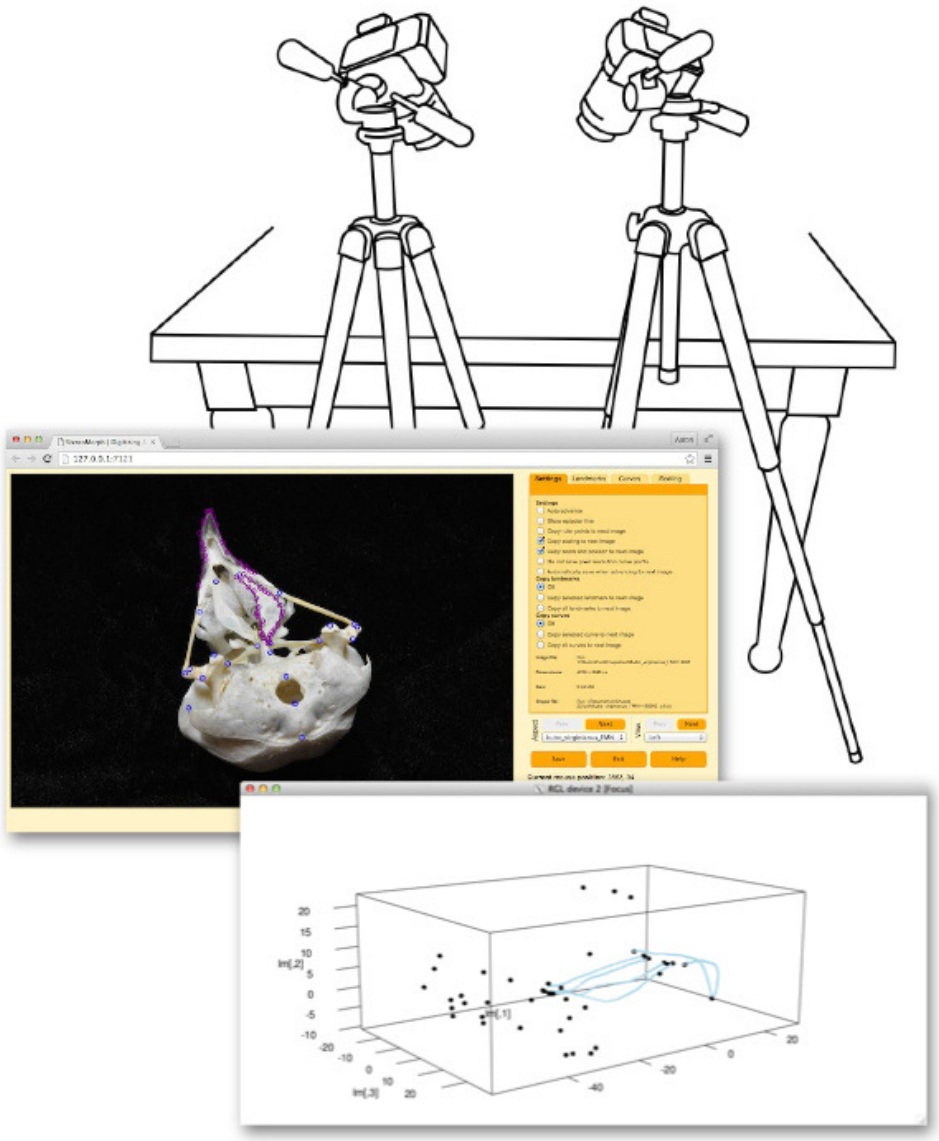
StereoMorph 3D Tutorial

How to collect 3D points and curves using a stereo camera setup



March 2016
Version 1.5

Table of Contents

1 Introduction

Users interested in collecting 3D shape data from biological specimens or other objects confront an ever-growing number of methods, each with its own advantages and disadvantages. There's 3D laser scanning, surface photogrammetry, and CT scanning - just to name the most popular methods out there. These methods are ideal for creating high-resolution 3D surface or volumetric reconstructions. However they require either specialized hardware for the scanning process or specialized software for the reconstruction and digitization of the 3D representations they produce. Additionally, these methods can be time-consuming at one or more steps in the data collection process, making them better suited to the collection of high quality data from relatively few specimens or objects.

I designed the StereoMorph R package specifically for cheap and rapid collection of relatively few landmarks and curves from a large number of specimens or objects. StereoMorph allows you to arbitrarily position two standard digital cameras around some volume of space, calibrate the cameras using a checkerboard, photograph a selection of objects, manually digitize points and curves on these objects from each camera view, and then reconstruct these points and curves into 3D (note that StereoMorph can only be used to reconstruct points and curves, not 3D surfaces).

