3D Human Pose Reconstruction from a 2D Monocular Image with Joint Ordering Uncertainty

Cedric Flamant

December 12, 2018

Introduction

Human pose estimation is important in machine vision due to its pervasive applications including human-computer interaction, intruder detection, security, assisted therapy, modeling historical footage, social behavior tracking, and self-driving cars. Artificial intelligence of the future will need to have a robust spatial awareness of surrounding humans, especially when a machine's actions could affect the safety of bystanders. For example, automated transport will be pressured to have layers of sensory redundancy, so the ability to extract the greatest amount of information from any single data stream would be invaluable. Three dimensional human pose reconstruction is already routinely performed in cinematography and in video game development for the purposes of aiding CGI and realistic human motion capture, but this is typically done with the aid of trackers and multiple view perspectives. Accurate inference of human pose from a monocular (single-view) camera is the one of the ultimate goals for 3D human analysis since it only requires a single localized sensor and does not require the subject to be specially equipped. Furthermore, the ubiquity of digital cameras makes such an approach readily applicable and cost-effective.

Hsi-Jian Lee and Zen Chen pioneered the monocular body posture field, proposing that knowledge of the human body's proportions, joint positions in the image, and their relative ordering in the z-direction could be used to determine the majority of pose information[1]. Taylor[2] generalized the approach to articulated objects and laid out the mathematical steps to unprojecting 2D joint locations back to their 3D sources used by this project and others[3]. Taylor's approach requires human intervention to pinpoint joint locations on the 2D image and indicate the relative z ordering of joints on each side of a limb. However, in recent years this task has been accomplished with machine learning techniques so that the process can be fully automated[4]. The pose reconstruction problem is often separated in this way, using neural networks to determine 2D features which can then be used to infer 3D information. In this project, following Taylor, I focus on the second part of the problem, assuming that certain 2D features are already identified through some means.

Taylor's method requires information about the z ordering of the joints on both sides of a given limb. In other words, it needs to know which side is closer as there are two allowed configurations for a limb with fixed length and 2D projection. When applying Taylor's approach to actual images, one finds that there are sometimes ambiguities to this ordering when limbs are nearly perpendicular to the z direction, *i.e.* orthogonal to the line of sight. These configurations are a fairly common occurrence since most pictures are taken of upright humans. When these ambiguities are present, they frequently introduce large errors in the results of Taylor's 3D reconstruction algorithm, as I will demonstrate. Additionally, it is challenging, even for a human, to accurately identify joint positions in an image, especially when the subject in the image is wearing thick clothing or has a bulky frame. In an attempt to address both of these issues in Taylor's approach, I had the idea of using the presence of these in-plane limbs as an optimization goal in the 3D reconstruction. Instead of just indicating joint locations as in Taylor's method, a user would also include the uncertainty in the joint positioning, which specifies the range over which constrained optimization can adjust the joint position to optimize the orthogonality of selected limbs to the z axis. This modification exhibits significant improvement over the original method, both quantitatively and qualitatively as I will show using simulated and real 3D poses.

Methods

The basis of this projects uses the foundations established by Taylor, which I will briefly summarize here. For use in later comparison to my method, I implemented Taylor's reconstruction algorithm in my setup.

Taylor's Method for Reconstruction of Articulated Objects from Point Correspondences

In general, it is not possible to unproject a 2D image of an unknown 3D image since projection is a non-invertible operation. However, the idea behind Taylor's approach is to use the known shape of a human to reduce the number of unknowns. Specifically, it is assumed that the human subject has limbs of known relative proportions and connectivity. As the body proportions of humans are all quite similar, we can use measurements of an average human as reference for arbitrary human subjects. Taylor uses the values shown in Table 1, which we also assume in this project.

Relative lengths of the segments in the human figure

Segment	Relative Length
Forearm	14
Upper Arm	15
Shoulder Girdle	18
Foreleg	20
Thigh	19
Pelvic Girdle	14
Spine	24
Height	70

Table 1: Values used in Taylor[2] to represent the limb proportions of a typical human.

The stick figure that Taylor uses to represent a human is shown in Figure 1. The method is easily extended to more intricate structures of rods and joints, but it would require more inputs for the additional joint locations.

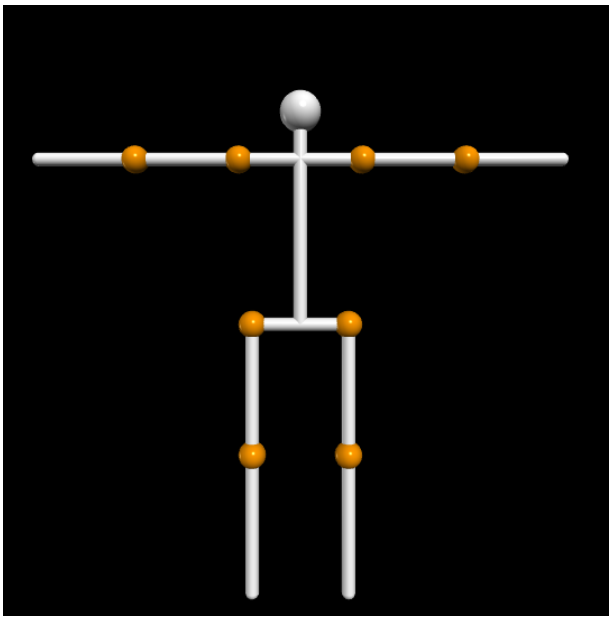


Figure 1: Stick figure to represent a human in pose reconstruction.

Taylor also assumes that point correspondences between a scene and its image can be modeled as a scaled

orthographic projection. In this model, the image coordinates (u, v) are related to the scene coordinates (x, y, z) by

$$\begin{pmatrix} u \\ v \end{pmatrix} = s \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}. \tag{1}$$

This model holds well in pictures with minimal perspective distortion, where the z extent of the subject is small compared to the distance to the camera. By comparing to a more realistic perspective projection model which accounts for the focal length of a simulated camera, Taylor showed that the scaled orthographic projection is sufficiently accurate for most photos of humans. Specifically, he showed that one can expect errors less than 5% when the camera distance is beyond 5 times the z-extent of the subject.

The stick figure has 25 independent degrees of freedom, 4 for each limb, 8 for the torso, and an unknown scale factor s. Comparing this to the 24 independent image measurements of the 2D joint locations of the right shoulder, left shoulder, right hip, left hip, right elbow, left elbow, right knee, left knee, right wrist, left wrist, right ankle, and left ankle, we will expect multiple solutions. Taylor chooses the free parameter in the reconstruction to be the scale s and uses geometrical considerations from the scaled orthographic projection to write the reconstruction as a function of s.

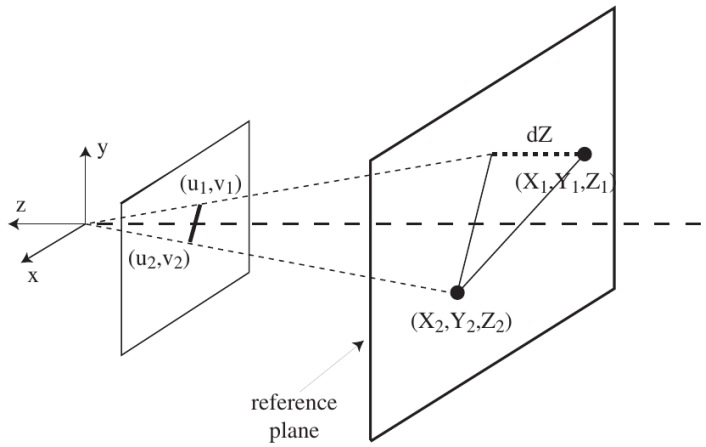


Figure 2: Figure from Taylor's paper[2] showing the projection of a line segment onto an image under scaled orthographic projection.

In Figure 2 we depict scaled orthographic projection of a limb of fixed length l. Knowing the scale s, it is a simple geometric matter to compute the allowed relative depth of the two endpoints dZ:

$$l^2 = (X_1 - X_2)^2 + (Y_1 - Y_2)^2 + (Z_1 - Z_2)^2 \tag{2}$$

$$(u_1 - u_2) = s(X_1 - X_2) \tag{3}$$

$$(v_1 - v_2) = s(Y_1 - Y_2) \tag{4}$$

$$dZ = (Z_1 - Z_2) \tag{5}$$

$$\Rightarrow dZ = \pm\sqrt{l^2 - \frac{(u_1 - u_2)^2 + (v_1 - v_2)^2}{s^2}}. \tag{6}$$

Notice that this equation for dZ constrains the allowable scale s since dZ cannot be complex. Thus, the quantity under the square root must be non-negative, giving the restriction

$$s \geq \frac{\sqrt{(u_1 - u_2)^2 + (v_1 - v_2)^2}}{l}. \tag{7}$$

Such a restriction exists for every limb in the stick figure, so the overall scale is has to be at least as large as the largest of the constraints.

From equation 6 we see that the scale determines how slanted a line segment is towards the camera. The larger the scale, the closer the subject is to the image plane. From these considerations, we can develop an intuitive understanding of the scale constraint in equation 7. A line segment with fixed endpoint projection (u_1, v_1),(u_2, v_2) on the image plane has to become increasingly upright as it is moved away from the camera (hence decreasing scale s). However, at one point it will be perpendicular to the z direction, *i.e.* it will be flush with the reference plane. It would no longer be possible to move the segment back further without changing the projection on the image plane, hence setting a minimum scale s corresponding to this maximum distance.

As mentioned in the introduction, in most pictures of humans there will be at least one limb nearly perpendicular to the line of sight. This means that simply using the minimum allowable scale in the 3D reconstruction will give reasonable results for the majority of cases.

Note that in equation 6 we do not know the sign of the Z difference in segment endpoints. In a system with multiple segments connected by joints at their endpoints, like the one shown in Figure 3, this ambiguity implies 2^N allowed reconstructions for fixed scale s, where N is the number of limbs. In the stick figure we use, this corresponds to $2^{11} = 2048$ possible reconstructions. However, by specifying the relative ordering of the endpoints for every limb, a unique solution is obtained.

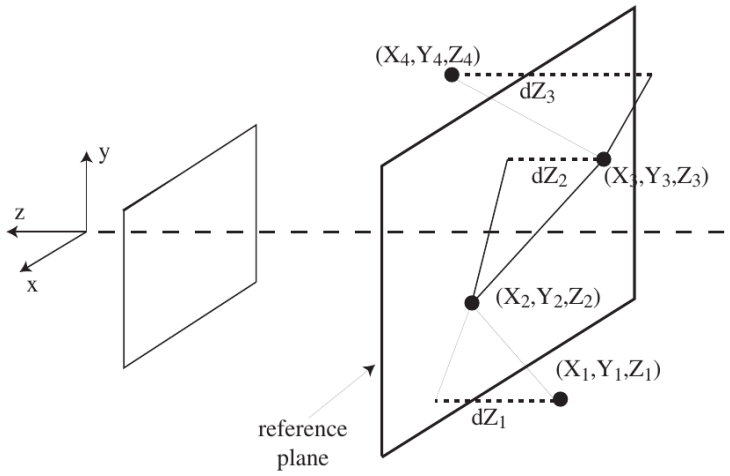


Figure 3: A figure from Taylor's paper[2] showing the projection of an articulated object onto an image under scaled orthographic projection

For this project I implemented Taylor's method (Listing 4 and designed a graphical user interface for selecting joints on an image and indicating which ends of limbs are closer using `tkinter`. An example of it in action is shown in Figure 4. The code for the GUI is in Listing 3 and the 3D rendering is done using `vpython`, with the script in Listing 5. Initially I used `matplotlib` for 3D rendering, with prototype code shown in Listing 6, but `matplotlib` is not well suited for 3D plotting and does not properly perform depth ordering so the resulting figures are harder to interpret.

My Method for Incorporating Joint Position Uncertainty and Optimizing in-plane Limbs

While Taylor's approach usually works well if joint positions and limb endpoint orderings are known exactly, if either of these is difficult to determine, the reconstruction can be very wrong. As shown in Figure 5, the difficulty in intuiting the hip joint locations and endpoint ordering results in a poor reconstruction. Furthermore, the difficulties are compounded by the fact that the subject likely has different limb proportions than the average human.

I discovered that the two aforementioned difficulties for Taylor's method can actually be turned into a strength. Not knowing which end of a limb is closer to the camera contains important information in itself; namely, it tells us

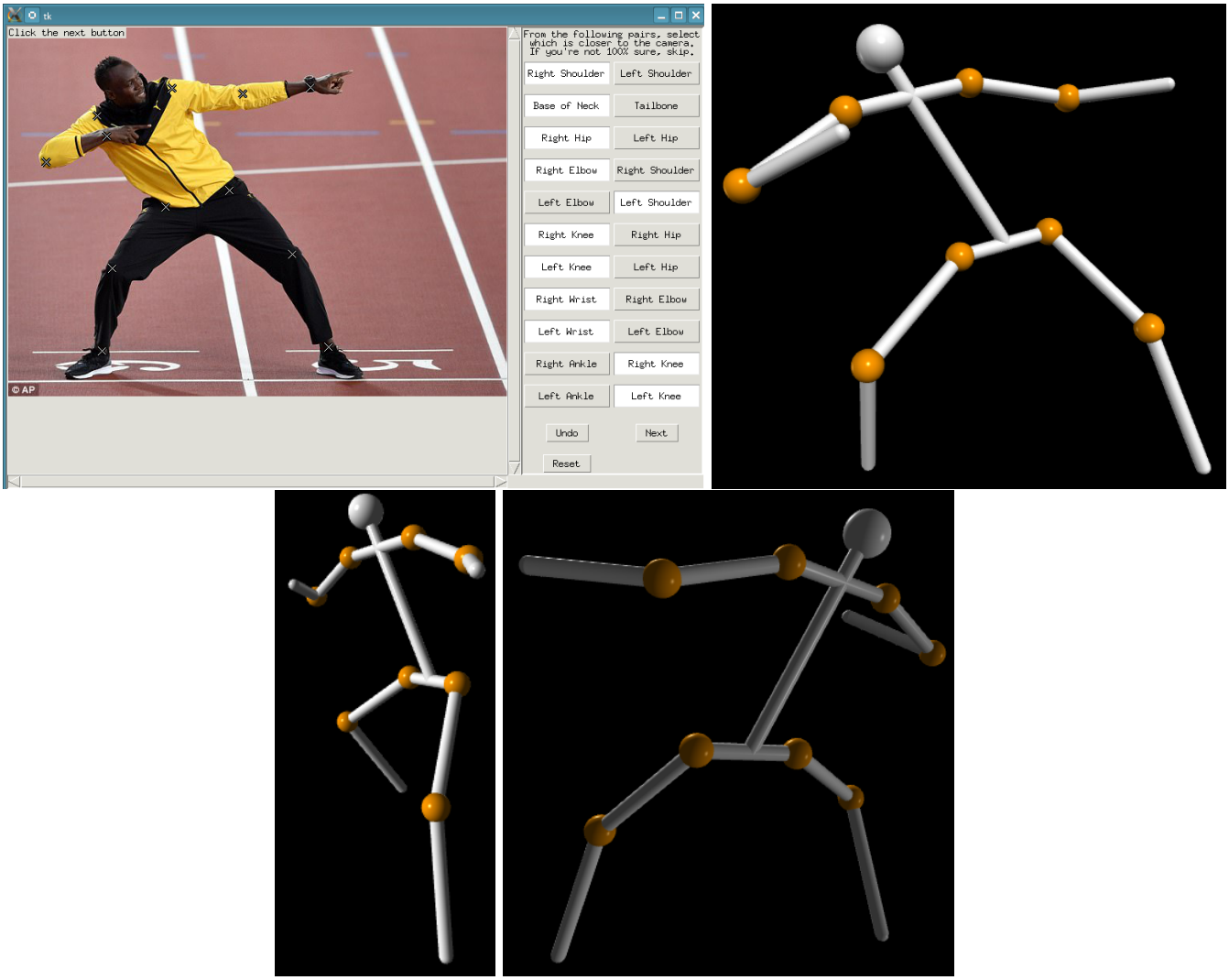


Figure 4: My implementation of Taylor's method applied to the well-known pose of Usain Bolt. In the graphical interface the user is prompted to select the positions of joints on the image, and to select which of the displayed joints are closer to the camera.

that a limb is very nearly perpendicular to the line of sight. Combined with a user conveying a range of uncertainty in 2D joint positions, it is possible to establish a constrained optimization problem.

