferRotOnly[:3,:3] = cummulativeJointTransform[:3,:3]
            proximodistalVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.proximodist
alVecTransformed)
            secondaryVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.secondaryVe
cTransformed)
            initialTransforms = []
            initialTransformsError = []
            icpTransformsErr = []
            icpTransforms = []
            icpT = []

            for spin in numpy.arange(0, 2*numpy.pi, 3*numpy.pi):
                  for elevation in numpy.arange(0, numpy.pi, numpy.pi/3):
                        for aximuth in numpy.arange(0, 2*numpy.pi, numpy.pi/3):
                              initialTransforms.append(
numpyTransform.rotation(aximuth, proximodistalVecTransformed, N=4) *
```

```
numpyTransform.rotation(elevation, secondaryVecTransformed, N=4) *
numpyTransform.rotation(spin, proximodistalVecTransformed, N=4) )

            for i in xrange(len(initialTransforms)):
                  initialTransform = initialTransforms[i]
                  dataPointsInitialTrandform =
numpyTransform.transformPoints(initialTransform, dataPoints)

                  #limit translation
#                  transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor
* scanData.sliceThickness)
                  transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
                  print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
                  #Step 4: Perform actual ICP
                  print 'ITERATION',i
                  icp = ICP(modelPoints, dataPointsInitialTrandform,
maxIterations=15, modelDownsampleFactor=1, dataDownsampleFactor=1,
minimizeMethod='fmincon')
                  initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
1.0, 1.0])
                  lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -
transBound, -transBound, 0.8, 0.8, 0.8])
                  upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
                  transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
                  del icp
                  transform = transform[-1]
                  print 'ICP Generated Transform for %s Joint' %
(jointOfInterest.name)
                  print transform

                  icpT.append(t)
                  icpTransformsErr.append(err)
                  icpTransforms.append(transform)

                  if visualize and False:   #display err plot of ICP
                        fig = plt.figure()
                        ax = fig.add_subplot(1,1,1)
                        ax.plot(t,err,'x--')
                        ax.set_xlabel('Time')
                        ax.set_ylabel('RMS error')
                        ax.set_title('Result of ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, len(initialTransformsError)))
                        plt.show()

            icpTransformsErr = numpy.array(icpTransformsErr)

            tindx = numpy.where(icpTransformsErr==icpTransformsErr.min())[0][0]
            transform = icpTransforms[tindx] * initialTransforms[tindx]
            print 'Best iteration was %d with transform:' % (tindx)
            print transform

            if visualize:#display err plot of ICP
```

```
                    fig = plt.figure()
                    ax = fig.add_subplot(1,1,1)
                    ax.plot(icpT[tindx],icpTransformsErr[tindx],'x--')
                    ax.set_xlabel('Time')
                    ax.set_ylabel('RMS error')
                    ax.set_title('Result of ICP on %s Joint, initial transform %d'
%(jointOfInterest.name, tindx))
                    plt.show()

            if visualize and False:
                    #Display initial filtered points clouds that will be passed to
ICP
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                    #display point clouds at final position
                    modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
```

```
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

      else:  #skip ICP, just use predetermined transform
            transform = numpy.matrix(fineAlignTransforms['Pelvis Hindlimb Right'])
#           transform = numpy.matrix(
#                 [[ 0.45810115,  0.3559261,   0.81453051,  8.06362508],
#                  [ 0.88408411, -0.27769195, -0.37587559,  8.33369202],
#                  [ 0.09240463,  0.89230252, -0.44187961, -6.87864907],
#                  [ 0.,          0.,          0.,          1.,        ]])

      #store transform
      fineAlignTransforms[jointOfInterest.name] = transform

      #Step 5: Apply ICP transform to joint
      R = numpy.matrix(numpy.identity(4))
      R[:3,:3] = transform[:3,:3]
      jointOfInterest.rotate(R, relative=True)
      cummulativeJointRotation = numpy.matrix(numpy.identity(4))
      cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
      modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
      #translation has to include rotation effects of cumulative transformations
#     jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
      jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

      if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
            kwargs['updateSceneFunc']()
            time.sleep(3)

      #print resulting joint location
      if 'getResults' in kwargs and kwargs['getResults']:
            atlasData.atlasJoint.transformVertices()
            print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
            meanDist = []
            for bone in bones:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  meanDist.append(numpy.sum(d)/len(i))
```

```
            print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
            dist = 0.0
            numPoints = 0.0
            for bone in scanData.bonesVol1:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  dist += numpy.sum(d)
                  numPoints += len(i)
            print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)

      #get indices of model vertices within 1mm of aligned atlas hip left vertices
      if 'femurRightVertexShortcut' in kwargs and
os.path.exists(kwargs['femurRightVertexShortcut']):
            vertexDict = loadmat(kwargs['femurRightVertexShortcut'])
            femurRightindx = vertexDict['indx'].astype(numpy.bool).squeeze()
      else:
            #create KDTree of newly aligned bone
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
            boneKDTree = KDTree(dataPoints)
            #find spine indices
            t1 = time.time()
            femurRightindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
            NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
            for vindx in NN:
                  femurRightindx[vindx] = True
            print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
            if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                  savemat(kwargs['savePrefix']+'femurRightvertices.mat',
{'indx':femurRightindx})

      ##########################################################################
##############################################################
      #align Left Tibia
      for joint in femurLeft.childJoints:
            if joint.name == 'Hindlimb Left Knee':
                  tibiaLeft = joint
                  break
      else:
            raise Exception('Correct Joint not found')
      jointOfInterest = tibiaLeft
      cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
      bones =['Lower Hindlimb Left']
      if 'Hindlimb Left Knee' not in fineAlignTransforms: #Do ICP
```

```
#      if True:
            #step 1: get data vertex list. These vertexes are based on the current
best transform
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

            #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
            #create filtered model point cloud
            modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
            modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #remove front half of model data
            normV = scanData.alignmentAxis[0]
            planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
            pointVecs = modelPoints - planePoint
            distance = numpy.dot(pointVecs, normV)
            modelIndx[distance > 0] = False