This tutorial will take you through all the steps of using the StereoMorph R package to reconstruct points and curves in 3D. Each step includes example code from an accompanying example project, "Bird 3D project.zip" (18 MB; download here). If you are using StereoMorph for the first time, you can follow all the steps in sequence and try each step using the example folder. The tutorial can then serve as a useful template for creating your own project. I hope you find this tutorial helpful and wish you the best with your project!

Aaron Olsen
March 2016

2 Getting started

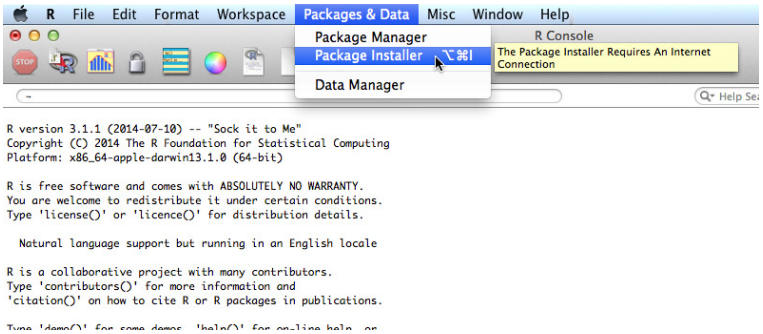
2.1 Installing StereoMorph

This section will show you how to install the R package StereoMorph. The R project is a computing language and platform that allows users to freely upload and share software packages.

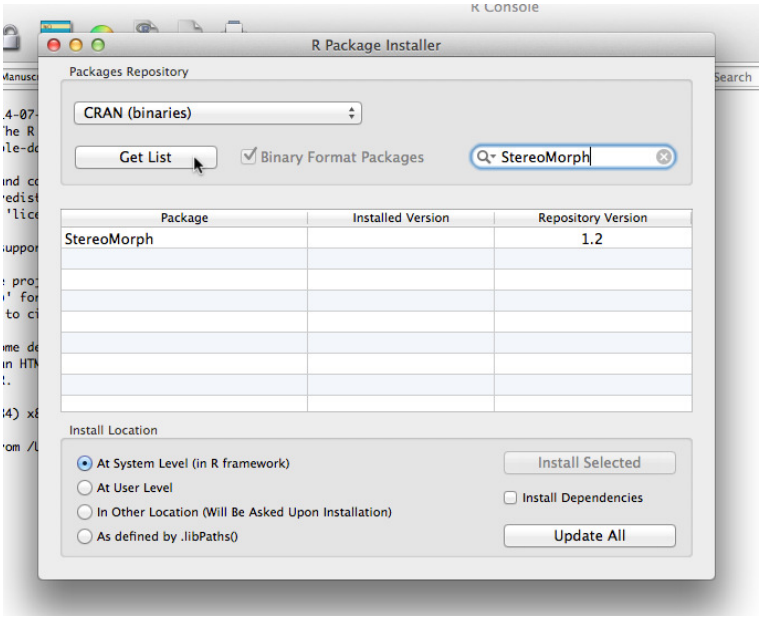
1. If you do not already have R installed on your computer, begin by installing R. R can be installed on Windows, Linux and Mac OS X.

2. Once installed, open R.

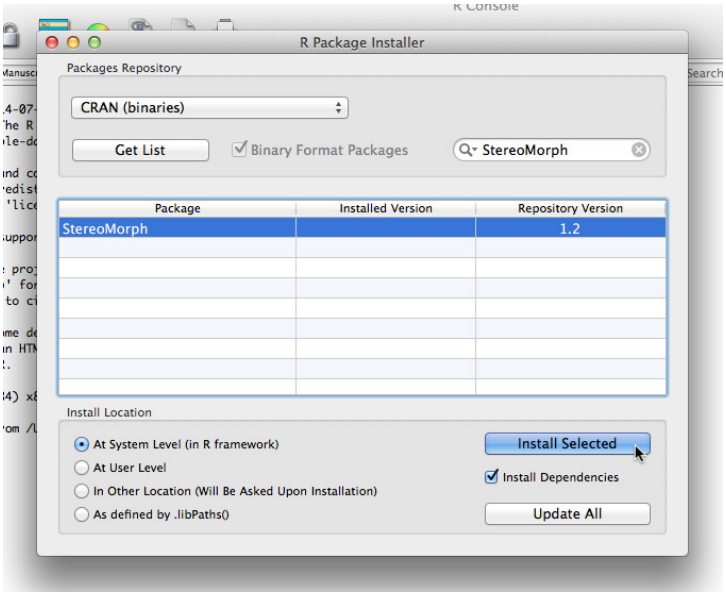
3. Go to *Packages & Data* → *Package Installer*



4. Find the StereoMorph package binary by typing "StereoMorph" into the *Package Search box* and clicking *Get List*.



5. Check the box next to *Install Dependencies*. This ensures that all the packages that StereoMorph requires to run will be installed as well. Then click *Install Selected* to install StereoMorph.