Mathematical Formulation

First we introduce some mathematical notation for the problem to aid in discussing the proposed optimization. Suppose we have joint index $j \in \{1, 2, \ldots, 12\} = \mathbb{J}$ where each index j corresponds to

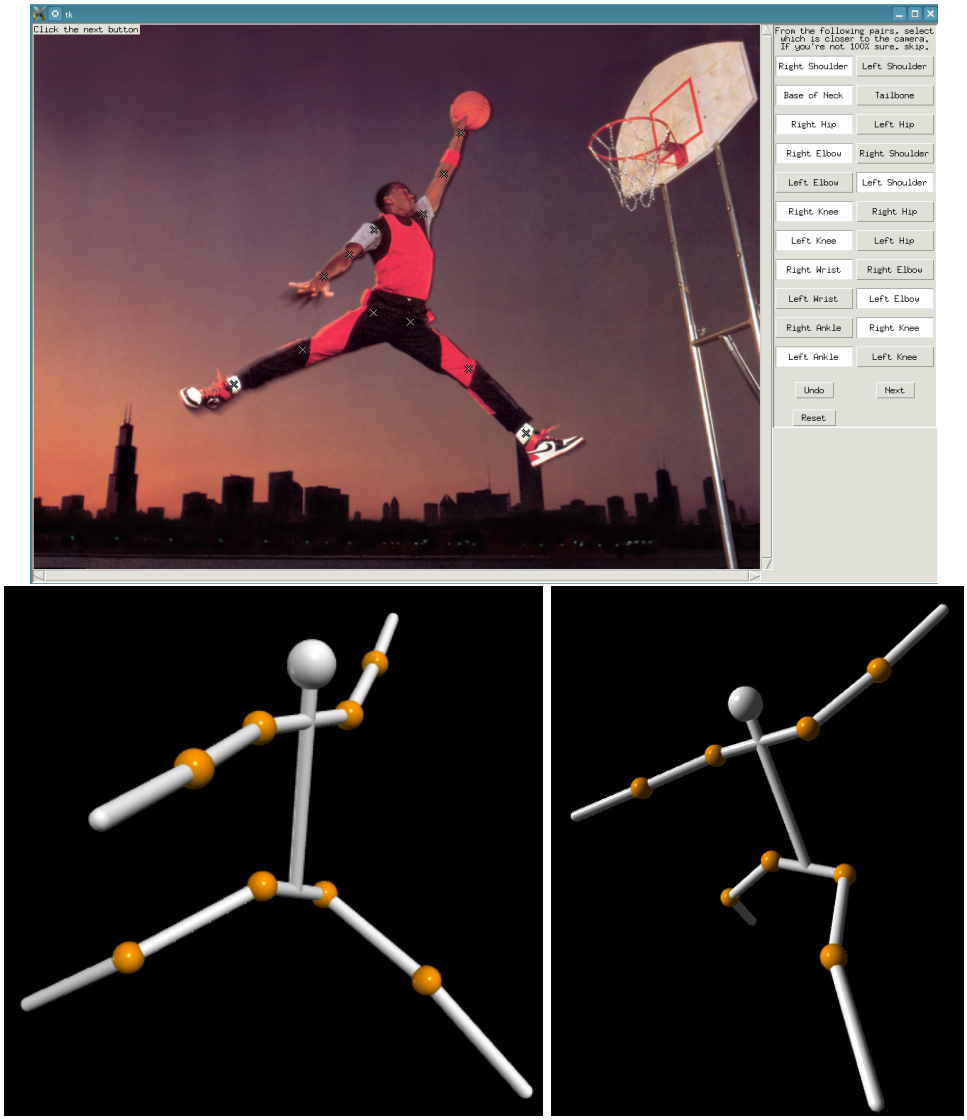


Figure 5: Taylor's method performing poorly on an iconic Michael Jordan pose where many limbs are parallel to the image plane. From the original image, we intuitively know that the legs and left arm are all nearly straight, but there is no way to convey this information to Taylor's algorithm.

joint index j	joint
1	right shoulder
2	left shoulder
3	right hip
4	left hip
5	right elbow
6	left elbow
7	right knee
8	left knee
9	right wrist
10	left wrist
11	right ankle
12	left ankle

in order. Let $\mathbf{U}$ be a vector constructed from the image points (u, v) of the joints in order, *i.e.* $\mathbf{U} = (u_1, v_1, u_2, v_2, \ldots, u_{12}, v_{12})$, and $\mathbf{P}$ be a vector of the actual 3D joint positions (X, Y, Z), also assembled in order. Let $\mathbf{o}$ be a "relative ordering" array consisting of -1 and $+1$s with each element depending on the answer to which neighboring joint is closer to the camera in the following pairwise comparisons:

"limb" index ℓ	"limb"
1	(right shoulder, left shoulder)
2	(collarbone, tailbone)
3	(right hip, left hip)
4	(right elbow, right shoulder)
5	(left elbow, left shoulder)
6	(right knee, right hip)
7	(left knee, left hip)
8	(right wrist, right elbow)
9	(left wrist, left elbow)
10	(right ankle, right knee)
11	(left ankle, left knee)

Let the aforementioned pairwise comparisons be indexed by ℓ for future reference. Taylor's reconstruction algorithm $\mathcal{T}$ is then a (single-valued) function which takes $\mathbf{U}$ and $\mathbf{o}$ and returns $\mathbf{P}$,

$$\mathcal{T}(\mathbf{U}, \mathbf{o}) = \mathbf{P}, \tag{8}$$

where we assume that the scale parameter s which the solution also depends on is always chosen to be the minimum allowed in order to make the function single-valued. While any vector $\mathbf{U} \in \mathbb{R}^{24}$ is allowed, note that $\mathbf{P} \in \mathbb{R}^{36}$ is constrained by the fact that neighboring joints have relative distances fixed by limb lengths. This constraint does not have to be explicitly imposed since it is true by construction in the Taylor algorithm.

My proposed method allows for radii of uncertainty r to be passed alongside each joint's image position, implicitly indicating that a given joint's true image position (u, v) is likely within a circle of radius r from the user-supplied joint image position (u_0, v_0), *i.e.* $\sqrt{(u-u_0)^2 + (u-v_0)^2} \leq r$. Let the array $\mathbf{r}$ be have elements r for each joint in order. Furthermore, define a new augmented relative ordering array $\mathbf{o_a}$ which follows the same rules as the relative ordering array $\mathbf{o}$ except an element is allowed to be 0 when the z-ordering endpoints of the corresponding limb is uncertain (and hence the limb is parallel to the image plane).

Optimization Problem Description

Now, given the initial guess vector of joint locations $\mathbf{U}_0$, There is a set of possible image point functions $\mathbb{U}_{\mathbf{U}_0,\mathbf{r}} \subset \mathbb{R}^{24}$ where any $\mathbf{U} \in \mathbb{U}_{\mathbf{U}_0,\mathbf{r}}$ has to obey

$$\|U_j - U_{0j}\|_2 \leq r_j, \tag{9}$$

for all $j \in \mathbb{J}$, where U_j is the image position (u_j, v_j) of joint j. This space $\mathbb{U}_{\mathbf{U}_0,\mathbf{r}}$ describes the domain of a constrained optimization function.

The objective function in this minimization problem penalizes nonzero dZ for limbs whose endpoints are said to be nearly in plane, *i.e.* where $o_{a,\ell} = 0$, with "limb" index ℓ defined earlier. Let the function $dZ(\ell, \mathbf{P})$ be defined to take a limb index ℓ and 3D joint positions $\mathbf{P}$ and return the difference in z coordinate of the two endpoints of limb ℓ. For the "limb" $\ell = 2$ with endpoints being the collarbone center and tailbone, note that these points are fully determined by the shoulder and hip positions so $dZ(2, \mathbf{P})$ is still well defined.

So, formally our objective function $f_{\mathbf{o}}$ is given by

$$f_{\mathbf{o}}(\mathbf{U}) = \sum_{\{\ell | o_{a,\ell} = 0\}} dZ^2[\ell, \mathcal{T}(\mathbf{U}, \mathbf{o})] \tag{10}$$

where $\mathcal{T}$ is the Taylor algorithm for turning image points $\mathbf{U}$ into 3D positions $\mathbf{P}$. Notice that the objective function has a subscript $\mathbf{o}$ which is to indicate that it depends on a relative ordering array $\mathbf{o}$, as required by the Taylor algorithm. For a given augmented relative ordering array $\mathbf{o_a}$, there are still 2^m possible relative ordering arrays $\mathbf{o}$ where m is the number of zeros in $\mathbf{o_a}$. This reflects the fact that for limbs very nearly parallel to the image plane, there is still one endpoint that is closer than another. We simply expand our minimization problem by computing $\min_{U \in \mathbb{U}_{\mathbf{U}_0,\mathbf{r}}} f_{\mathbf{o}}(\mathbf{U})$ for all $\mathbf{o}$ compatible with $\mathbf{o_a}$, and picking the configuration with the smallest penalty of all of them. To be clear, a relative ordering $\mathbf{o}$ is compatible with $\mathbf{o_a}$ if its elements match all the nonzero elements of $\mathbf{o_a}$.

So, writing the optimization problem down completely,

$$\mathbf{U}_{\min} = \underset{U \in \mathbb{U}_{\mathbf{U}_0,\mathbf{r}}}{\arg\min} f_{\mathbf{o}_{\min}}(U), \quad \text{where} \quad \mathbf{o}_{\min} = \underset{\mathbf{o} \text{ compatible } \mathbf{o_a}}{\arg\min} \left[\min_{U \in \mathbb{U}_{\mathbf{U}_0,\mathbf{r}}} f_{\mathbf{o}}(U) \right] \tag{11}$$

To solve this optimization problem, I used the `scipy.optimize.minimize` implementation of the COBYLA (constrained optimization by linear approximation) method for constrained nonlinear optimization. COBYLA does not need derivative information. It iteratively approaches a solution using linear approximations of the objective function, solving them on a simplex. This method works well with the inequality constraints we have in this problem, equation 9. I chose to use this method since local minima of the objective function appear to be adequate for our purposes, and global minimization would take too long with the high dimensionality of the problem.

Taylor Method with Joint Uncertainty and In-Plane Optimization

Now we can write down my modification to the Taylor method in terms of the previous mathematical quantities. Given an initial image position guess vector $\mathbf{U}_0$, uncertainty array $\mathbf{r}$, and augmented relative ordering array $\mathbf{o_a}$, the reconstructed 3D positions are given by

$$\mathcal{F}(\mathbf{U}_0, \mathbf{r}, \mathbf{o_a}) = \mathcal{T}(\mathbf{U}_{\min}, \mathbf{o}_{\min}). \tag{12}$$

Note that if the augmented relative ordering array contains no zero elements, then it is simply a relative ordering array $\mathbf{o}$ and my method reduces to the Taylor method.

The code for the optimization problem and for my improvement to the Taylor algorithm is shown in Listing 7.

The interface I created allows users to specify a radius of uncertainty for each joint by clicking and dragging to create a circle. As a demonstration of my approach, see Figure 6. Notice that it much better captures the subject's pose. Remember that this is accomplished by perturbing the joint positions *and* by choosing the optimal relative ordering array for ordering limb endpoints for limbs that are very nearly parallel to the image plane.

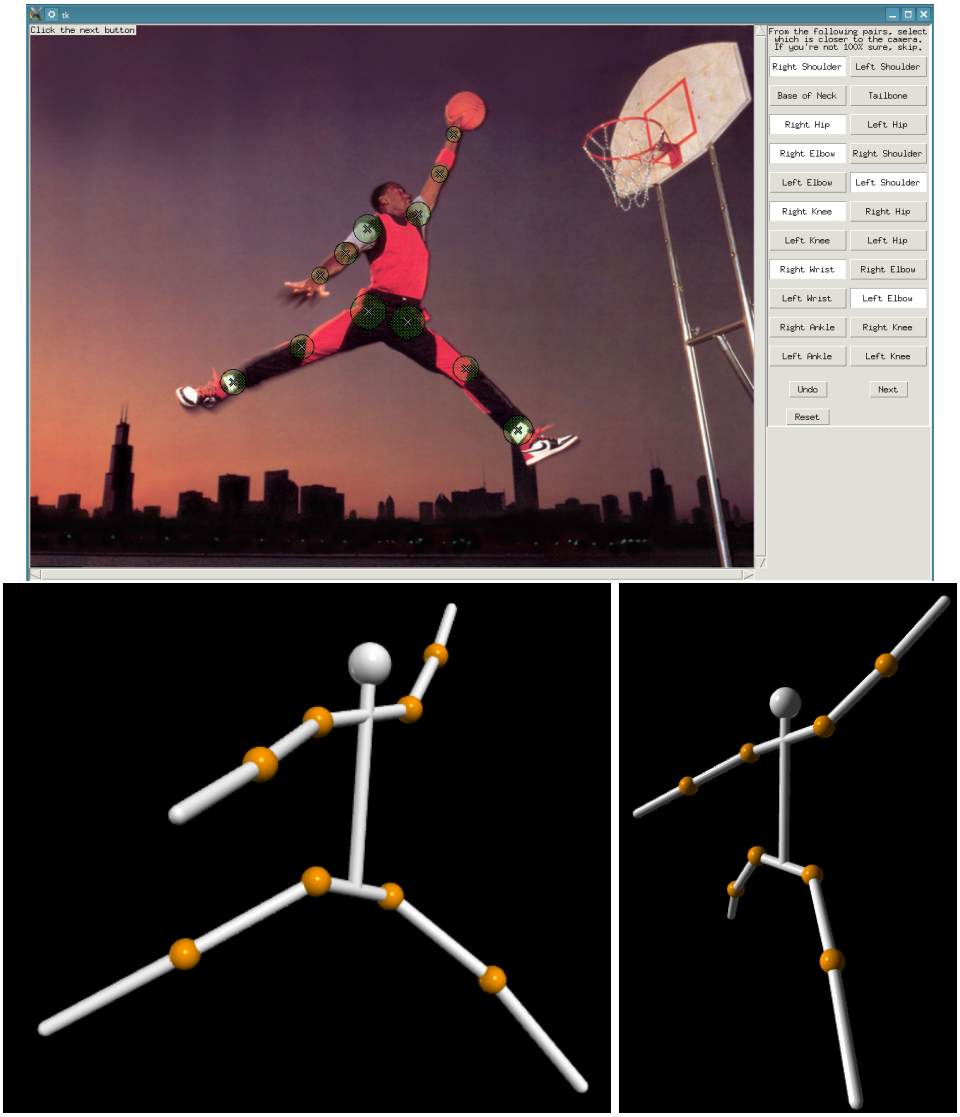


Figure 6: My improvement to Taylor's method by allowing uncertainties in joint positions, as well as optimization of in-plane limbs. Compare this reconstruction to one with the Taylor method shown in Figure 5.