            #remove previously aligned vertecies
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(hipLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(hipRightindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurRightindx))

            #Step 3: Reorient Point Clouds
            #transform points back so that joint of interest is in original
position/orientation
            #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
            #ICP must be done on original points not points midway through a
transformation
            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)

            #move point clouds so that joint is at origin
            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
            modelPoints -= jointLocation
            dataPoints -= jointLocation
```

```
            #Remove any points now that model is reoriented
            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #Get just points of interest
            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]

            #create different starting transformations
            cummJointTransferRotOnly = numpy.matrix(numpy.identity(4))
            cummJointTransferRotOnly[:3,:3] = cummulativeJointTransform[:3,:3]
            proximodistalVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.proximodist
alVecTransformed)
            secondaryVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.secondaryVe
cTransformed)
            initialTransforms = []
            initialTransformsError = []
            icpTransformsErr = []
            icpTransforms = []
            icpT = []

            for spin in numpy.arange(0, 2*numpy.pi, 3*numpy.pi):
                  for elevation in numpy.arange(0, numpy.pi, numpy.pi/4):
                        for aximuth in numpy.arange(0, 2*numpy.pi, numpy.pi/4):
                              initialTransforms.append(
numpyTransform.rotation(aximuth, proximodistalVecTransformed, N=4) *
numpyTransform.rotation(elevation, secondaryVecTransformed, N=4) *
numpyTransform.rotation(spin, proximodistalVecTransformed, N=4) )

#            initialTransforms = initialTransforms[24:26]

            for i in xrange(len(initialTransforms)):
                  initialTransform = initialTransforms[i]
                  dataPointsInitialTrandform =
numpyTransform.transformPoints(initialTransform, dataPoints)

                  #limit translation
#                 transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor
* scanData.sliceThickness)
                  transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
                  print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
                  #Step 4: Perform actual ICP
                  print 'ITERATION',i
                  icp = ICP(modelPoints, dataPointsInitialTrandform,
maxIterations=15, modelDownsampleFactor=1, dataDownsampleFactor=1,
minimizeMethod='fmincon')
                  initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
1.0, 1.0])
                  lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -
transBound, -transBound, 0.8, 0.8, 0.8])
                  upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
```

```
                    transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
                    del icp
                    transform = transform[-1]
                    print 'ICP Generated Transform for %s Joint' %
(jointOfInterest.name)
                    print transform

                    if transform[0,3] > upperBounds[3] or transform[0,3] <
lowerBounds[3]:
                          print 'x translation incorrect'
                    if transform[1,3] > upperBounds[4] or transform[1,3] <
lowerBounds[4]:
                          print 'Y translation incorrect'
                    if transform[2,3] > upperBounds[5] or transform[2,3] <
lowerBounds[5]:
                          print 'Z translation incorrect'

                    icpT.append(t)
                    icpTransformsErr.append(err)
                    icpTransforms.append(transform)

                    if visualize and False:   #display err plot of ICP
                          fig = plt.figure()
                          ax = fig.add_subplot(1,1,1)
                          ax.plot(t,err,'x--')
                          ax.set_xlabel('Time')
                          ax.set_ylabel('RMS error')
                          ax.set_title('Result of ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, i))
                          plt.show()

             icpTransformsErr = numpy.array(icpTransformsErr)

#            tindx = numpy.where(icpTransformsErr==icpTransformsErr.min())[0][0]
             tindx = numpy.where(icpTransformsErr==icpTransformsErr[:,-
1].min())[0][0]
             transform = icpTransforms[tindx] * initialTransforms[tindx]
             print 'Best iteration was %d with transform:' % (tindx)
             print transform

             if visualize:#display err plot of ICP
                    fig = plt.figure()
                    ax = fig.add_subplot(1,1,1)
                    ax.plot(icpT[tindx],icpTransformsErr[tindx],'x--')
                    ax.set_xlabel('Time')
                    ax.set_ylabel('RMS error')
                    ax.set_title('Result of ICP on %s Joint, initial transform %d'
%(jointOfInterest.name, tindx))
                    plt.show()

             if visualize and False:
                    #Display initial filtered points clouds that will be passed to
ICP
```

```
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                    #display point clouds at final position
                    modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
```

```
                  TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

      else:  #skip ICP, just use predetermined transform
            transform = numpy.matrix(fineAlignTransforms['Hindlimb Left Knee'])
#           transform = numpy.matrix(
#                        [[  5.65588121e-01,   7.25237403e-01,   3.92607675e-01,  -
2.81949076e+01],
#                         [ -2.21102136e-01,  -3.25288973e-01,   9.19402485e-01,
2.16992939e+01],
#                         [  7.94496018e-01,  -6.06809520e-01,  -2.36280401e-02,  -
2.50706200e+01],
#                         [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
1.00000000e+00]])

      #store transform
      fineAlignTransforms[jointOfInterest.name] = transform

      #Step 5: Apply ICP transform to joint
      R = numpy.matrix(numpy.identity(4))
      R[:3,:3] = transform[:3,:3]
      jointOfInterest.rotate(R, relative=True)
      cummulativeJointRotation = numpy.matrix(numpy.identity(4))
      cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
      modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
      #translation has to include rotation effects of cumulative transformations
#     jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
      jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

      if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
            kwargs['updateSceneFunc']()
            time.sleep(3)

      #print resulting joint location
      if 'getResults' in kwargs and kwargs['getResults']:
            atlasData.atlasJoint.transformVertices()
            print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
            meanDist = []
            for bone in bones:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  meanDist.append(numpy.sum(d)/len(i))
            print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
            dist = 0.0
            numPoints = 0.0
            for bone in scanData.bonesVol1:
```

```
                        d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                        dist += numpy.sum(d)
                        numPoints += len(i)
                print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)

#       #rotote femur and tibia so that tibia hinge joint lines up with secondary axis
#       atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels['Upper
Hindlimb Left']))
#       atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels['Lower
Hindlimb Left']))
#       sv = numpy.cross(femurLeft.proximodistalVecTransformed,
tibiaLeft.proximodistalVecTransformed)
#       tv = numpy.cross(tibiaLeft.proximodistalVecTransformed, sv)
#       coordAlignTform =
numpyTransform.coordinateSystemConversionMatrix([tibiaLeft.proximodistalVecTransforme
d, tibiaLeft.secondaryVecTransformed, tibiaLeft.tertiaryVecTransformed],
[tibiaLeft.proximodistalVecTransformed, sv, tv], N=4)
#       jointOfInterest.rotate(coordAlignTform, relative=True)