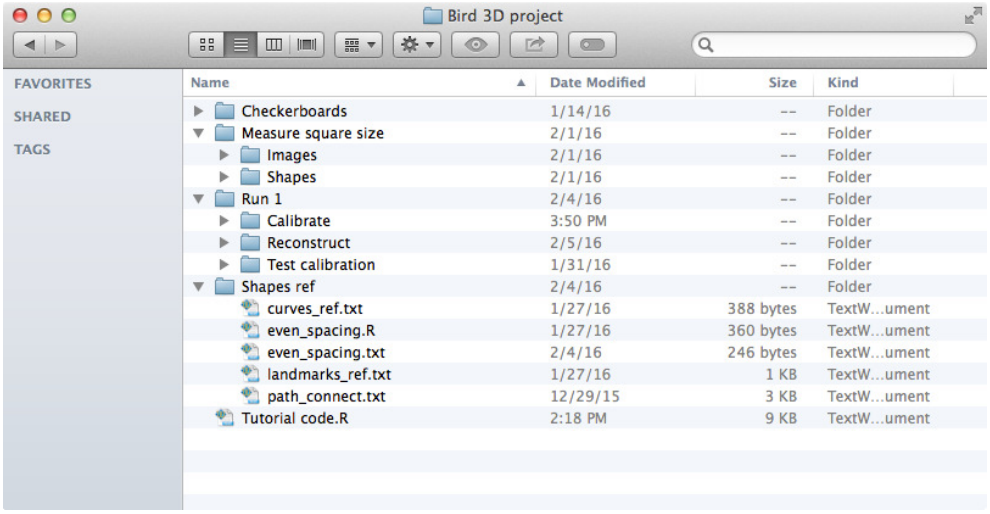


6. Load the StereoMorph package into the current R session using the library command.

```
> library(StereoMorph)
```

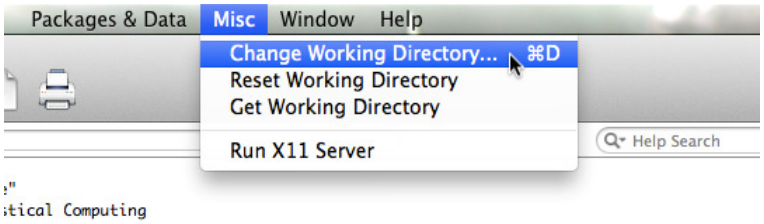
For the rest of this manual, text in blue typewriter font preceded by a ">" will be used to indicate commands to be entered into the R console. You can copy and paste the text from the PDF into the R console (omitting the ">" symbol).

7. If you'd like to run the code in this manual yourself, download and unzip the accompanying tutorial project, "Bird 3D project.zip" (18 MB; download here). This folder contains all the files needed to perform the steps in this manual. Unzip the folder anywhere you'd like on your system.



StereoMorph 3D Tutorial project folder

8. Change the working directory in R to the StereoMorph Tutorial folder so that we easily access the files in R. Go to *Misc* → *Change Working Directory...*



9. Locate and select the unzipped StereoMorph 3D Tutorial project folder and click Open.

2.2 Choosing cameras

In order to use StereoMorph for stereo reconstruction you will need two digital cameras and two tripods. Because the cameras are calibrated for a particular arrangement, the cameras must be able to remain absolutely motionless throughout the entire process of taking potentially hundreds of photographs. Additionally, since the calibration is specific to a particular zoom and focus, these must not change either. This necessitates a few key technical specifications:

- Remote trigger (since touching the shutter button repeatedly causes the camera to move); a wireless remote works best.
- Manual focus mode (i.e. ability to turn off auto-focus).
- Manual zoom mode (i.e. ability to turn off auto-zoom).
- Lens with negligible image distortion.

To meet these specifications you'll probably need to use a DSLR (digital single-lens reflex camera) camera. DSLRs typically permit complete manual control over the zoom and focus of the lens and remote shutter triggering. Cheaper digital camera models tend to have power-saving modes and other automated-only features that make it impossible to have the camera remain motionless for up to an hour while taking photographs. Although web- or computer-cameras might also work these can introduce significant distortion; StereoMorph is not able to correct lens distortion (although I am working on this currently). The two cameras do not have to have the same resolution.