Quantitative Comparison of Improved Method to the Original Taylor Method

In order to quantitatively compare my modified Taylor method to the original Taylor method, we need a ground truth that we can compare to. One way to achieve this is by posing a stick figure in 3D and computing the scaled orthographic projection as a forward problem. In order to make the performance test objective, I add Gaussian noise to the 2D joint image positions to simulate a user not being able to exactly identify where a skeletal joint is located in the bulk of a subject's body. This is all Taylor's method gets in terms of information, while my method additionally receives a reasonable estimate of the standard deviation of the Gaussian noise to represent the user's ability to specify their uncertainty in their selected joint location. Both methods then reconstruct the 3D joint position information, which is then compared to the known ground truth.

Generating Simulated Human Positions and Image Data

In order to produce a valid set of 3D joint positions $\mathbf{P}$ which obey the limb length constraints, it is easiest to use a coordinate system where this constraint is built in. I chose to have limb directions be specified by two local angles

θ, ϕ each, with the head setting the initial position for the body. These rotations are all specified relative to a default rotation vector defined locally for each joint. Then, θ describes the rotation in the direction of the rotation vector (counter-clockwise), while ϕ angles the rotation vector itself in the z direction. To obtain the Cartesian coordinates $\mathbf{P}$, the default stick figure configuration shown in Figure 1 has its limb vectors rotated by rotation matrices specified by these θ and ϕ. This is performed numerically using the Euler-Rodrigues formula to compute a rotation matrix which rotates by θ about the rotation vector which has been tilted towards or away from $\hat{\mathbf{z}}$ by ϕ. The code for this can be found in Listing 8. In order obtain the proper direction for the extremal limb vectors, the rotation matrices for all the limbs connecting it back to the head are applied. For example, for the right forearm, its limb vector is rotated by the rotation matrix at the right elbow, then by the rotation matrix of the right shoulder, then by the rotation matrix of the right collarbone, then finally by the rotation matrix of the head. Once all the limb vectors are pointing the specified directions, a subset can simply be added in sequence to find a given joint position. The code for computing $\mathbf{P}$ from the local angles is in Listing 9.

Computing the Scaled Orthographic Projection

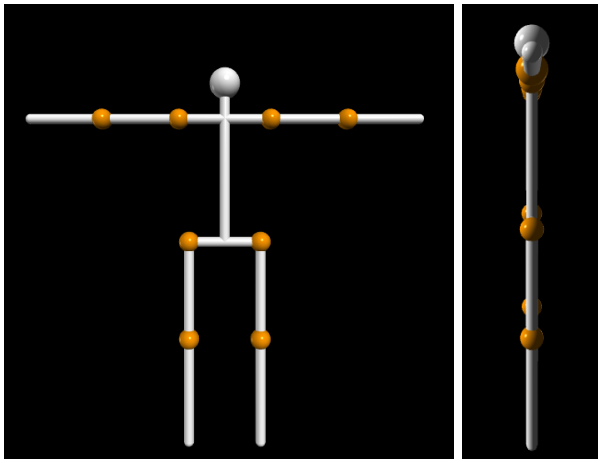
The scaled orthographic projection is simply calculated from equation 1, where we are free to choose a scale. We pick $s = 1$ since it does not matter for the forward problem; it just sets the "size" of the image. The relative 2D joint positions of the projection will be the same either way. The code for this part is in Listing 10. After the projection is computed, for the purposes of the performance test we add some 2D Gaussian noise with a standard deviation of $\sigma = 0.05 \times$ height in each direction, to the position of each limb. For a person of height 178 cm, this is an uncertainty of 8.9 cm in the limb position, which is not too unreasonable for the shoulder and hip location uncertainty in a bulky human. I applied the same amount of uncertainty to every limb for simplicity, but it would be more realistic to have smaller uncertainties for elbow, wrist, and ankle joints due to their smaller size.

Poses

For the following performance test, I created a few test poses that have a certain number of nearly in-plane limbs (limbs perpendicular to line of sight).

Test Pose 1, All Limbs Perpendicular

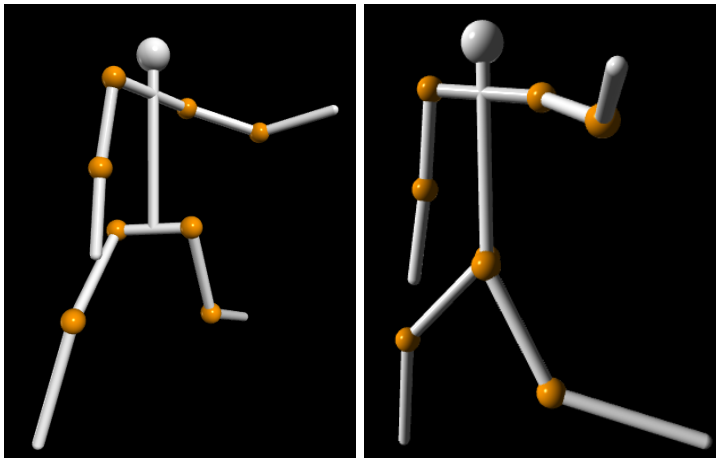
The code to generate this pose is in Listing 11.



In this very simple pose, all limbs are parallel to the image plane.

Test Pose 2: Four Limbs Perpendicular

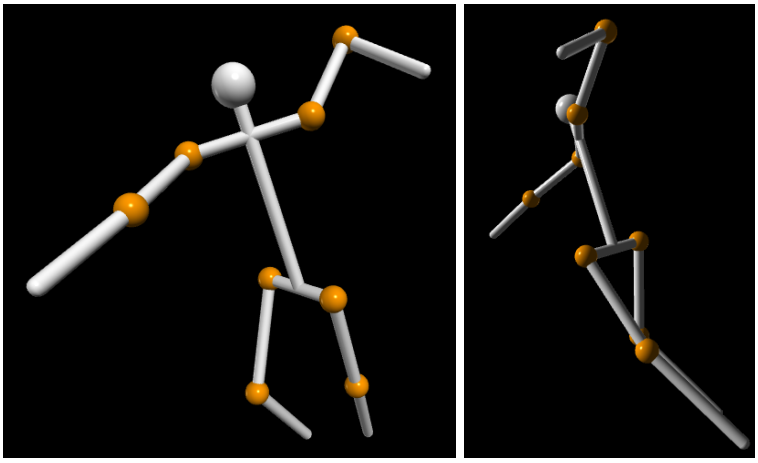
The code to generate this pose is in Listing 12.



In this pose, the right upper arm, right lower arm, hips, and right foreleg are nearly parallel to the image plane.

Test Pose 3: Two Limbs Perpendicular

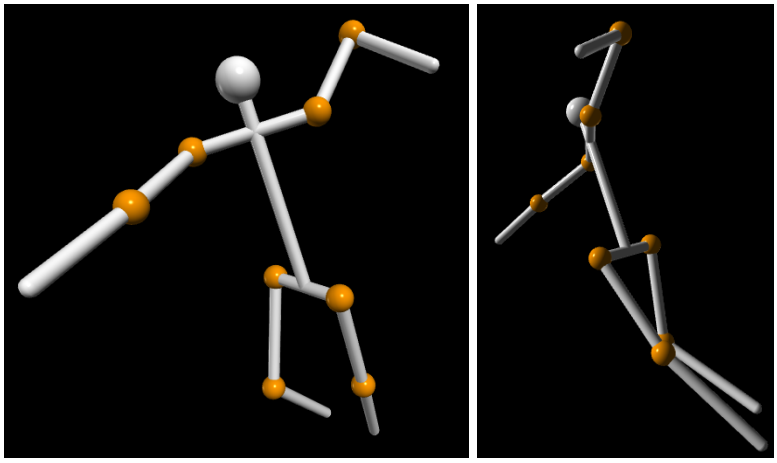
The code to generate this pose is in Listing 13.



In this pose, the shoulders and the right thigh are nearly parallel to the image plane.

Test Pose 4: One Limbs Perpendicular

The code to generate this pose is in Listing 14.



This pose is a slight modification to Test Pose 3, where I moved the right thigh back a bit more so that only the shoulders are nearly parallel to the image plane.

Pose Reconstruction Error

In order to compare the methods, I needed a definition of error in the 3D reconstruction. A natural definition for this quantity would penalize for the distance by which each joint position differs from the corresponding true position. However, we do not concern ourselves determining the size of the reconstructed 3D stick figure, as this overall scaling factor does not change the pose. To remove penalties associated with getting this factor wrong, the error function I use contains an optimization problem as well. Given 3D joint positions $\mathbf{P}$ and a reference $\mathbf{P}_0$, first I subtract away the average joint position of both to set the center of mass of the points at the origin, and then I allow $\mathbf{P}$ to be rescaled by a factor α. The error as a function of rescaling parameter α is given by

$$E(\alpha, \mathbf{P}, \mathbf{P}_0) = \frac{1}{N} \sum_{j \in \mathbb{J}} \|\alpha \mathbf{P}_j - \mathbf{P}_{0j}\|_2 \tag{13}$$

where $N = 12$ is the number of joints. This has the interpretation of being the average joint distance away from the reference joint positions. Now, in order to remove the effect of an overall scaling factor α, I define the error to be

$$E(\mathbf{P}, \mathbf{P}_0) \equiv \min_{\alpha \in \mathbb{R}} E(\alpha, \mathbf{P}, \mathbf{P}_0). \tag{14}$$

To solve this minimization problem I use the `scipy.optimize.minimize` implementation of BFGS, which works well in this problem since $E(\alpha, \mathbf{P}, \mathbf{P}_0)$ is coercive, and a smooth function of α. The fact that it is coercive also guarantees the existence of a global minimum, so our error function is well defined. I have not ensured that all local minima are global minima for this function, so in principle it would be better to use a global minimization routine. However, the minimization is given $\alpha_0 = 1$ as a starting value, and $\mathbf{P}_j$ is expected to be very close to the right scale to start with anyways. Empirically I found that the optimal α are always very close to 1, so we would not expect the error to be significantly different even if there happened to be a local minimum that is slightly larger than the global minimum, as it would have been nearby anyways.

Results

For each of the test poses presented, I generated a set of 10 noisy 2D joint positions which were then given to Taylor's method and my modification to the method. The error definition given in 14 is divided by the height of the stick figure to normalize the distance. This results in an error that can be interpreted as the average distance of a reconstructed joint to the true joint position, relative to the height of the human. Then I computed the average errors from the 10 runs for each test pose and their standard deviations. This information is shown in Figure 7.

The results show that when more than one limb is parallel to the image plane, my method is a significant improvement to the Taylor approach alone. Furthermore, note that in the case where two or four limbs are parallel to the image plane, the average 3D joint position error is comparable to $\sigma\sqrt{8/\pi} = 0.05\sqrt{8/\pi} \approx 0.079$, which is the expected distance a joint would be from the ideal position if the standard deviation $\sigma = 0.05$ we applied to each of the 2D directions were applied to all three directions. This follows from the fact that the expected distance of three independent normally distributed variables of standard deviation σ is given by the mean of the χ_3 distribution, $\sigma\sqrt{8/\pi}$.

When there is only one limb parallel to the image plane, my method performs worse than the Taylor method. This could be due to the optimization sacrificing the positions of other joints in order to meet the one constraint. This potentially improves the joint positions on both sides of the one limb whose dZ it tries to optimize, but since the optimization does not penalize motion of the other joint positions, it can result in a worse overall configuration. However, note that these results were only done for a single test pose, Test Pose 4, and it is possible that there are other poses with only one limb parallel to the image plane where my method can outperform the Taylor method. A more thorough investigation would need to randomly sample poses.

The code for these performance tests is shown in Listing 15.