        #get indices of model vertices within 1mm of aligned atlas hip left vertices
        if 'tibiaLeftVertexShortcut' in kwargs and
os.path.exists(kwargs['tibiaLeftVertexShortcut']):
                vertexDict = loadmat(kwargs['tibiaLeftVertexShortcut'])
                tibiaLeftindx = vertexDict['indx'].astype(numpy.bool).squeeze()
        else:
                #create KDTree of newly aligned bone
                dataPoints = []
                for bone in bones:

        atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                        dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
                dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
                boneKDTree = KDTree(dataPoints)
                #find spine indices
                t1 = time.time()
                tibiaLeftindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
                NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
                for vindx in NN:
                        tibiaLeftindx[vindx] = True
                print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
                if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                        savemat(kwargs['savePrefix']+'tibiaLeftvertices.mat',
{'indx':tibiaLeftindx})

        ##########################################################################
##############################################################
        #align Right Tibia
        for joint in femurRight.childJoints:
                if joint.name == 'Hindlimb Right Knee':
```

```
                    tibiaRight = joint
                    break
        else:
                raise Exception('Correct Joint not found')
        jointOfInterest = tibiaRight
        cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
        bones =['Lower Hindlimb Right']
        if 'Hindlimb Right Knee' not in fineAlignTransforms:#Do ICP
                #step 1: get data vertex list. These vertexes are based on the current
best transform
                dataPoints = []
                for bone in bones:

        atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                    dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
                dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

                #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
                #create filtered model point cloud
                modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
                modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

                #remove front half of model data
                normV = scanData.alignmentAxis[0]
                planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
                pointVecs = modelPoints - planePoint
                distance = numpy.dot(pointVecs, normV)
                modelIndx[distance > 0] = False

                #remove previously aligned vertecies
                modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
                modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
                modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))
                modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(hipLeftindx))
                modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(hipRightindx))
                modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurLeftindx))
                modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurRightindx))
                modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(tibiaLeftindx))

                #Step 3: Reorient Point Clouds
                #transform points back so that joint of interest is in original
position/orientation
                #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
                #ICP must be done on original points not points midway through a
transformation
```

```
            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)

            #move point clouds so that joint is at origin
            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
            modelPoints -= jointLocation
            dataPoints -= jointLocation

            #Remove any points now that model is reoriented
            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #Get just points of interest
            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]

            #create different starting transformations
            cummJointTransferRotOnly = numpy.matrix(numpy.identity(4))
            cummJointTransferRotOnly[:3,:3] = cummulativeJointTransform[:3,:3]
            proximodistalVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.proximodist
alVecTransformed)
            secondaryVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.secondaryVe
cTransformed)
            initialTransforms = []
            initialTransformsError = []
            icpTransformsErr = []
            icpTransforms = []
            icpT = []

            for spin in numpy.arange(0, 2*numpy.pi, 3*numpy.pi):
                  for elevation in numpy.arange(0, numpy.pi, numpy.pi/4):
                        for aximuth in numpy.arange(0, 2*numpy.pi, numpy.pi/4):
                              initialTransforms.append(
numpyTransform.rotation(aximuth, proximodistalVecTransformed, N=4) *
numpyTransform.rotation(elevation, secondaryVecTransformed, N=4) *
numpyTransform.rotation(spin, proximodistalVecTransformed, N=4) )

            for i in xrange(len(initialTransforms)):
                  initialTransform = initialTransforms[i]
                  dataPointsInitialTrandform =
numpyTransform.transformPoints(initialTransform, dataPoints)

                  #limit translation
#                 transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor
* scanData.sliceThickness)
                  transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
                  print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
                  #Step 4: Perform actual ICP
                  print 'ITERATION',i
```

```
                    icp = ICP(modelPoints, dataPointsInitialTrandform,
maxIterations=15, modelDownsampleFactor=1, dataDownsampleFactor=1,
minimizeMethod='fmincon')
                    initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
1.0, 1.0])
                    lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -
transBound, -transBound, 0.8, 0.8, 0.8])
                    upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
                    transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
                    del icp
                    transform = transform[-1]
                    print 'ICP Generated Transform for %s Joint' %
(jointOfInterest.name)
                    print transform

                    if transform[0,3] > upperBounds[3] or transform[0,3] <
lowerBounds[3]:
                          print 'x translation incorrect'
                    if transform[1,3] > upperBounds[4] or transform[1,3] <
lowerBounds[4]:
                          print 'Y translation incorrect'
                    if transform[2,3] > upperBounds[5] or transform[2,3] <
lowerBounds[5]:
                          print 'Z translation incorrect'

                    icpT.append(t)
                    icpTransformsErr.append(err)
                    icpTransforms.append(transform)

                    if visualize and False:   #display err plot of ICP
                          fig = plt.figure()
                          ax = fig.add_subplot(1,1,1)
                          ax.plot(t,err,'x--')
                          ax.set_xlabel('Time')
                          ax.set_ylabel('RMS error')
                          ax.set_title('Result of ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, i))
                          plt.show()

             icpTransformsErr = numpy.array(icpTransformsErr)

#            tindx = numpy.where(icpTransformsErr==icpTransformsErr.min())[0][0]
             tindx = numpy.where(icpTransformsErr==icpTransformsErr[:,-
1].min())[0][0]
             transform = icpTransforms[tindx] * initialTransforms[tindx]
             print 'Best iteration was %d with transform:' % (tindx)
             print transform

             if visualize:#display err plot of ICP
                    fig = plt.figure()
                    ax = fig.add_subplot(1,1,1)
                    ax.plot(icpT[tindx],icpTransformsErr[tindx],'x--')
                    ax.set_xlabel('Time')
```