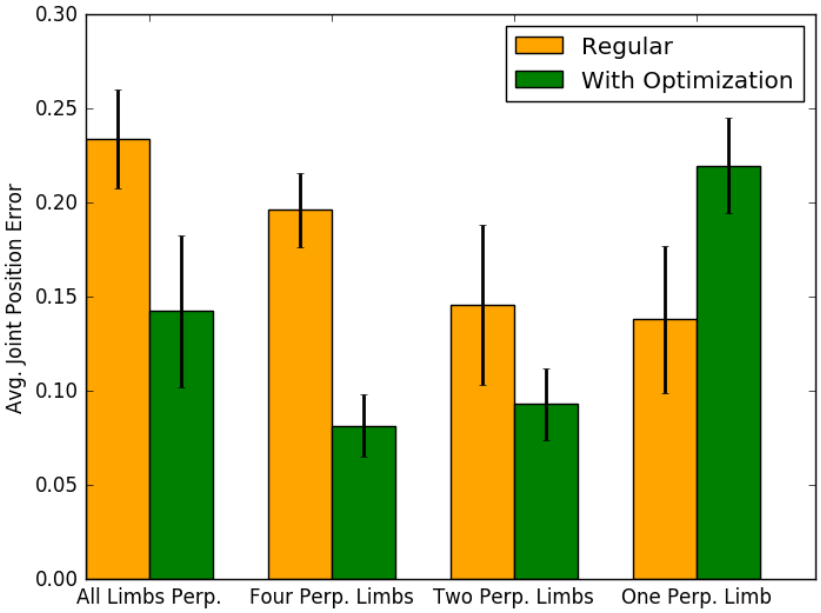
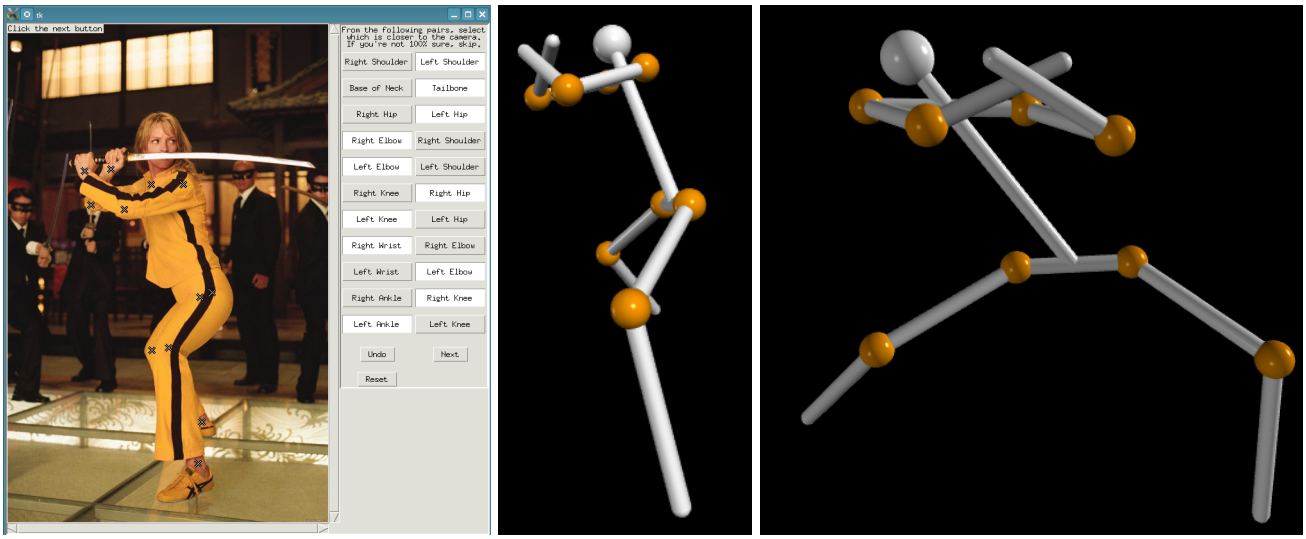


Figure 7: The average joint position error is measured relative to the height of the original stick figure. The bars labeled "Regular" refer to the Taylor method, while the bars labeled "With Optimization" incorporate my modification. Notice that when there is more than one limb perpendicular to the line of sight (parallel to the image plane), my method outperforms Taylor's algorithm.

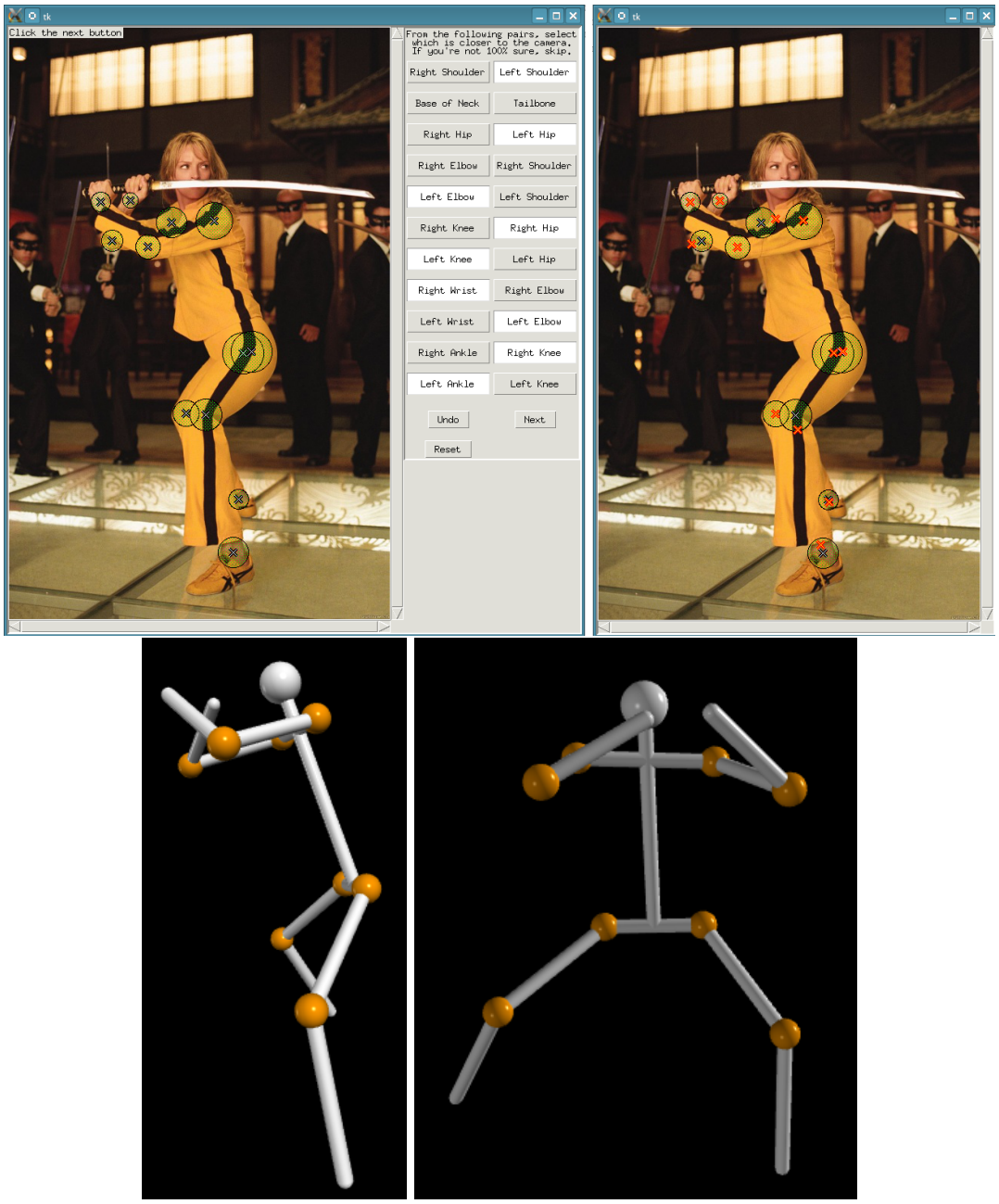
Qualitative Performance Comparison

In the following examples, my method performs noticeably better, where our intuition regarding the pose is sufficient to be the judge. Each of these tests was performed on the first try without any tweaking whatsoever.

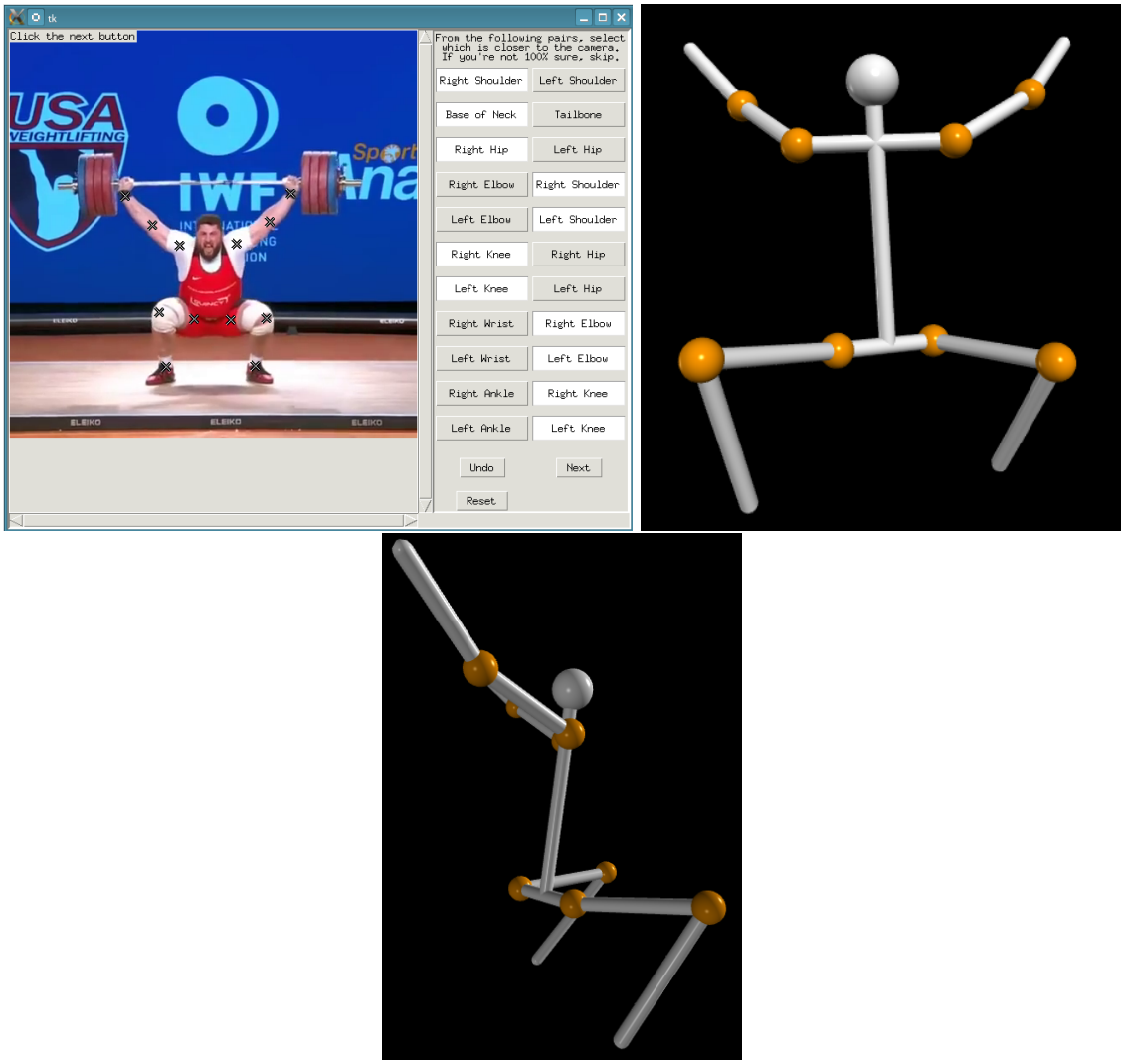
Taylor Method



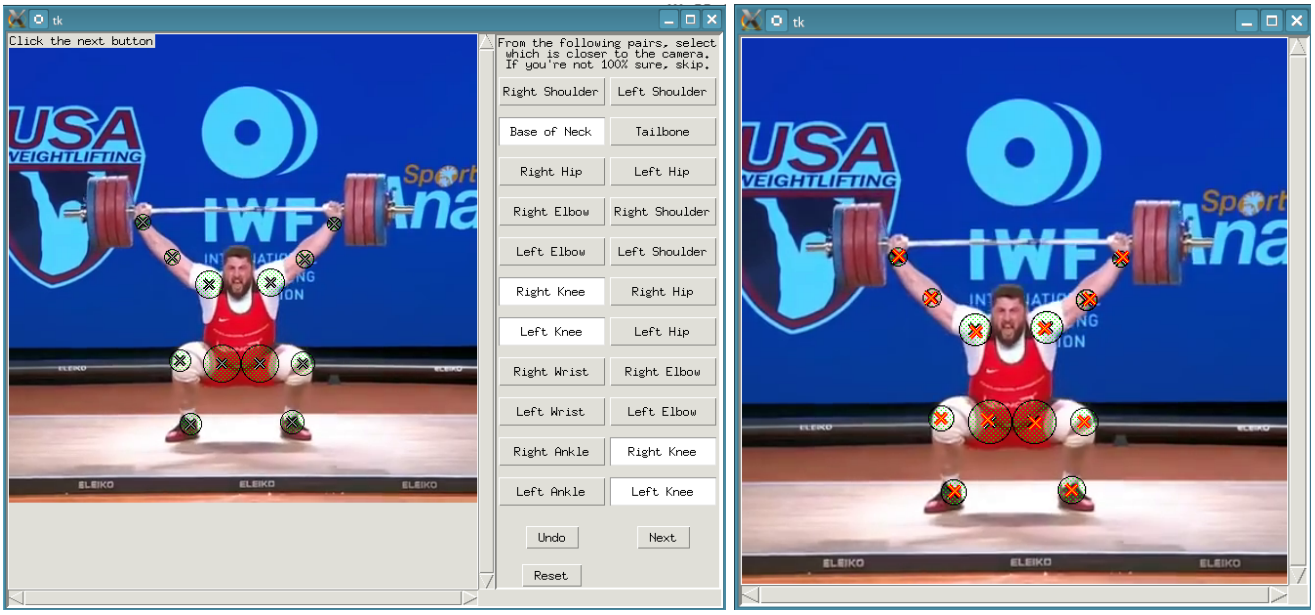
My Method

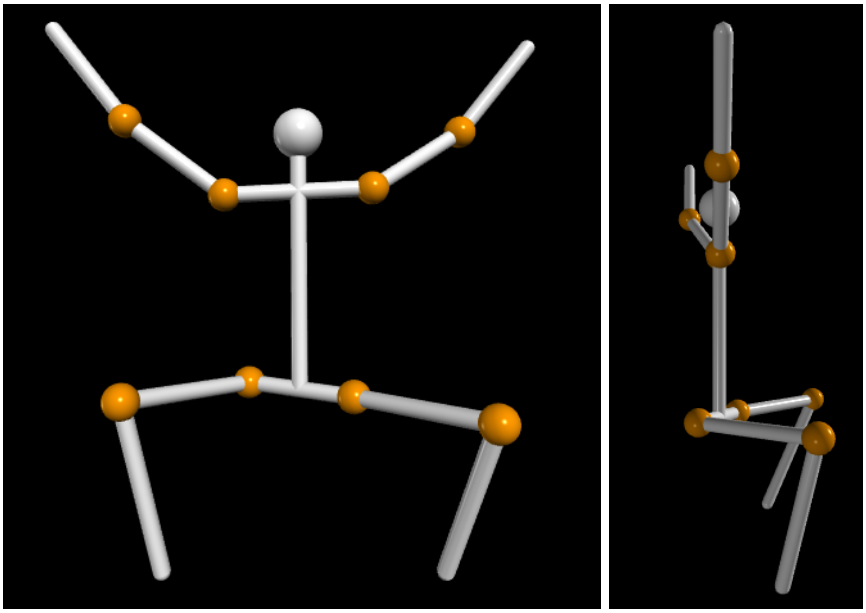

tk
Click the next button
From the following pairs, select which is closer to the camera. If you're not 100% sure, skip.
Right Shoulder
Left Shoulder
Base of Neck
Tailbone
Right Hip
Left Hip
Right Elbow
Right Shoulder
Left Elbow
Left Shoulder
Right Knee
Right Hip
Left Knee
Left Hip
Right Wrist
Right Elbow
Left Wrist
Left Elbow
Right Ankle
Right Knee
Left Ankle
Left Knee
Undo
Next
Reset
tk

Taylor Method

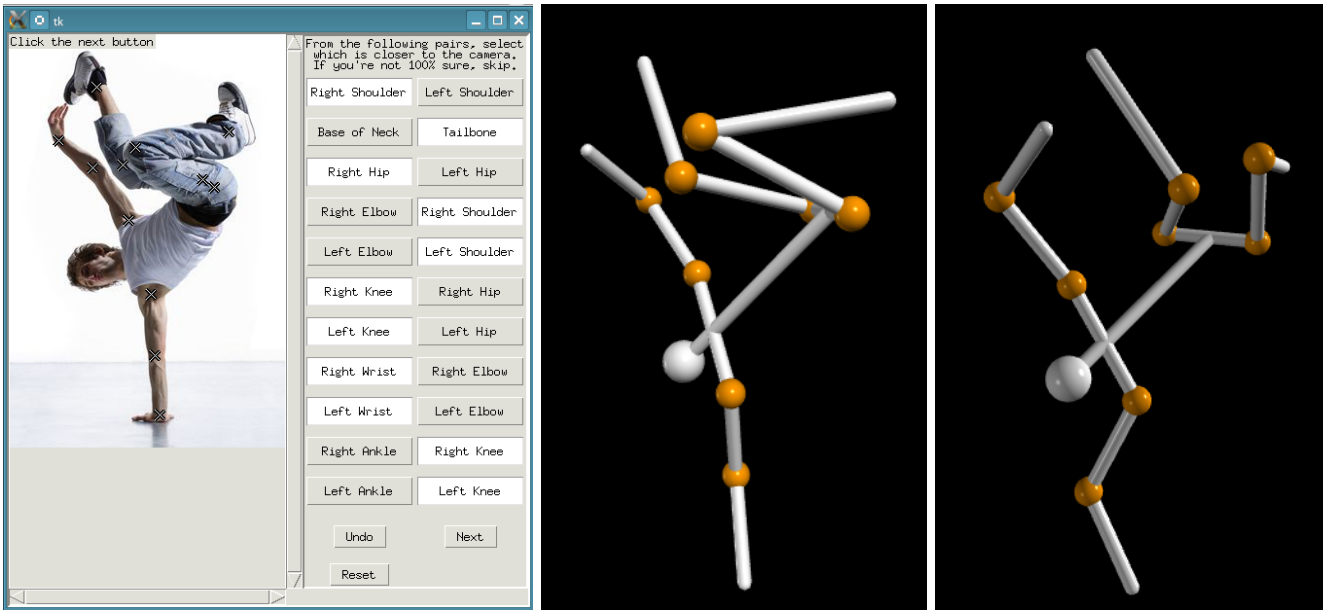


My Method

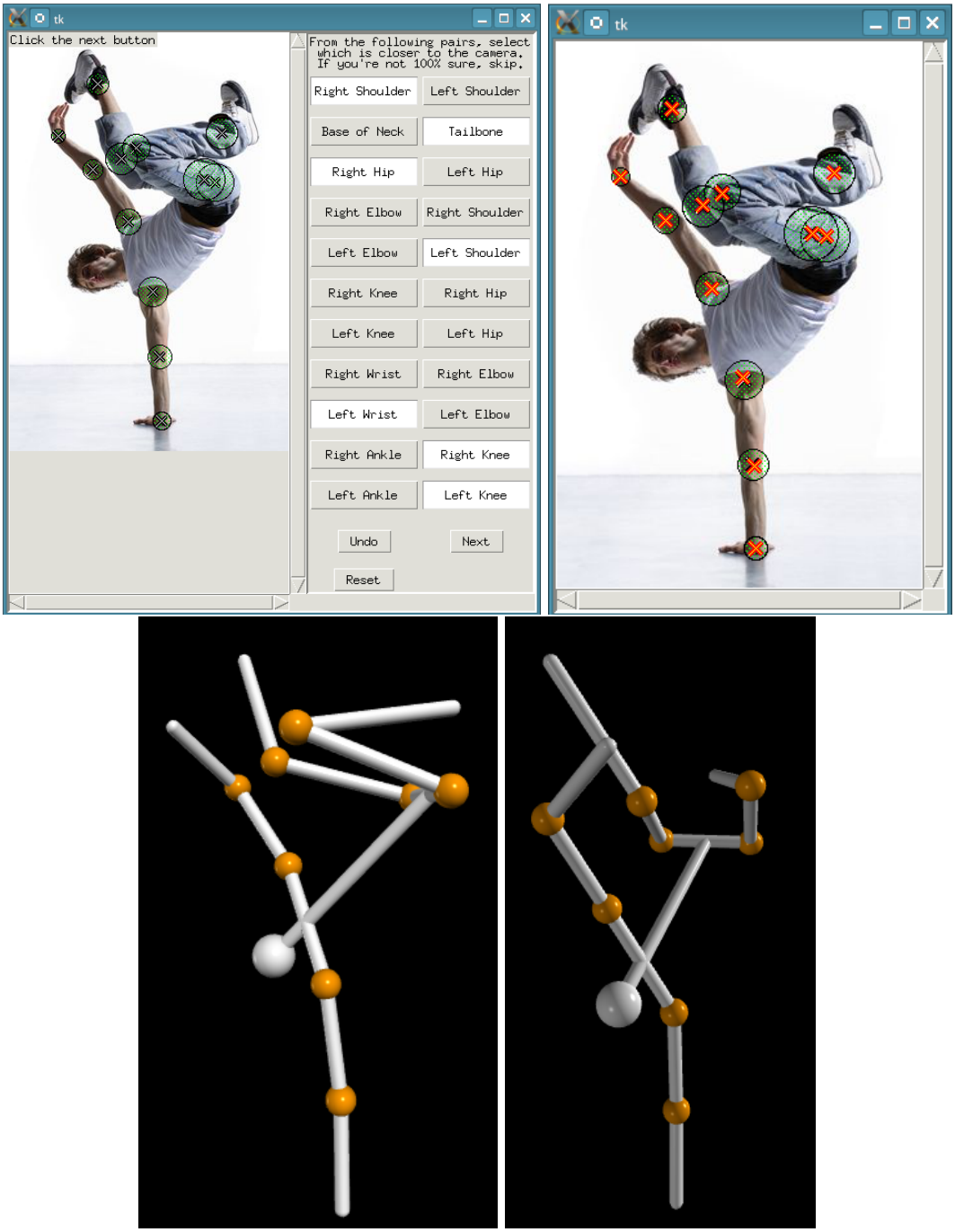




Taylor Method



My Method



Conclusions and Outlook

My method directly addresses the uncertainties that negatively impact the Taylor model, and I show that, surprisingly, these supposed weaknesses can be turned into strengths for the reconstruction process. One possible limitation that I have observed, as shown in the section on Quantitative Results, is that sometimes my method can be worse than the Taylor method when only one limb is perpendicular to the line of sight. If this turns out to generally be the case, we could simply use Taylor's method whenever the number of limbs perpendicular to $\hat{\mathbf{z}}$ is one or zero. To better determine quantitative performance, a natural extension to this project would involve automatically generating many more forward problems than the four we have considered here. The results would give a better sense of my method's performance. Furthermore, since the goal of this project is to reconstruct human poses, an even more refined test of

performance would consider the likelihoods of given 3D poses. Improved performance on common poses should be more heavily weighted than performance on very uncommon human poses.

For real world application, it would also be useful to pair my method with image recognition software to eliminate manual input of joint positions and orderings. Taylor's method has already been successfully paired with neural networks[4] for full automation, so it should be straightforward to implement my approach. If the algorithm for image recognition already has a way to report its uncertainty in a joint location or ordering, this information could be directly passed to the optimization used in the approach presented in this project.

However, this also brings to light a possible limitation to my method, which is shared by the Taylor method. In decoupling the image recognition and the 2D to 3D unprojection, some information is unnecessarily lost. Serra has shown that an approach that closely links the image recognition step with the 3D reconstruction step can outperform two-step approaches when the feature detector in the image recognition step performs poorly[5]. Serra's approach uses observations in the image and from trial 3D reconstructions to iteratively approach a self-consistent result.

My method could also be improved by considering joint angle constraints. There are many limb configurations that are not physiologically possible for the majority of humans, so eliminating this from the search space could result in performance increases. For example, in my method it is possible for the reconstructed pose to have knees bending in the wrong direction, although it is usually not likely if the relative ordering array is correctly set up.

Closely related to monocular pose reconstruction from single images is human motion reconstruction from a monocular video stream. While one could apply single image techniques to each frame of a video stream, much better performance could be obtained from examining frames temporally near the frame in consideration to glean information about the present pose. For example, the scale constraint in equation 7 becomes increasingly restrictive as more frames are analyzed in a video since it becomes almost guaranteed that a limb orients to be perpendicular to the line of sight. Furthermore, temporal information opens up a variety of physically-motivated inference techniques. By constraining motions to be physical, one can again learn a lot about a subject's pose at any given time.

References

[1] H.-J. Lee and Z. Chen, "Determination of 3-d human body postures from a single view," *Computer Vision, Graphics, and Image Processing*, vol. 30, pp. 148–168, 05 1985.

[2] C. J. Taylor, "Reconstruction of articulated objects from point correspondences in a single uncalibrated image," *Comput. Vis. Image Underst.*, vol. 80, pp. 349–363, Dec. 2000.

[3] H. Jiang, "3d human pose reconstruction using millions of exemplars," pp. 1674–1677, 08 2010.

[4] F. X., Z. K., Z. Y., and W. S., "Pose locality constrained representation for 3d human pose reconstruction," *Computer Vision*, vol. 8689, 2014.

[5] E. Simo-Serra, A. Quattoni, C. Torras, and F. Moreno-Noguer, "A joint model for 2d and 3d pose estimation from a single image," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3634–3641, 2013.

Code

Listing 1: meas.py - Relative limb measurements

```
#!/usr/bin/env python
#Author: Cedric Flamant

# Relative body proportion variables

FOREARM = 14.0
UPPER_ARM = 15.0
SHOULDER = 18.0
FORELEG = 20.0
THIGH = 19.0
```

```
PELVIC = 14.0
SPINE = 24.0
NECK = 7.0
HEIGHT = 70.0
```

Listing 2: run.py - Script to launch GUI and pose reconstruction

```
#!/usr/bin/env python
#Author: Cedric Flamant

# How to launch:
# In a terminal, while in this directory, simply run

# python run.py

import numpy as np
import pointstopos
import renderstick
import gui
import displaynewpoints
import optimizepoints


points,pradii,relorder,File=gui.selectpoints()
newpoints,newrelorder = optimizepoints.opt(points,pradii,relorder)
pos,scale = pointstopos.points2pos(newpoints,newrelorder,0.)
displaynewpoints.showpoints(points,newpoints,pradii,File)
renderstick.renderstick(pos)
```