```
                    ax.set_ylabel('RMS error')
                    ax.set_title('Result of ICP on %s Joint, initial transform %d'
%(jointOfInterest.name, tindx))
                    plt.show()

             if visualize and False:
                    #Display initial filtered points clouds that will be passed to
ICP
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                    #display point clouds at final position
                    modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])
```

```
                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

      else:  #skip ICP, just use predetermined transform
            transform = numpy.matrix(fineAlignTransforms['Hindlimb Right Knee'])
#           transform = numpy.matrix(
#                 [[  0.61328114,  -0.74659998,  -0.2578269,   25.08829485],
#                  [  0.23768732,  -0.13684834,   0.9616534,    9.49064482],
#                  [ -0.7532536,   -0.65104608,   0.09353084, -26.06480535],
#                  [  0.,           0.,           0.,           1.        ]])

      #store transform
      fineAlignTransforms[jointOfInterest.name] = transform

      #Step 5: Apply ICP transform to joint
      R = numpy.matrix(numpy.identity(4))
      R[:3,:3] = transform[:3,:3]
      jointOfInterest.rotate(R, relative=True)
      cummulativeJointRotation = numpy.matrix(numpy.identity(4))
      cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
      modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
      #translation has to include rotation effects of cumulative transformations
#     jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
      jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

      if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
            kwargs['updateSceneFunc']()
            time.sleep(3)

      #print resulting joint location
      if 'getResults' in kwargs and kwargs['getResults']:
            atlasData.atlasJoint.transformVertices()
            print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
            meanDist = []
            for bone in bones:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  meanDist.append(numpy.sum(d)/len(i))
            print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
            dist = 0.0
            numPoints = 0.0
```

```
            for bone in scanData.bonesVol1:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  dist += numpy.sum(d)
                  numPoints += len(i)
            print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)

#     #rotote femur and tibia so that tibia hinge joint lines up with secondary axis
#     atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels['Upper
Hindlimb Right']))
#     atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels['Lower
Hindlimb Right']))
#     sv = numpy.cross(femurRight.proximodistalVecTransformed,
tibiaRight.proximodistalVecTransformed)
#     tv = numpy.cross(tibiaRight.proximodistalVecTransformed, sv)
#     coordAlignTform =
numpyTransform.coordinateSystemConversionMatrix([tibiaRight.proximodistalVecTransform
ed, tibiaRight.secondaryVecTransformed, tibiaRight.tertiaryVecTransformed],
[tibiaRight.proximodistalVecTransformed, sv, tv], N=4)
#     jointOfInterest.rotate(coordAlignTform, relative=True)

      #get indices of model vertices within 1mm of aligned atlas hip left vertices
      if 'tibiaRightVertexShortcut' in kwargs and
os.path.exists(kwargs['tibiaRightVertexShortcut']):
            vertexDict = loadmat(kwargs['tibiaRightVertexShortcut'])
            tibiaRightindx = vertexDict['indx'].astype(numpy.bool).squeeze()
      else:
            #create KDTree of newly aligned bone
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
            boneKDTree = KDTree(dataPoints)
            #find spine indices
            t1 = time.time()
            tibiaRightindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
            NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
            for vindx in NN:
                  tibiaRightindx[vindx] = True
            print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
            if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                  savemat(kwargs['savePrefix']+'tibiaRightvertices.mat',
{'indx':tibiaRightindx})

      ##########################################################################
##############################################################
      #align Lower Left Paw
      for joint in tibiaLeft.childJoints:
```

```
            if joint.name == 'Hindlimb Left Ankle':
                  ankleLeft = joint
                  break
      else:
            raise Exception('Correct Joint not found')
      jointOfInterest = ankleLeft
      cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
      bones =['HindPaw Left']
      if 'Hindlimb Left Ankle' not in fineAlignTransforms: #Do ICP
            #step 1: get data vertex list. These vertexes are based on the current
best transform
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

            #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
            #create filtered model point cloud
            modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
            modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #remove front half of model data
            normV = scanData.alignmentAxis[0]
            planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
            pointVecs = modelPoints - planePoint
            distance = numpy.dot(pointVecs, normV)
            modelIndx[distance > 0] = False

            #remove previously aligned vertecies
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(hipLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(hipRightindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurRightindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(tibiaLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(tibiaRightindx))

            #Step 3: Reorient Point Clouds
            #transform points back so that joint of interest is in original
position/orientation
```

```
            #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
            #ICP must be done on original points not points midway through a
transformation
            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)

            #move point clouds so that joint is at origin
            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
            modelPoints -= jointLocation
            dataPoints -= jointLocation

            #Remove any points now that model is reoriented
            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #Get just points of interest
            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]

            #create different starting transformations
            cummJointTransferRotOnly = numpy.matrix(numpy.identity(4))
            cummJointTransferRotOnly[:3,:3] = cummulativeJointTransform[:3,:3]
            proximodistalVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.proximodist
alVecTransformed)
            secondaryVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.secondaryVe
cTransformed)
            initialTransforms = []
            initialTransformsError = []
            icpTransformsErr = []
            icpTransforms = []
            icpT = []

            for spin in numpy.arange(0, 2*numpy.pi, 3*numpy.pi):
                  for elevation in numpy.arange(0, numpy.pi, numpy.pi/2):
                        for aximuth in numpy.arange(0, 2*numpy.pi, numpy.pi):
                              initialTransforms.append(
numpyTransform.rotation(aximuth, proximodistalVecTransformed, N=4) *
numpyTransform.rotation(elevation, secondaryVecTransformed, N=4) *
numpyTransform.rotation(spin, proximodistalVecTransformed, N=4) )

            for i in xrange(len(initialTransforms)):
                  initialTransform = initialTransforms[i]
                  dataPointsInitialTrandform =
numpyTransform.transformPoints(initialTransform, dataPoints)

                  #limit translation
#                 transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor
* scanData.sliceThickness)
                  transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
                  print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
```