Listing 3: gui.py - Tkinter code for generating a GUI for joint selection and ordering

```
import tkinter as tk
from tkinter import E, W, N, S
from tkinter import filedialog
from PIL import ImageTk
from PIL import Image
import numpy as np


def selectpoints():
    """Brings up tkinter gui for selecting an image, and then prompts
    the user to enter joint locations in order. There are also buttons
    on the right that are used for setting up the relative order array.

    Returns =>
      points: 2D image points of joint locations
      pradii: Radius of uncertainty in the joint locations
      relorder: augmented relative order array
      File: filename of opened image
    """
    root = tk.Tk()
    ijoint = 0
    jointcount = 12
    points = np.zeros((12,2))
    pradii = np.zeros(12)
    relorder = np.zeros(11)
```

```
    jointnames = ["right shoulder", "left shoulder", "right hip", "left hip", "right
        ↪  elbow", "left elbow", "right knee", "left knee", "right wrist", "left
        ↪ wrist", "right ankle", "left ankle"]

    buttonnames=[["Right Shoulder","Left Shoulder"],["Base of Neck","Tailbone"],["
        ↪ Right Hip","Left Hip"],["Right Elbow","Right Shoulder"],["Left Elbow", "
        ↪ Left Shoulder"],["Right Knee","Right Hip"],["Left Knee","Left Hip"],["
        ↪ Right Wrist","Right Elbow"],["Left Wrist","Left Elbow"],["Right Ankle","
        ↪ Right Knee"],["Left Ankle","Left Knee"]]

    buttonvars = []
    for i in range(11):
        buttonvars.append(tk.IntVar())

    buttons = []

    isDown = False
    circle = None

    #setting up a tkinter canvas with scrollbars
    frame = tk.Frame(root, bd=2, relief=tk.SUNKEN)
    frame.grid_rowconfigure(0, weight=1)
    frame.grid_columnconfigure(0, weight=1)
    xscroll = tk.Scrollbar(frame, orient=tk.HORIZONTAL)
    xscroll.grid(row=1, column=0, sticky=E+W)
    yscroll = tk.Scrollbar(frame)
    yscroll.grid(row=0, column=1, sticky=N+S)
    canvas = tk.Canvas(frame, bd=0, xscrollcommand=xscroll.set, yscrollcommand=
        ↪ yscroll.set)
    canvas.grid(row=0, column=0, sticky=N+S+E+W)
    xscroll.config(command=canvas.xview)
    yscroll.config(command=canvas.yview)
    frame.pack(fill=tk.BOTH,expand=1)

    #adding the image
    File = filedialog.askopenfilename(parent=root, initialdir="../images",title='
        ↪ Choose an image.')
    img = ImageTk.PhotoImage(Image.open(File))
    canvas.create_image(0,0,image=img,anchor="nw")
    canvas.config(width=img.width(), height=img.height(), scrollregion=canvas.bbox(
        ↪ tk.ALL))
    selectiontext = tk.Label(frame, text="")
    selectiontext.grid(row=0,column=0, sticky=N+W)

    buttonframe = tk.Frame(frame, bd=2, relief=tk.SUNKEN)
    buttonframe.grid(row=0, column=2, sticky=N)
    instructions = tk.Label(buttonframe, text="From the following pairs, select\
        ↪ nwhich is closer to the camera.\nIf you're not 100% sure, skip.")
    instructions.grid(row=0,columnspan=2)

    for i in range(len(buttonnames)):
        choice = []
        for j in range(len(buttonnames[0])):
            # value is +1 if first choice is selected, -1 if second
            choice.append(tk.Radiobutton(buttonframe, indicatoron=0,
                                         text=buttonnames[i][j], pady=8,
```

```
                                        variable=buttonvars[i],width=15,
                                        value=1-2*j))
        buttons.append(choice)

    for i in range(len(buttons)):
        for j in range(len(buttons[0])):
            buttons[i][j].grid(row=i+1, column=j,pady=5)


    # Set the text informing which joint should be selected
    def updateselectiontext():
        if(ijoint >= jointcount):
            newtext = "Specify joint ordering then click the next button"
        else:
            newtext = "Click (and drag) on the " + jointnames[ijoint]
        selectiontext.config(text=newtext)

    updateselectiontext()

    #Return the points and relative order
    def nextaction():
        nonlocal relorder
        for i in range(len(buttonvars)):
            relorder[i] = buttonvars[i].get()
        root.destroy()

    nextbutton = tk.Button(buttonframe, text="Next", state='disabled',command=
        ↪ nextaction)
    nextbutton.grid(row=len(buttonnames)+1,column=1,pady=14)

    #undo one joint selection
    def undo():
        nonlocal ijoint
        if ijoint > 0:
            ijoint -= 1
            canvas.delete("mark"+str(ijoint))
            updateselectiontext()
            nextbutton.config(state='disabled')


    # undo button
    undobutton = tk.Button(buttonframe, text="Undo", command=undo)
    undobutton.grid(row=len(buttonnames)+1,column=0,pady=14)

    #Reset joints
    def resetjoints():
        nonlocal ijoint
        ijoint = 0
        canvas.delete("marker")
        updateselectiontext()
        nextbutton.config(state='disabled')


    # Reset button
    resetbutton = tk.Button(buttonframe, text="Reset", command=resetjoints)
    resetbutton.grid(row=len(buttonnames)+2,column=0,pady=0)
```

```
    #function to be called when mouse is clicked
    def getcoords(event):
        nonlocal ijoint, points, isDown, circle
        if ijoint < jointcount:
            linewidthouter=3
            linewidthinner=1
            # Start uncertainty circle
            circle = canvas.create_oval(event.x-1,event.y-1,event.x+1,event.y+1,tags
                ↪ =("marker","mark"+str(ijoint)),stipple="gray12",fill="green")
            canvas.create_line(event.x-5,event.y-5,event.x+5,event.y+5,width=
                ↪ linewidthouter,tags=("marker","mark"+str(ijoint)))
            canvas.create_line(event.x-5,event.y+5,event.x+5,event.y-5,width=
                ↪ linewidthouter,tags=("marker","mark"+str(ijoint)))
            canvas.create_line(event.x-5,event.y-5,event.x+5,event.y+5,width=
                ↪ linewidthinner,fill="white",tags=("marker","mark"+str(ijoint)))
            canvas.create_line(event.x-5,event.y+5,event.x+5,event.y-5,width=
                ↪ linewidthinner,fill="white",tags=("marker","mark"+str(ijoint)))
            #saving x and y coords to points array
            points[ijoint,0] = float(event.x)
            points[ijoint,1] = img.height()-float(event.y)
        isDown = True
    #function to be called when mouse is released
    def release(event):
        nonlocal isDown, ijoint, pradii
        if ijoint < jointcount:
            coordinates=canvas.coords(circle)
            radius = np.abs(coordinates[2]-coordinates[0])/2.
            pradii[ijoint] = radius
            ijoint += 1
        isDown = False
        if ijoint >= jointcount:
            nextbutton.config(state='normal')
        updateselectiontext()

    # Only activate when mouse is held down.
    def motion(event):
        nonlocal circle
        if isDown:
            coordinates=canvas.coords(circle)
            centerx = (coordinates[0]+coordinates[2])//2
            centery = (coordinates[1]+coordinates[3])//2
            radius = int(np.sqrt( (event.x-centerx)**2+(event.y-centery)**2))
            canvas.coords(circle,centerx-radius,centery-radius,centerx+radius,
                ↪ centery+radius)

    #mouseclick event
    canvas.bind("<Button 1>",getcoords)
    canvas.bind("<ButtonRelease 1>",release)
    canvas.bind("<Motion>",motion)


    root.mainloop()
    return points, pradii, relorder, File
```

Listing 4: pointstopos.py - Taylor method implementation

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
from meas import *

def points2pos(points,relorder,scale):
    """
    Input =>
    points:
        2d array of u,v image points for each of the following:
         -right shoulder
         -left shoulder
         -right hip
         -left hip
         -right elbow
         -left elbow
         -right knee
         -left knee
         -right wrist
         -left wrist
         -right ankle
         -left ankle
    relorder:
        relative depth order, (+1 if first closer, -1 otherwise)
        of the following segments:
         -right shoulder - left shoulder
         -collarbone - tailbone
         -right hip - left hip
         -right elbow - right shoulder
         -left elbow - left shoulder
         -right knee - right hip
         -left knee - left hip
         -right wrist - right elbow
         -left wrist - left elbow
         -right ankle - right knee
         -left ankle - left knee
    scale:
        Guess of scale. If it is not consistent with constraint,
        gets set to the minimum.

    Returns =>
    pos:
      2d array of positions, where the row index corresponds to:
       -head
       -collarbone
       -tailbone
       -right shoulder
       -left shoulder
       -right hip
       -left hip
       -right elbow
       -left elbow
       -right knee
       -left knee
```

```
        -right wrist
        -left wrist
        -right ankle
        -left ankle
    scale:
      The scale used, whether it is the supplied one or the minimum one
    """
    pos = np.zeros((15,3))
    collarpoint = (points[0,:] + points[1,:])/2.
    tailpoint = (points[2,:] + points[3,:])/2.

    # First compute the constraint on the minimum size of s.
    smins = np.zeros(14)
    smins[0] = s_constraint(points[0,:]-collarpoint,SHOULDER/2.)
    smins[1] = s_constraint(points[1,:]-collarpoint,SHOULDER/2.)
    smins[2] = s_constraint(collarpoint-tailpoint,SPINE)
    smins[3] = s_constraint(points[2,:]-tailpoint,PELVIC/2.)
    smins[4] = s_constraint(points[3,:]-tailpoint,PELVIC/2.)
    smins[5] = s_constraint(points[4,:]-points[0,:],UPPER_ARM)
    smins[6] = s_constraint(points[5,:]-points[1,:],UPPER_ARM)
    smins[7] = s_constraint(points[6,:]-points[2,:],THIGH)
    smins[8] = s_constraint(points[7,:]-points[3,:],THIGH)
    smins[9] = s_constraint(points[8,:]-points[4,:],FOREARM)
    smins[10] = s_constraint(points[9,:]-points[5,:],FOREARM)
    smins[11] = s_constraint(points[10,:]-points[6,:],FORELEG)
    smins[12] = s_constraint(points[11,:]-points[7,:],FORELEG)
    smins[13] = scale
    s = np.amax(smins) #Set to specified scale or minimum allowed scale

    # Now compute the 3D positions assuming the scale and using the
    # relative ordering.
    # Set right shoulder at (0,0,0).
    pos[3,:] = np.zeros(3)
    # left shoulder
    pos[4,:2] = pos[3,:2] + (points[1,:]-points[0,:])/s
    pos[4,2] = pos[3,2] - relorder[0]*dz(points[1,:]-points[0,:],SHOULDER,s)
    # collarbone
    pos[1,:] = (pos[3,:]+pos[4,:])/2.
    # tailbone
    pos[2,:2] = pos[1,:2] + (tailpoint-collarpoint)/s
    pos[2,2] = pos[1,2] - relorder[1]*dz(tailpoint-collarpoint,SPINE,s)
    # right hip
    pos[5,:2] = pos[2,:2] + (points[2,:]-tailpoint)/s
    pos[5,2] = pos[2,2] + relorder[2]*dz(points[2,:]-tailpoint,PELVIC/2.,s)
    # left hip
    pos[6,:2] = pos[2,:2] + (points[3,:]-tailpoint)/s
    pos[6,2] = pos[2,2] - relorder[2]*dz(points[3,:]-tailpoint,PELVIC/2.,s)
    # right elbow
    pos[7,:2] = pos[3,:2] + (points[4,:]-points[0,:])/s
    pos[7,2] = pos[3,2] + relorder[3]*dz(points[4,:]-points[0,:],UPPER_ARM,s)
    # left elbow
    pos[8,:2] = pos[4,:2] + (points[5,:]-points[1,:])/s
    pos[8,2] = pos[4,2] + relorder[4]*dz(points[5,:]-points[1,:],UPPER_ARM,s)
    # right knee
    pos[9,:2] = pos[5,:2] + (points[6,:]-points[2,:])/s
    pos[9,2] = pos[5,2] + relorder[5]*dz(points[6,:]-points[2,:],THIGH,s)
```

```
    # left knee
    pos[10,:2] = pos[6,:2] + (points[7,:]-points[3,:])/s
    pos[10,2] = pos[6,2] + relorder[6]*dz(points[7,:]-points[3,:],THIGH,s)
    # right wrist
    pos[11,:2] = pos[7,:2] + (points[8,:]-points[4,:])/s
    pos[11,2] = pos[7,2] + relorder[7]*dz(points[8,:]-points[4,:],FOREARM,s)
    # left wrist
    pos[12,:2] = pos[8,:2] + (points[9,:]-points[5,:])/s
    pos[12,2] = pos[8,2] + relorder[8]*dz(points[9,:]-points[5,:],FOREARM,s)
    # right ankle
    pos[13,:2] = pos[9,:2] + (points[10,:]-points[6,:])/s
    pos[13,2] = pos[9,2] + relorder[9]*dz(points[10,:]-points[6,:],FORELEG,s)
    # left ankle
    pos[14,:2] = pos[10,:2] + (points[11,:]-points[7,:])/s
    pos[14,2] = pos[10,2] + relorder[10]*dz(points[11,:]-points[7,:],FORELEG,s)
    # compute the head position
    pos[0,:] = NECK/SPINE*(pos[1,:]-pos[2,:]) + pos[1,:]

    return pos,s



def s_constraint(dpoint,l):
    """ Check the minimum value of scale implied by the constraint
    on segment length and image position delta

    Input =>
    dpoint: array corresponding to (u1-u2,v1-v2) point difference
    l: relative length of joint segment

    Returns =>
    smin: minimum scale allowed by constraint
    """
    # add a little to the constraint to avoid floating point issues
    return np.sqrt(dpoint[0]**2 + dpoint[1]**2)/l + 1e-10

def dz(dpoint,l,s):
    """ Compute the |delta Z| implied by the delta image point,
    length of segment, and scale.

    Input =>
    dpoint: array corresponding to (u1-u2,v1-v2) point difference
    l: relative length of joint segment
    s: scale

    Returns =>
    dz: absolute value of the predicted delta z in 3D reconstruction
    """
    return np.sqrt(l**2 - (dpoint[0]**2+dpoint[1]**2)/s**2)
```

Listing 5: renderstick.py - Vpython code for rendering a stick figure in the determined 3D pose

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
import vpython as vp
```

```
#import matplotlib.pyplot as plt
#from mpl_toolkits.mplot3d import Axes3D
import angletopos


def tovec(arr):
    """Converts numpy 3-vector into vp.vector
    Input =>
        arr: 3 element numpy array
    Returns =>
        vec: vpython vector
    """
    return vp.vec(arr[0],arr[1],arr[2])

def renderstick(positions):
    '''Draws the stick figure in 3D

    Input
      positions: 2d array of joint positions.
    '''
    vp.scene.caption = """Right button drag or Ctrl-drag to rotate "camera" to view
        ↪ scene.
    To zoom, drag with middle button or Alt/Option depressed, or use scroll wheel.
        On a two-button mouse, middle is left + right.
    Shift-drag to pan left/right and up/down.
    Touch screen: pinch/extend to zoom, swipe or two-finger rotate."""
    vp.scene.width = 800
    vp.scene.height = 600
    avpos = np.average(positions,axis=0)
    pos = positions - np.tile(avpos,(15,1))
    rarm = vp.curve(pos=[tovec(pos[3,:]),tovec(pos[7,:]),tovec(pos[11,:])],radius=1)
    shoulders = vp.curve(pos=[tovec(pos[3,:]),tovec(pos[4,:])],radius=1)
    larm = vp.curve(pos=[tovec(pos[4,:]),tovec(pos[8,:]),tovec(pos[12,:])],radius=1)
    spine = vp.curve(pos=[tovec(pos[0,:]),tovec(pos[2,:])],radius=1)
    hips = vp.curve(pos=[tovec(pos[5,:]),tovec(pos[6,:])],radius=1)
    rleg = vp.curve(pos=[tovec(pos[5,:]),tovec(pos[9,:]),tovec(pos[13,:])],radius=1)
    lleg = vp.curve(pos=[tovec(pos[6,:]),tovec(pos[10,:]),tovec(pos[14,:])],radius
        ↪ =1)

    head = vp.sphere(pos=tovec(pos[0,:]), radius=3.)
    rshoulder = vp.sphere(pos=tovec(pos[3,:]), radius=2., color=vp.color.orange)
    lshoulder = vp.sphere(pos=tovec(pos[4,:]), radius=2., color=vp.color.orange)
    rhip = vp.sphere(pos=tovec(pos[5,:]), radius=2., color=vp.color.orange)
    lhip = vp.sphere(pos=tovec(pos[6,:]), radius=2., color=vp.color.orange)
    relbow = vp.sphere(pos=tovec(pos[7,:]), radius=2., color=vp.color.orange)
    lelbow = vp.sphere(pos=tovec(pos[8,:]), radius=2., color=vp.color.orange)
    rknee = vp.sphere(pos=tovec(pos[9,:]), radius=2., color=vp.color.orange)
    lknee = vp.sphere(pos=tovec(pos[10,:]), radius=2., color=vp.color.orange)

if __name__=="__main__":
    angles = np.zeros((11,2))
    pos = angletopos.ang2pos(np.zeros(3),angles)
    renderstick(pos)
```

Listing 6: drawstick.py - Prototype code for drawing a 3D stick figure using matplotlib

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import angletopos

def set_axes_equal(ax):
    '''Make axes of 3D plot have equal scale so that spheres appear as spheres,
    cubes as cubes, etc..  This is one possible solution to Matplotlib's
    ax.set_aspect('equal') and ax.axis('equal') not working for 3D.

    Input
      ax: a matplotlib axis, e.g., as output from plt.gca().

    Credit for this function: karlo on StackOverflow
    https://stackoverflow.com/questions/13685386/matplotlib-equal-unit-length-with-
        ↪ equal-aspect-ratio-z-axis-is-not-equal-to
    '''

    x_limits = ax.get_xlim3d()
    y_limits = ax.get_ylim3d()
    z_limits = ax.get_zlim3d()

    x_range = abs(x_limits[1] - x_limits[0])
    x_middle = np.mean(x_limits)
    y_range = abs(y_limits[1] - y_limits[0])
    y_middle = np.mean(y_limits)
    z_range = abs(z_limits[1] - z_limits[0])
    z_middle = np.mean(z_limits)

    # The plot bounding box is a sphere in the sense of the infinity
    # norm, hence I call half the max range the plot radius.
    plot_radius = 0.5*max([x_range, y_range, z_range])

    ax.set_xlim3d([x_middle - plot_radius, x_middle + plot_radius])
    ax.set_ylim3d([y_middle - plot_radius, y_middle + plot_radius])
    ax.set_zlim3d([z_middle - plot_radius, z_middle + plot_radius])

def drawstick(pos):
    '''Draws the stick figure in 3D

    Input
      pos: 2d array of joint positions.
    '''
    fig = plt.figure()
    ax = fig.add_subplot(111, projection = '3d')
    ax.set_aspect('equal')

    ax.plot([pos[0,2],pos[2,2]],[pos[0,0],pos[2,0]],[pos[0,1],pos[2,1]],linewidth=4)
    ax.plot([pos[0,2]],[pos[0,0]],[pos[0,1]],'o',markersize=12,linewidth=4)
    ax.plot([pos[3,2],pos[4,2]],[pos[3,0],pos[4,0]],[pos[3,1],pos[4,1]],'o-',
        ↪ markersize=8,linewidth=4)
    ax.plot([pos[5,2],pos[6,2]],[pos[5,0],pos[6,0]],[pos[5,1],pos[6,1]],'o-',
        ↪ markersize=8,linewidth=4)
```

```
    ax.plot([pos[3,2],pos[7,2]],[pos[3,0],pos[7,0]],[pos[3,1],pos[7,1]],'o-',
        ↪ markersize=8,linewidth=4)
    ax.plot([pos[4,2],pos[8,2]],[pos[4,0],pos[8,0]],[pos[4,1],pos[8,1]],'o-',
        ↪ markersize=8,linewidth=4)
    ax.plot([pos[5,2],pos[9,2]],[pos[5,0],pos[9,0]],[pos[5,1],pos[9,1]],'o-',
        ↪ markersize=8,linewidth=4)
    ax.plot([pos[6,2],pos[10,2]],[pos[6,0],pos[10,0]],[pos[6,1],pos[10,1]],'o-',
        ↪ markersize=8,linewidth=4)
    ax.plot([pos[7,2],pos[11,2]],[pos[7,0],pos[11,0]],[pos[7,1],pos[11,1]],'o-',
        ↪ markersize=8,linewidth=4)
    ax.plot([pos[8,2],pos[12,2]],[pos[8,0],pos[12,0]],[pos[8,1],pos[12,1]],'o-',
        ↪ markersize=8,linewidth=4)
    ax.plot([pos[9,2],pos[13,2]],[pos[9,0],pos[13,0]],[pos[9,1],pos[13,1]],'o-',
        ↪ markersize=8,linewidth=4)
    ax.plot([pos[10,2],pos[14,2]],[pos[10,0],pos[14,0]],[pos[10,1],pos[14,1]],'o-',
        ↪ markersize=8,linewidth=4)
    #ax.scatter(pos[:,0],pos[:,1],pos[:,2], marker = "o", s=110)
    ax.set_xlabel('z')
    ax.set_ylabel('x')
    ax.set_zlabel('y')

    set_axes_equal(ax)
    plt.show()
```

Listing 7: optimizepoints.py - My improvement to the Taylor algorithm for 3D pose reconstruction using optimization

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
import pointstopos
from scipy.optimize import minimize


def calc_com(points):
    """Calculates the center of mass of the given points"""
    return np.average(points,axis=0)

def objectivefn(flatpoints,com,relorder,indexlist):
    """The objective function minimized by my method.

    Inputs =>
      flatpoints: 1D (flattened) array of 2D image points
      com: center of mass of joints (no longer used)
      relorder: relative order array
      indexlist: indices of limbs in plane

    Returns =>
      penalty: the penalty for having designated perpendicular
              limbs not be parallel to image plane
    """
    points = np.reshape(flatpoints,(-1,2))
    pos,scale=pointstopos.points2pos(points,relorder,0.)
    penalty = 0.
    for i in indexlist:
```

```
        if i == 0: # r shoulder & l shoulder
            penalty += (pos[3,2]-pos[4,2])**2
        elif i == 1: # collarbone & tailbone
            penalty += (pos[1,2]-pos[2,2])**2
        elif i == 2: # r hip & l hip
            penalty += (pos[5,2]-pos[6,2])**2
        elif i == 3: # r elbow & r shoulder
            penalty += (pos[7,2]-pos[3,2])**2
        elif i == 4: # l elbow & l shoulder
            penalty += (pos[8,2]-pos[4,2])**2
        elif i == 5: # r knee & r hip
            penalty += (pos[9,2]-pos[5,2])**2
        elif i == 6: # l knee & l hip
            penalty += (pos[10,2]-pos[6,2])**2
        elif i == 7: # r wrist & r elbow
            penalty += (pos[11,2]-pos[7,2])**2
        elif i == 8: # l wrist & l elbow
            penalty += (pos[12,2]-pos[8,2])**2
        elif i == 9: # r ankle & r knee
            penalty += (pos[13,2]-pos[9,2])**2
        elif i == 10: # l ankle & l knee
            penalty += (pos[14,2]-pos[10,2])**2
    return penalty

def reversedbinary(i,numbits):
    """Takes integer i and returns its binary representation
    as a list, but in reverse order, with total number of bits
    numbits. Useful for trying every possibility of numbits choices
    of two."""
    num = i
    count = 0
    revbin = []
    while count < numbits:
        revbin.append(num%2)
        num = num//2
        count += 1
    return revbin

def ineq_constraint(flatpoints,points0,pradii):
    """Inequality constraint passed to minimizer. It ensures that
    trial points are within the radius of uncertainty.

    Inputs =>
      flatpoints: 1D (flattened) array of image points
      points0: User-selected points at the center of uncertainty circle
      pradii: array of uncertainty radii for each joint

    Returns =>
      radius - (distance between point and point0 for each joint)
    """
    dists = np.linalg.norm(np.reshape(flatpoints,(-1,2))-points0,axis=1)
    return pradii - dists

def opt(points0,pradii,relorder,method='COBYLA'):
    """ Optimizer at the heart of the method. Takes guess 2D
    joint positions and their radii of uncertainty and optimizes in
```

```
    order to keep in-plane joints in-plane.

    Inputs =>
      points0: center points guessed by user
      pradii: array of uncertainty radii for each joint
      relorder: relative order array
      method: COBYLA or SLSQP (slower).
    """
    com = calc_com(points0)
    indexlist = []
    relorders = []
    res = []
    # constraints
    cons = ({'type':'ineq','fun':ineq_constraint,'args':(points0,pradii)})
    for i in range(len(relorder)):
        if relorder[i] == 0:
            indexlist.append(i)
    if len(indexlist) > 0:
        numtrials = 2**len(indexlist)
        for i in range(numtrials):
            print("Trial {} of {}".format(i,numtrials))
            trialrelorder = np.copy(relorder)
            revbin = reversedbinary(i,len(indexlist))
            for j,index in enumerate(indexlist):
                trialrelorder[index] = revbin[j]*2 - 1 #make it +1 or -1
            relorders.append(trialrelorder)
            res.append(minimize(objectivefn,points0.flatten(),args=(com,
                ↪ trialrelorder,indexlist),method=method, constraints=cons))
            print("penalty = {}".format(res[i].fun))
        minind = 0
        for i in range(len(res)):
          if res[i].fun < res[minind].fun:
            minind = i
        optpoints = np.reshape(res[minind].x,(-1,2))
        optrelorder = relorders[minind]
    else: # Nothing to optimize, return givens
        optpoints = points0
        optrelorder = relorder
    return optpoints,optrelorder
```

Listing 8: rotmat.py - Rotation matrix generated using the Euler-Rodrigues formula

```
#!/usr/bin/env python
#Author: Cedric Flamant
#
# Produces a rotation matrix for a rotation of theta
# about an axis.

import numpy as np


def rotation_matrix(axis, theta):
    """
    Return rotation matrix associated with CCW rotation about the
    given axis by an angle theta. Uses the Euler-Rodrigues formula
    (written following a stackoverflow post and Wikipedia)
    """
```

```
    axis = np.asarray(axis)
    axis = axis / np.linalg.norm(axis)
    a = np.cos(theta/2.0)
    b, c, d = axis * np.sin(theta / 2.0)
    aa, bb, cc, dd = a*a, b*b, c*c, d*d
    bc, ad, ac, ab, bd, cd = b*c, a*d, a*c, a*b, b*d, c*d
    return np.array([[aa+bb-cc-dd, 2*(bc-ad), 2*(bd+ac)],
                     [2*(bc+ad), aa+cc-bb-dd, 2*(cd-ab)],
                     [2*(bd-ac), 2*(cd+ab), aa+dd-bb-cc]])
```

Listing 9: angletopos.py - For generating joint positions from local angles

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
from rotmat import rotation_matrix as RM
from meas import *

def ang2pos(head_pos,angles):
    """
    Input =>
    head_pos:
        (x,y,z) position of the head
    angles:
        2d array of theta,phi for each of the following (in order):
         -neck
         -shoulder girdle
         -pelvis girdle
         -right upper arm
         -left upper arm
         -right thigh
         -left thigh
         -right forearm
         -left forearm
         -right foreleg
         -left foreleg

    Returns =>
    pos:
      2d array of positions, where the row index corresponds to:
       -head
       -collarbone
       -tailbone
       -right shoulder
       -left shoulder
       -right hip
       -left hip
       -right elbow
       -left elbow
       -right knee
       -left knee
       -right wrist
       -left wrist
       -right ankle
       -left ankle
    """
```

```
    pos = np.zeros((15,3))

    # initialize default limb vectors
    neck = NECK*np.array([0.,-1.,0.])
    spine = SPINE*np.array([0.,-1.,0.])
    rcollar = SHOULDER/2.*np.array([-1.,0.,0.])
    rpelvis = PELVIC/2.*np.array([-1.,0.,0.])
    rupperarm = UPPER_ARM*np.array([-1.,0.,0.])
    lupperarm = UPPER_ARM*np.array([1.,0.,0.])
    rthigh = THIGH*np.array([0.,-1.,0.])
    lthigh = THIGH*np.array([0.,-1.,0.])
    rforearm = FOREARM*np.array([-1.,0.,0.])
    lforearm = FOREARM*np.array([1.,0.,0.])
    rforeleg = FORELEG*np.array([0.,-1.,0.])
    lforeleg = FORELEG*np.array([0.,-1.,0.])

    # calculate rotation matrices for each joint
    neckRM = RM([-np.cos(angles[0,1]),0.,np.sin(angles[0,1])],angles[0,0])
    collarRM = RM([0.,np.cos(angles[1,1]),-np.sin(angles[1,1])],angles[1,0])
    pelvisRM = RM([0.,np.cos(angles[2,1]),-np.sin(angles[2,1])],angles[2,0])
    rshoulderRM = RM([0.,np.cos(angles[3,1]),-np.sin(angles[3,1])],angles[3,0])
    lshoulderRM = RM([0.,-np.cos(angles[4,1]),-np.sin(angles[4,1])],angles[4,0])
    rhipRM = RM([-np.cos(angles[5,1]),0.,-np.sin(angles[5,1])],angles[5,0])
    lhipRM = RM([-np.cos(angles[6,1]),0.,-np.sin(angles[6,1])],angles[6,0])
    relbowRM = RM([0.,np.cos(angles[7,1]),-np.sin(angles[7,1])],angles[7,0])
    lelbowRM = RM([0.,-np.cos(angles[8,1]),-np.sin(angles[8,1])],angles[8,0])
    rkneeRM = RM([-np.cos(angles[9,1]),0.,-np.sin(angles[9,1])],angles[9,0])
    lkneeRM = RM([-np.cos(angles[10,1]),0.,-np.sin(angles[10,1])],angles[10,0])

    # calculate the rotated vectors
    neck = neckRM @ neck
    spine = neckRM @ spine
    rcollar = neckRM @ collarRM @ rcollar
    rpelvis = neckRM @ pelvisRM @ rpelvis
    rupperarm = neckRM @ collarRM @ rshoulderRM @ rupperarm
    lupperarm = neckRM @ collarRM @ lshoulderRM @ lupperarm
    rthigh = neckRM @ pelvisRM @ rhipRM @ rthigh
    lthigh = neckRM @ pelvisRM @ lhipRM @ lthigh
    rforearm = neckRM @ collarRM @ rshoulderRM @ relbowRM @ rforearm
    lforearm = neckRM @ collarRM @ lshoulderRM @ lelbowRM @ lforearm
    rforeleg = neckRM @ pelvisRM @ rhipRM @ rkneeRM @ rforeleg
    lforeleg = neckRM @ pelvisRM @ lhipRM @ lkneeRM @ lforeleg

    # set the head position first
    pos[0,:] = head_pos
    # set collar position
    pos[1,:] = pos[0,:] + neck
    # set tailbone position
    pos[2,:] = pos[1,:] + spine
    # set right shoulder position
    pos[3,:] = pos[1,:] + rcollar
    # set left shoulder position
    pos[4,:] = pos[1,:] - rcollar
    # set right hip position
    pos[5,:] = pos[2,:] + rpelvis
    # set left hip position
```

```
    pos[6,:] = pos[2,:] - rpelvis
    # set right elbow
    pos[7,:] = pos[3,:] + rupperarm
    # set left elbow
    pos[8,:] = pos[4,:] + lupperarm
    # set right knee
    pos[9,:] = pos[5,:] + rthigh
    # set left knee
    pos[10,:] = pos[6,:] + lthigh
    # set right wrist
    pos[11,:] = pos[7,:] + rforearm
    # set left wrist
    pos[12,:] = pos[8,:] + lforearm
    # set right ankle
    pos[13,:] = pos[9,:] + rforeleg
    # set left ankle
    pos[14,:] = pos[10,:] + lforeleg

    return pos
```

Listing 10: postopoints.py - Scaled orthographic projection

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np

def project(pos,scale):
    """ Given 3D positions and a scale factor, this method
    performs scaled orthographic projection to return 2D image points
    """
    # First 3 points of position vector, for the head, collarbone,
    # and tailbone, are not needed in the points vector which
    # only stores the image position of the joints (those three
    # can be calculated from the rest of the joint positions)
    points = np.copy(pos[3:,:2])*scale
    return points
```