```
                    #Step 4: Perform actual ICP
                    print 'ITERATION',i
                    icp = ICP(modelPoints, dataPointsInitialTrandform,
maxIterations=15, modelDownsampleFactor=1, dataDownsampleFactor=1,
minimizeMethod='fmincon')
                    initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
1.0, 1.0])
                    lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -
transBound, -transBound, 0.8, 0.8, 0.8])
                    upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
                    transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
                    del icp
                    transform = transform[-1]
                    print 'ICP Generated Transform for %s Joint' %
(jointOfInterest.name)
                    print transform

                    if transform[0,3] > upperBounds[3] or transform[0,3] <
lowerBounds[3]:
                          print 'x translation incorrect'
                    if transform[1,3] > upperBounds[4] or transform[1,3] <
lowerBounds[4]:
                          print 'Y translation incorrect'
                    if transform[2,3] > upperBounds[5] or transform[2,3] <
lowerBounds[5]:
                          print 'Z translation incorrect'

                    icpT.append(t)
                    icpTransformsErr.append(err)
                    icpTransforms.append(transform)

                    if visualize and False:   #display err plot of ICP
                          fig = plt.figure()
                          ax = fig.add_subplot(1,1,1)
                          ax.plot(t,err,'x--')
                          ax.set_xlabel('Time')
                          ax.set_ylabel('RMS error')
                          ax.set_title('Result of ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, i))
                          plt.show()

             icpTransformsErr = numpy.array(icpTransformsErr)

             tindx = numpy.where(icpTransformsErr==icpTransformsErr[:,-
1].min())[0][0]
             transform = icpTransforms[tindx] * initialTransforms[tindx]
             print 'Best iteration was %d with transform:' % (tindx)
             print transform

             if visualize:#display err plot of ICP
                    fig = plt.figure()
                    ax = fig.add_subplot(1,1,1)
                    ax.plot(icpT[tindx],icpTransformsErr[tindx],'x--')
```

```
                    ax.set_xlabel('Time')
                    ax.set_ylabel('RMS error')
                    ax.set_title('Result of ICP on %s Joint, initial transform %d'
%(jointOfInterest.name, tindx))
                    plt.show()

             if visualize and False:
                    #Display initial filtered points clouds that will be passed to
ICP
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                    #display point clouds at final position
                    modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])
```

```
                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

      else:  #skip ICP, just use predetermined transform
            transform = numpy.matrix(fineAlignTransforms['Hindlimb Left Ankle'])
#           transform = numpy.matrix(
#                 [[  0.8903688,    0.0683342,   -0.45008203,   7.15709156],
#                  [  0.11739263,   0.92076859,   0.37202711,  18.74429597],
#                  [  0.43984357,  -0.38407764,   0.8118017,   10.02141994],
#                  [  0.,           0.,           0.,           1.        ]])

      #store transform
      fineAlignTransforms[jointOfInterest.name] = transform

      #Step 5: Apply ICP transform to joint
      R = numpy.matrix(numpy.identity(4))
      R[:3,:3] = transform[:3,:3]
      jointOfInterest.rotate(R, relative=True)
      cummulativeJointRotation = numpy.matrix(numpy.identity(4))
      cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
      modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
      #translation has to include rotation effects of cumulative transformations
#     jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
      jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

      if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
            kwargs['updateSceneFunc']()
            time.sleep(3)

      #print resulting joint location
      if 'getResults' in kwargs and kwargs['getResults']:
            atlasData.atlasJoint.transformVertices()
            print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
            meanDist = []
            for bone in bones:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  meanDist.append(numpy.sum(d)/len(i))
            print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
            dist = 0.0
```

```
            numPoints = 0.0
            for bone in scanData.bonesVol1:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  dist += numpy.sum(d)
                  numPoints += len(i)
            print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)

      #get indices of model vertices within 1mm of aligned atlas hip left vertices
      if 'lowerLeftPawVertexShortcut' in kwargs and
os.path.exists(kwargs['lowerLeftPawVertexShortcut']):
            vertexDict = loadmat(kwargs['lowerLeftPawVertexShortcut'])
            lowerLeftPawindx = vertexDict['indx'].astype(numpy.bool).squeeze()
      else:
            #create KDTree of newly aligned bone
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
            boneKDTree = KDTree(dataPoints)
            #find spine indices
            t1 = time.time()
            lowerLeftPawindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
            NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
            for vindx in NN:
                  lowerLeftPawindx[vindx] = True
            print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
            if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                  savemat(kwargs['savePrefix']+'lowerLeftPawvertices.mat',
{'indx':lowerLeftPawindx})

      ##########################################################################
##############################################################
      #align Lower Right Paw
      for joint in tibiaRight.childJoints:
            if joint.name == 'Hindlimb Right Ankle':
                  ankleRight = joint
                  break
      else:
            raise Exception('Correct Joint not found')
      jointOfInterest = ankleRight
      cummulativeJointTransform =
atlasData.atlasJoint.getCummulativeTransform(id(jointOfInterest))
      bones =['HindPaw Right']
      if 'Hindlimb Right Ankle' not in fineAlignTransforms:      #Do ICP
            #step 1: get data vertex list. These vertexes are based on the current
best transform
            dataPoints = []
```

```
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )

            #step 2: Filter model points as much as possible, ideally only bones to
be matched will remain
            #create filtered model point cloud
            modelPoints = scanData.isosurfaceModel1.transformedVertexList.copy()
            modelIndx = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #remove front half of model data
            normV = scanData.alignmentAxis[0]
            planePoint = (scanData.isosurfaceModel.maxPointTransformed +
scanData.isosurfaceModel.minPointTransformed) / 2.0
            pointVecs = modelPoints - planePoint
            distance = numpy.dot(pointVecs, normV)
            modelIndx[distance > 0] = False

            #remove previously aligned vertecies
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(spineindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(tailindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(skullindx))
            modelIndx = numpy.logical_and(modelIndx, numpy.logical_not(hipLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(hipRightindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(femurRightindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(tibiaLeftindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(tibiaRightindx))
            modelIndx = numpy.logical_and(modelIndx,
numpy.logical_not(lowerLeftPawindx))