Listing 11: testpose1.py - Test Pose 1 generator

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
import angletopos
import renderstick


# Test 1
# Simple default pose, all limbs in plane

angles1 = np.zeros((11,2))
pos1 = angletopos.ang2pos(np.zeros(3),angles1)

# relorder
relorder1 = np.array([1,1,-1,1,-1,1,-1,1,-1,1,-1],dtype=int)
```

```
# relative order with the constraints of all joints being
# equal in z
consrelorder1 = np.array([0,0,0,0,0,0,0,0,0,0,0],dtype=int)
if __name__ == "__main__":
    renderstick.renderstick(pos1)
```

Listing 12: testpose2.py - Test Pose 2 generator

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
import angletopos
import renderstick


# Test 2
# Slightly more intricate pose, four limbs nearly in plane

angles2 = np.zeros((11,2))
# Tilt torso slightly forward
angles2[0,0] = -0.02
# Angle shoulders
angles2[1,0] = 0.8
angles2[1,1] = 0.2
# Angle hips
angles2[2,0] = -0.3
angles2[2,1] = 0.2
# right arm down
angles2[3,0] = np.pi/2
angles2[3,1] = -np.pi/2
# left arm somewhere
angles2[4,0] = 0.67
angles2[4,1] = 0.24
# right thigh somewhere
angles2[5,0] = 0.8
angles2[5,1] = 0.2
# left thigh somewhere
angles2[6,0] = -0.45
angles2[6,1] = 0.1
# right forearm
angles2[7,0] = 0.14
angles2[7,1] = -0.1
# left forearm
angles2[8,0] = 0.74
angles2[8,1] = -0.71
# right foreleg angled down
angles2[9,0] = -0.68
angles2[9,1] = -0.27
# left foreleg
angles2[10,0] = -0.84
angles2[10,1] = 0.21

# relorder
relorder2 = np.array([1,1,-1,1,-1,1,-1,1,1,1,-1],dtype=int)
# relative order with the constraints of certain joints being
# roughly equal in z
```

```
consrelorder2 = np.array([1,0,-1,0,-1,1,-1,0,1,0,-1],dtype=int)

pos2 = angletopos.ang2pos(np.zeros(3),angles2)
if __name__ == "__main__":
    print(pos2)
    renderstick.renderstick(pos2)
```

Listing 13: testpose3.py - Test Pose 3 generator

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
import angletopos
import renderstick


# Test 3
# Trickier pose, two limbs in plane

angles3 = np.zeros((11,2))
# Tilt torso
angles3[0,0] = -0.42
angles3[0,1] = -0.82
# square up Angle shoulders
angles3[1,0] = 0.04
angles3[1,1] = 0.02
# Angle hips
angles3[2,0] = -0.9
angles3[2,1] = -0.2
# right arm down
angles3[3,0] = 1.1
angles3[3,1] = -0.26
# left arm somewhere
angles3[4,0] = -0.91
angles3[4,1] = 0.84
# right thigh in plane
angles3[5,0] = 0.43
angles3[5,1] = -0.2
# left thigh somewhere
angles3[6,0] = -0.25
angles3[6,1] = -0.31
# right forearm
angles3[7,0] = -0.14
angles3[7,1] = -0.19
# left forearm
angles3[8,0] = 1.94
angles3[8,1] = 0.71
# right foreleg angled down
angles3[9,0] = -0.98
angles3[9,1] = -0.17
# left foreleg
angles3[10,0] = -0.24
angles3[10,1] = -0.91

# relorder
relorder3 = np.array([-1,1,-1,1,-1,-1,-1,1,1,-1,-1],dtype=int)
```

```
# relative order with the constraints of certain joints being
# roughly equal in z
consrelorder3 = np.array([0,1,-1,1,-1,0,-1,1,1,-1,-1],dtype=int)

pos3 = angletopos.ang2pos(np.zeros(3),angles3)
if __name__ == "__main__":
    print(pos3)
    renderstick.renderstick(pos3)
```

Listing 14: testpose4.py - Test Pose 4 generator

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
import angletopos
import renderstick


# Test 4
# Tricky pose, only one limb in plane

angles4 = np.zeros((11,2))
# Tilt torso
angles4[0,0] = -0.42
angles4[0,1] = -0.82
# square up shoulders
angles4[1,0] = 0.04
angles4[1,1] = 0.02
# Angle hips
angles4[2,0] = -0.9
angles4[2,1] = -0.2
# right arm down
angles4[3,0] = 1.1
angles4[3,1] = -0.26
# left arm somewhere
angles4[4,0] = -0.91
angles4[4,1] = 0.84
# right thigh somewhere
angles4[5,0] = 0.28
angles4[5,1] = -0.2
# left thigh somewhere
angles4[6,0] = -0.25
angles4[6,1] = -0.31
# right forearm
angles4[7,0] = -0.14
angles4[7,1] = -0.19
# left forearm
angles4[8,0] = 1.94
angles4[8,1] = 0.71
# right foreleg angled down
angles4[9,0] = -0.98
angles4[9,1] = -0.17
# left foreleg
angles4[10,0] = -0.24
angles4[10,1] = -0.91
```

```
# relorder
relorder4 = np.array([-1,1,-1,1,-1,-1,-1,1,1,-1,-1],dtype=int)
# relative order with the constraints of certain joints being
# roughly equal in z
consrelorder4 = np.array([0,1,-1,1,-1,-1,-1,1,1,-1,-1],dtype=int)

pos4 = angletopos.ang2pos(np.zeros(3),angles4)
if __name__ == "__main__":
    print(pos4)
    renderstick.renderstick(pos4)
```

Listing 15: performancetest.py - Tests performance of my method against Taylor's for Test Pose 1-4

```
#!/usr/bin/env python
#Author: Cedric Flamant

import numpy as np
import matplotlib.pyplot as plt
import pointstopos
import postopoints
import angletopos
import optimizepoints
from meas import HEIGHT
from scipy.optimize import minimize
from testpose1 import angles1, relorder1, consrelorder1, pos1
from testpose2 import angles2, relorder2, consrelorder2, pos2
from testpose3 import angles3, relorder3, consrelorder3, pos3
from testpose4 import angles4, relorder4, consrelorder4, pos4

# For this test we take scale = 1 for simplicity
# Thus, height in the picture is the same as height in reality
numsamples = 10
noisespread = 0.05*HEIGHT
pradius = 3*noisespread

def calcerror(rescale, pos, refpos):
    """ Calculates the error as explained in the writeup.

    Inputs =>
      rescale: rescaling factor alpha
      pos: trial 3D position
      refpos: reference joint positions

    Returns =>
      error
    """
    avgdist = np.sum(np.sqrt(np.sum(np.square(rescale*pos-refpos),axis=1)))/refpos.
        ↪ shape[0]
    return avgdist/HEIGHT # normalize this value by the height

def opterror(pos, refpos):
    """Optimized error as explained in writeup. Optimizes away alpha parameter

    Inputs =>
      pos: trial 3D positions
      refpos: reference joint positions
```

```
    Returns =>
      error
    """
    avpos = np.average(pos,axis=0)
    avrefpos = np.average(refpos,axis=0)
    poscentered = pos - np.tile(avpos,(15,1))
    refposcentered = refpos - np.tile(avrefpos,(15,1))
    res = minimize(calcerror, 1.0, method='BFGS',args=(poscentered,refposcentered))
    return res.fun

def getdata(points,relorder,consrelorder,pos,noiseamp):
    """Generates data to make a quantitative performance plot for a
    given pose.

    Inputs =>
      points: exact image points
      relorder: relative order array
      consrelorder: augmented order array
      pos: 3D positions of stick figure
      noiseamp: amplitude of 2D noise to add to image.

    Returns =>
      errorA: error in Taylor method
      errorB: error in my method
    """
    errorA = np.zeros(numsamples)
    errorB = np.zeros(numsamples)
    errorC = np.zeros(numsamples)
    for t in range(numsamples):
        noise = np.random.normal(np.zeros_like(points),noiseamp)
        givenpoints = points + noise

        pradii = np.ones(points.shape[0])*pradius
        # No optimization
        posA,scaleA = pointstopos.points2pos(givenpoints,relorder,0.0)
        errorA[t] = opterror(posA,pos)
        # COBYLA
        newpointsB,newrelorderB = optimizepoints.opt(givenpoints,pradii,consrelorder
            ↪ ,method='COBYLA')
        posB,scaleB = pointstopos.points2pos(newpointsB,newrelorderB,0.)
        errorB[t] = opterror(posB,pos)
    return errorA, errorB


regmean = []
regsdev = []
optmean = []
optsdev = []

# Test 1
# Simple default pose, all limbs in plane
points1 = postopoints.project(pos1,1.0)
regerrors1, opterrors1 = getdata(points1,relorder1,consrelorder1,pos1,noisespread)
regmean.append(np.average(regerrors1))
regsdev.append(np.std(regerrors1))
optmean.append(np.average(opterrors1))
```

```
optsdev.append(np.std(opterrors1))

# Test 2
points2 = postopoints.project(pos2,1.0)
regerrors2, opterrors2 = getdata(points2,relorder2,consrelorder2,pos2,noisespread)
regmean.append(np.average(regerrors2))
regsdev.append(np.std(regerrors2))
optmean.append(np.average(opterrors2))
optsdev.append(np.std(opterrors2))


# Test 3
# Two limbs nearly in plane, rest heavily distorted
points3 = postopoints.project(pos3,1.0)
regerrors3, opterrors3 = getdata(points3,relorder3,consrelorder3,pos3,noisespread)
regmean.append(np.average(regerrors3))
regsdev.append(np.std(regerrors3))
optmean.append(np.average(opterrors3))
optsdev.append(np.std(opterrors3))

# Test 4
# Only one limb nearly in plane, rest heavily distorted
points4 = postopoints.project(pos4,1.0)
regerrors4, opterrors4 = getdata(points4,relorder4,consrelorder4,pos4,noisespread)
regmean.append(np.average(regerrors4))
regsdev.append(np.std(regerrors4))
optmean.append(np.average(opterrors4))
optsdev.append(np.std(opterrors4))


ind = np.arange(len(regmean)) # x locations for groups
width = 0.35 # width of bars

# Make bar plot of results
fig, ax = plt.subplots()
regrects = ax.bar(ind, regmean, width, color='orange', yerr=regsdev,ecolor='black',
    ↪ capsize=2,error_kw={'linewidth':2})
optrects = ax.bar(ind+width, optmean, width, color='green', yerr=optsdev, ecolor='
    ↪ black',capsize=2,error_kw={'linewidth':2})

ax.set_ylabel('Avg. Joint Position Error')
ax.set_xticks(ind + width)
ax.set_xticklabels(('All Limbs Perp.','Four Perp. Limbs','Two Perp. Limbs', 'One
    ↪ Perp. Limb'))
ax.legend((regrects[0],optrects[1]),('Regular','With Optimization'))

plt.savefig('../fig/performance.png',bbox_inches='tight')
plt.show()
```

Listing 16: displaynewpoints.py - Launches a tkinter instance to display the 2D image along with the optimized 2D joint locations

```
import tkinter as tk
from tkinter import E, W, N, S
from PIL import ImageTk
from PIL import Image
```

```
import numpy as np


def showpoints(oldpoints,newpoints,pradii,File):
    """Show the image again along with the uncertainty circles,
    plotting the optimized points on top.

    Inputs =>
      oldpoints: old points guessed by user
      newpoints: optimized joint image points
      pradii: uncertainty radii
      File: filename of image
    """
    root = tk.Tk()

    #setting up a tkinter canvas with scrollbars
    frame = tk.Frame(root, bd=2, relief=tk.SUNKEN)
    frame.grid_rowconfigure(0, weight=1)
    frame.grid_columnconfigure(0, weight=1)
    xscroll = tk.Scrollbar(frame, orient=tk.HORIZONTAL)
    xscroll.grid(row=1, column=0, sticky=E+W)
    yscroll = tk.Scrollbar(frame)
    yscroll.grid(row=0, column=1, sticky=N+S)
    canvas = tk.Canvas(frame, bd=0, xscrollcommand=xscroll.set, yscrollcommand=
        ↪ yscroll.set)
    canvas.grid(row=0, column=0, sticky=N+S+E+W)
    xscroll.config(command=canvas.xview)
    yscroll.config(command=canvas.yview)
    frame.pack(fill=tk.BOTH,expand=1)

    #adding the image
    img = ImageTk.PhotoImage(Image.open(File))
    canvas.create_image(0,0,image=img,anchor="nw")
    canvas.config(width=img.width(), height=img.height(), scrollregion=canvas.bbox(
        ↪ tk.ALL))

    oldpoints = np.copy(oldpoints)
    newpoints = np.copy(newpoints)
    oldpoints[:,1] = img.height()-oldpoints[:,1]
    newpoints[:,1] = img.height()-newpoints[:,1]
    linewidthouter=3
    linewidthinner=1
    for i in range(len(pradii)):
        circle = canvas.create_oval(oldpoints[i,0]-pradii[i],oldpoints[i,1]-pradii[i
            ↪ ],oldpoints[i,0]+pradii[i],oldpoints[i,1]+pradii[i],stipple="gray12",
            ↪ fill="green")
        canvas.create_line(oldpoints[i,0]-5,oldpoints[i,1]-5,oldpoints[i,0]+5,
            ↪ oldpoints[i,1]+5,width=linewidthouter)
        canvas.create_line(oldpoints[i,0]-5,oldpoints[i,1]+5,oldpoints[i,0]+5,
            ↪ oldpoints[i,1]-5,width=linewidthouter)
        canvas.create_line(oldpoints[i,0]-5,oldpoints[i,1]-5,oldpoints[i,0]+5,
            ↪ oldpoints[i,1]+5,width=linewidthinner,fill="white")
        canvas.create_line(oldpoints[i,0]-5,oldpoints[i,1]+5,oldpoints[i,0]+5,
            ↪ oldpoints[i,1]-5,width=linewidthinner,fill="white")
    for i in range(len(pradii)):
        canvas.create_line(newpoints[i,0]-5,newpoints[i,1]-5,newpoints[i,0]+5,
```

```
            ↪ newpoints[i,1]+5,width=linewidthouter, fill="red")
        canvas.create_line(newpoints[i,0]-5,newpoints[i,1]+5,newpoints[i,0]+5,
            ↪ newpoints[i,1]-5,width=linewidthouter, fill="red")
        canvas.create_line(newpoints[i,0]-5,newpoints[i,1]-5,newpoints[i,0]+5,
            ↪ newpoints[i,1]+5,width=linewidthinner,fill="yellow")
55      canvas.create_line(newpoints[i,0]-5,newpoints[i,1]+5,newpoints[i,0]+5,
            ↪ newpoints[i,1]-5,width=linewidthinner,fill="yellow")


    root.mainloop()
```