            #Step 3: Reorient Point Clouds
            #transform points back so that joint of interest is in original
position/orientation
            #the reason for this is that P=C*V then P = M*P, is not equal to P =
(C*M)*V. Where C is parent transform, M is new ICP transform, V is original vector
            #ICP must be done on original points not points midway through a
transformation
            modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,modelPoints)
            dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform.I,dataPoints)

            #move point clouds so that joint is at origin
            jointLocation =
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
```

```
            modelPoints -= jointLocation
            dataPoints -= jointLocation

            #Remove any points now that model is reoriented
            modelIndx2 = numpy.ones(modelPoints.shape[0], dtype=numpy.bool)

            #Get just points of interest
            modelPoints = modelPoints[numpy.logical_and(modelIndx, modelIndx2)]

            #create different starting transformations
            cummJointTransferRotOnly = numpy.matrix(numpy.identity(4))
            cummJointTransferRotOnly[:3,:3] = cummulativeJointTransform[:3,:3]
            proximodistalVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.proximodist
alVecTransformed)
            secondaryVecTransformed =
numpyTransform.transformPoints(cummJointTransferRotOnly.I,jointOfInterest.secondaryVe
cTransformed)
            initialTransforms = []
            initialTransformsError = []
            icpTransformsErr = []
            icpTransforms = []
            icpT = []

            for spin in numpy.arange(0, 2*numpy.pi, 3*numpy.pi):
                  for elevation in numpy.arange(0, numpy.pi, numpy.pi/2):
                        for aximuth in numpy.arange(0, 2*numpy.pi, numpy.pi):
                              initialTransforms.append(
numpyTransform.rotation(aximuth, proximodistalVecTransformed, N=4) *
numpyTransform.rotation(elevation, secondaryVecTransformed, N=4) *
numpyTransform.rotation(spin, proximodistalVecTransformed, N=4) )

            for i in xrange(len(initialTransforms)):
                  initialTransform = initialTransforms[i]
                  dataPointsInitialTrandform =
numpyTransform.transformPoints(initialTransform, dataPoints)

                  #limit translation
#                 transBound = jointOfInterest.DOFtrans / (scanData.resampleFactor
* scanData.sliceThickness)
                  transBound = jointOfInterest.DOFtrans / scanData.sliceThickness
                  print '%s translation limited to %fmm aka %f pixels'%
(jointOfInterest.name, jointOfInterest.DOFtrans, transBound)
                  #Step 4: Perform actual ICP
                  print 'ITERATION',i
                  icp = ICP(modelPoints, dataPointsInitialTrandform,
maxIterations=15, modelDownsampleFactor=1, dataDownsampleFactor=1,
minimizeMethod='fmincon')
                  initialGuess = numpy.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
1.0, 1.0])
                  lowerBounds = numpy.array([-pi, -pi, -pi, -transBound, -
transBound, -transBound, 0.8, 0.8, 0.8])
                  upperBounds = numpy.array([pi, pi, pi, transBound, transBound,
transBound, 1.2, 1.2, 1.2])
```

```
                    transform, err, t = icp.runICP(x0=initialGuess,
lb=lowerBounds,ub=upperBounds)
                    del icp
                    transform = transform[-1]
                    print 'ICP Generated Transform for %s Joint' %
(jointOfInterest.name)
                    print transform

                    if transform[0,3] > upperBounds[3] or transform[0,3] <
lowerBounds[3]:
                          print 'x translation incorrect'
                    if transform[1,3] > upperBounds[4] or transform[1,3] <
lowerBounds[4]:
                          print 'Y translation incorrect'
                    if transform[2,3] > upperBounds[5] or transform[2,3] <
lowerBounds[5]:
                          print 'Z translation incorrect'

                    icpT.append(t)
                    icpTransformsErr.append(err)
                    icpTransforms.append(transform)

                    #check to see if end error is worse then start error, if so then

                    if visualize and False:   #display err plot of ICP
                          fig = plt.figure()
                          ax = fig.add_subplot(1,1,1)
                          ax.plot(t,err,'x--')
                          ax.set_xlabel('Time')
                          ax.set_ylabel('RMS error')
                          ax.set_title('Result of ICP on %s Joint, initial transform
%d' %(jointOfInterest.name, i))
                          plt.show()

             icpTransformsErr = numpy.array(icpTransformsErr)

#            tindx = numpy.where(icpTransformsErr==icpTransformsErr.min())[0][0]
             tindx = numpy.where(icpTransformsErr==icpTransformsErr[:,-
1].min())[0][0]
             transform = icpTransforms[tindx] * initialTransforms[tindx]
             print 'Best iteration was %d with transform:' % (tindx)
             print transform

             if visualize:#display err plot of ICP
                    fig = plt.figure()
                    ax = fig.add_subplot(1,1,1)
                    ax.plot(icpT[tindx],icpTransformsErr[tindx],'x--')
                    ax.set_xlabel('Time')
                    ax.set_ylabel('RMS error')
                    ax.set_title('Result of ICP on %s Joint, initial transform %d'
%(jointOfInterest.name, tindx))
                    plt.show()

             if visualize and False:
```

```
                    #Display initial filtered points clouds that will be passed to
ICP
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Initial Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed = numpyTransform.transformPoints(transform,
dataPoints)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Initial Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

                    #display point clouds at final position
                    modelPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    modelPoints =
numpyTransform.transformPoints(cummulativeJointTransform, modelPoints)
                    tri = numpy.array(range(modelPoints.shape[0]-
modelPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(modelPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Model Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[1.0,0.0,0.0])

                    dataPoints +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPoints =
numpyTransform.transformPoints(cummulativeJointTransform, dataPoints)
                    tri = numpy.array(range(dataPoints.shape[0]-
dataPoints.shape[0]%3))
                    tri = tri.reshape((tri.shape[0]/3,3))
                    TriModel(dataPoints, tri, scanData.isosurfaceJoint.parentJoint,
name='Final Filtered Data Point Cloud for %s Joint' % (jointOfInterest.name),
displayAsPoints=True, color=[0.0,1.0,0.0])

                    dataPointsTransformed +=
jointOfInterest.initialLocationMat[:3,3].getA().squeeze()
                    dataPointsTransformed =
numpyTransform.transformPoints(cummulativeJointTransform, dataPointsTransformed)
                    tri = numpy.array(range(dataPointsTransformed.shape[0]-
dataPointsTransformed.shape[0]%3))
```

```
                  tri = tri.reshape((tri.shape[0]/3,3))
                  TriModel(dataPointsTransformed, tri,
scanData.isosurfaceJoint.parentJoint, name='Final Translated Filtered Data Point
Cloud for %s Joint' % (jointOfInterest.name), displayAsPoints=True,
color=[0.0,0.0,1.0])

      else:  #skip ICP, just use predetermined transform
            transform = numpy.matrix(fineAlignTransforms['Hindlimb Right Ankle'])
#           transform = numpy.matrix(
#                 [[  0.42756724,   0.53466485,   0.72891683, -28.76283195],
#                  [ -0.88905172,   0.39466056,   0.23201311, -13.92154962],
#                  [ -0.16362547,  -0.74724597,   0.64408863,  22.98018009],
#                  [  0.,           0.,           0.,           1.        ]])

      #store transform
      fineAlignTransforms[jointOfInterest.name] = transform

      #Step 5: Apply ICP transform to joint
      R = numpy.matrix(numpy.identity(4))
      R[:3,:3] = transform[:3,:3]
      jointOfInterest.rotate(R, relative=True)
      cummulativeJointRotation = numpy.matrix(numpy.identity(4))
      cummulativeJointRotation[:3,:3] = cummulativeJointTransform[:3,:3]
      modifiedTranslate =
cummulativeJointRotation*numpyTransform.translation(transform[:3,3].getA().squeeze())
      #translation has to include rotation effects of cumulative transformations
#     jointOfInterest.translate(modifiedTranslate[:3,3].getA().squeeze(),
absolute=False)
      jointOfInterest.translate(transform[:3,3].getA().squeeze(), absolute=False)

      if 'updateSceneFunc' in kwargs and kwargs['updateSceneFunc'] is not None:
            kwargs['updateSceneFunc']()
            time.sleep(3)

      #print resulting joint location
      if 'getResults' in kwargs and kwargs['getResults']:
            atlasData.atlasJoint.transformVertices()
            print '%s Joint Location: %s' % (jointOfInterest.name,
str(jointOfInterest.location))
            meanDist = []
            for bone in bones:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  meanDist.append(numpy.sum(d)/len(i))
            print '%s Joint Mean surface to bone distance: %f' %
(jointOfInterest.name, numpy.mean(meanDist))
            dist = 0.0
            numPoints = 0.0
            for bone in scanData.bonesVol1:
                  d, i =
scanData.isosurfaceModel1.kdtree.query(atlasData.atlasModels[bone].transformedVertexL
ist)
                  dist += numpy.sum(d)
                  numPoints += len(i)
```

```
            print 'Post %s Align Mean distance all bones to surface: %f' %
(jointOfInterest.name, dist/numPoints)

      #get indices of model vertices within 1mm of aligned atlas hip left vertices
      if 'lowerRightPawVertexShortcut' in kwargs and
os.path.exists(kwargs['lowerRightPawVertexShortcut']):
            vertexDict = loadmat(kwargs['lowerRightPawVertexShortcut'])
            lowerRightPawindx = vertexDict['indx'].astype(numpy.bool).squeeze()
      else:
            #create KDTree of newly aligned bone
            dataPoints = []
            for bone in bones:

      atlasData.atlasJoint.transformVertices(modelID=id(atlasData.atlasModels[bone])
)
                  dataPoints = numpy.append(dataPoints,
atlasData.atlasModels[bone].transformedVertexList)
            dataPoints = dataPoints.reshape( (dataPoints.shape[0]/3,3) )
            boneKDTree = KDTree(dataPoints)
            #find spine indices
            t1 = time.time()
            lowerRightPawindx = numpy.zeros(iso1KDTree.n, dtype=numpy.bool)
            NN = boneKDTree.query_ball_tree(iso1KDTree, 1.0/scanData.sliceThickness)
            for vindx in NN:
                  lowerRightPawindx[vindx] = True
            print 'Took %f seconds to calculate %s vertices' % (time.time() - t1,
bone)
            if 'savePrefix' in kwargs and kwargs['savePrefix'] is not None:
                  savemat(kwargs['savePrefix']+'lowerRightPawvertices.mat',
{'indx':lowerRightPawindx})


      #save transforms to file
      if  'savePrefix' in kwargs:
            savemat(kwargs['savePrefix']+'FineAlignTransforms.mat',
fineAlignTransforms)

def testGetSpine():
      data = loadmat('spineSeg.mat')
      getSpine(data['largestVolLabel'], data['neckPosAligned'])

if __name__ == '__main__':
#     alignmentAxis = RoughAlignTest(verbose=True, visualize=True)
#     print 'Rough Align Results'
#     print 'Alignment Axis:'
#     print alignmentAxis[0]
#     print 'Neck Position:'
#     print alignmentAxis[1]

#     savedData=loadmat('Saved Data.mat')
#     affineMatrixTest(savedData['referenceVolume'], savedData['axes'],
alignedVolumeGoal=savedData['alignedCTVolume'])

#     roughAlignProfile()
```

```
testGetSpine()
```

mlabMin.m

```
function [R, T, S] = mlabMin()
global modelPoints dataPoints x0 lb ub
R = [1.0 0.0 0.0;
     0.0 1.0 0.0;
     0.0 0.0 1.0];
T = [0.0; 0.0; 0.0];

A = [];
b = [];
Aeq = [];
beq = [];
nonlcon = [];

% options=optimset('Algorithm','trust-region-reflective');
options=optimset('Algorithm','active-set');
% options=optimset('Algorithm','interior-point');
% options=optimset('Algorithm','sqp')
[x,fval] = fmincon(@objectiveFunc, x0, A, b, Aeq, beq, lb, ub, nonlcon,
options);

rxOut=x(1);
ryOut=x(2);
rzOut=x(3);

RxOut = [1 0 0 0;
      0 cos(rxOut) -sin(rxOut) 0;
      0 sin(rxOut) cos(rxOut) 0;
        0 0 0 1];

RyOut = [cos(ryOut) 0 sin(ryOut) 0;
      0 1 0 0;
      -sin(ryOut) 0 cos(ryOut) 0;
        0 0 0 1];

RzOut = [cos(rzOut) -sin(rzOut) 0 0;
      sin(rzOut) cos(rzOut) 0 0;
      0 0 1 0;
        0 0 0 1];

% Rotation matrix
% R = RxOut*RyOut*RzOut;
R = RyOut*RzOut*RxOut;

if length(x) >= 6
T = [ 1 0 0 x(4);
        0 1 0 x(5);
        0 0 1 x(6);
        0 0 0 1];
else
T = [ 1 0 0 0;
        0 1 0 0;
        0 0 1 0;
        0 0 0 1];
end
```

```
if length(x) >= 9
S = [ x(7) 0 0 0;
            0 x(8) 0 0;
            0 0 x(9) 0;
            0 0 0 1];
else
S = [ 1 0 0 0;
            0 1 0 0;
            0 0 1 0;
            0 0 0 1];
end


    function err = objectiveFunc( inputargs )
%         global modelPoints dataPoints
        rx=inputargs(1);
        ry=inputargs(2);
        rz=inputargs(3);

        Rx = [1 0 0 0;
              0 cos(rx) -sin(rx) 0;
              0 sin(rx) cos(rx) 0;
              0 0 0 1];

        Ry = [cos(ry) 0 sin(ry) 0;
              0 1 0 0;
              -sin(ry) 0 cos(ry) 0;
              0 0 0 1];

        Rz = [cos(rz) -sin(rz) 0 0;
              sin(rz) cos(rz) 0 0;
              0 0 1 0;
              0 0 0 1];

        % Rotation matrix
        % Rr = Rx*Ry*Rz;
            Rr = Ry*Rz*Rx;

            if length(inputargs) >= 9
                  sx = inputargs(7);
                  sy = inputargs(8);
                  sz = inputargs(9);
                  Ss = [      sx 0 0 0;
                              0 sy 0 0;
                              0 0 sz 0;
                              0 0 0 1];
            else
                  Ss = [      1 0 0 0;
                              0 1 0 0;
                              0 0 1 0;
                              0 0 0 1];
            end

            if length(inputargs) >= 6
                tx=inputargs(4);
                  ty=inputargs(5);
                  tz=inputargs(6);
```

```
                Tt = [1 0 0 tx;
                      0 1 0 ty;
                      0 0 1 tz;
                      0 0 0 1];
            else
                Tt = [1 0 0 0;
                      0 1 0 0;
                      0 0 1 0;
                      0 0 0 1];
            end

        % Transform data-matrix plus noise into model-matrix
        data = ones(4, size(dataPoints, 2));
        data(1:3,:) = dataPoints;
        M = Tt * Rr * Ss;
        DataICP = M * data;

        err = rms_error(DataICP(1:3,:), modelPoints);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Determine the RMS error between two point equally sized point clouds with
% point correspondance.
% ER = rms_error(p1,p2) where p1 and p2 are 3xn matrices.

function ER = rms_error(p1,p2)
dsq = sum(power(p1 - p2, 2),1);
ER = sqrt(mean(dsq));
end
```

mlabMinScale.m

```
function [R, T, S] = mlabMinScale()
global modelPoints dataPoints x0 lb ub
R = [ 1 0 0 0;
      0 1 0 0;
      0 0 1 0;
      0 0 0 1];
T = [ 1 0 0 0;
      0 1 0 0;
      0 0 1 0;
      0 0 0 1];

A = [];
b = [];
Aeq = [];
beq = [];
nonlcon = [];

% options=optimset('Algorithm','trust-region-reflective');
options=optimset('Algorithm','active-set');
% options=optimset('Algorithm','interior-point');
% options=optimset('Algorithm','sqp')
[x,fval] = fmincon(@objectiveFunc, x0, A, b, Aeq, beq, lb, ub, nonlcon,
options);

S = [ x(1) 0 0 0;
          0 x(2) 0 0;
          0 0 x(3) 0;
          0 0 0 1];

    function err = objectiveFunc( inputargs )
          sx = inputargs(1);
          sy = inputargs(2);
          sz = inputargs(3);
          Ss = [      sx 0 0 0;
                      0 sy 0 0;
                      0 0 sz 0;
                      0 0 0 1];

        % Transform data-matrix plus noise into model-matrix
        data = ones(4, size(dataPoints, 2));
        data(1:3,:) = dataPoints;
        DataICP = Ss * data;

        err = rms_error(DataICP(1:3,:), modelPoints);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Determine the RMS error between two point equally sized point clouds with
% point correspondance.
% ER = rms_error(p1,p2) where p1 and p2 are 3xn matrices.

function ER = rms_error(p1,p2)
dsq = sum(power(p1 - p2, 2),1);
```

```
ER = sqrt(mean(dsq));
end
```

mlabMinScaleIso.m

```
function [R, T, S] = mlabMinScaleIso()
global modelPoints dataPoints x0 lb ub
R = [ 1 0 0 0;
      0 1 0 0;
      0 0 1 0;
      0 0 0 1];
T = [ 1 0 0 0;
      0 1 0 0;
      0 0 1 0;
      0 0 0 1];

x = fminbnd(@objectiveFunc,lb(1),ub(1))

S = [ x(1) 0 0 0;
           0 x(1) 0 0;
           0 0 x(1) 0;
           0 0 0 1];

    function err = objectiveFunc( inputargs )
           siso = inputargs(1);
           Ss = [      siso 0 0 0;
                       0 siso 0 0;
                       0 0 siso 0;
                       0 0 0 1];

        % Transform data-matrix plus noise into model-matrix
        data = ones(4, size(dataPoints, 2));
        data(1:3,:) = dataPoints;
        DataICP = Ss * data;

        err = rms_error(DataICP(1:3,:), modelPoints);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Determine the RMS error between two point equally sized point clouds with
% point correspondance.
% ER = rms_error(p1,p2) where p1 and p2 are 3xn matrices.

function ER = rms_error(p1,p2)
dsq = sum(power(p1 - p2, 2),1);
ER = sqrt(mean(dsq));
end
```