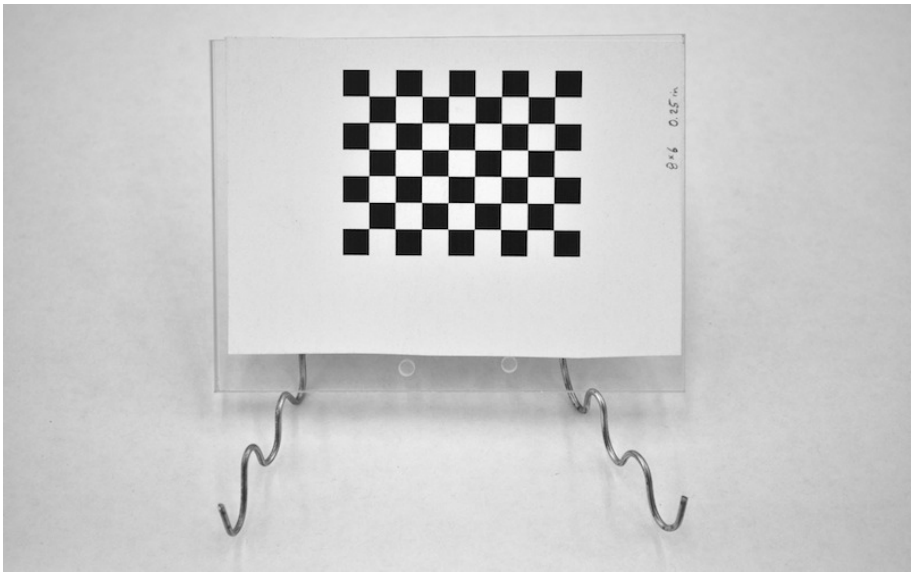
I have been satisfied with the Nikon D5000 that I use but obviously any brand would work (Nikon regularly updates their camera models so the D5000 has been replaced with a more advanced model). For a lens, I use an AF-S DX Nikkor 18-55mm lens zoomed in to 55mm; these are the basic lenses that Niikon sells with many of their cameras. Using this lens at 55mm the distortion is negligible. Any small distortion does not have a measurable effect on the calibration accuracy. Note that if an 18-55mm lens is zoomed

out to 18mm (wide-angle), however, the lens does introduce a lot of distortion. If you have a zoom lens, zooming in as much as possible will reduce the distortion. The cost of a basic camera model (including a lens) that meets these specifications is typically less than $600 (USD).

The section Arranging the cameras has more details regarding specific camera settings and features for stereo arrangement and calibration.

3 Creating a checkerboard

Camera calibration in StereoMorph is based on a checkerboard pattern. The checkerboard provides a sampling of points in a plane that can easily be detected automatically. This saves the user time by not having to manually digitize calibration points. The tutorial uses two checkerboard patterns with squares of two different sizes. The checkerboard with the larger squares is used to calibrate the cameras and the second checkerboard is used to test the calibration accuracy. This section will show you how to create a checkerboard with a particular square size and a simple stand that will permit the checkerboard to stand freely in different positions and at different angles throughout the calibration space.



The checkerboard used in the tutorial to calibrate the cameras.

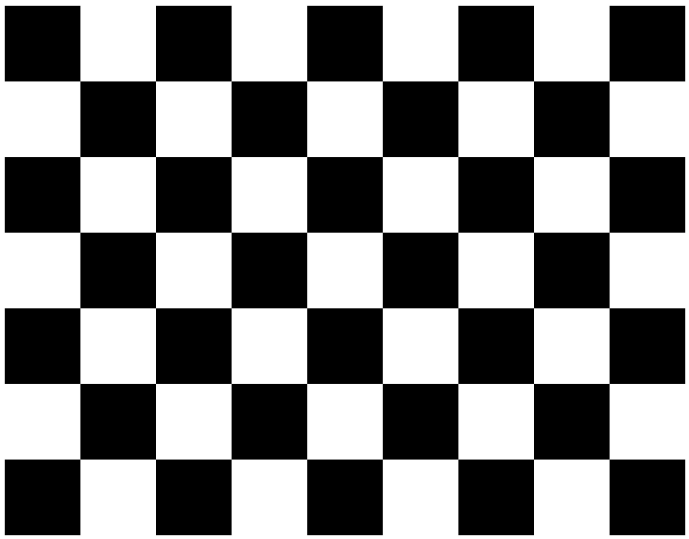
Materials needed for this section:

- Cardstock paper
- A flat, hard surface of the same dimensions as the desired checkerboard
- Glue or tape with which to attach paper to surface
- (Optional) Wire and pliers to create stand to hold checkerboard at different angles

1. Before proceeding, be sure that you've completed all of the steps in the Getting started section, including making the StereoMorph Tutorial folder your current working directory and loading the StereoMorph library into the current R session.

```
> library(StereoMorph)
```

2. In the tutorial project folder you'll find a folder "Checkerboards" that has checkerboards of different sizes saved as .jpeg files. For example, the file "Checkerboard 8x6, 180px (Print at 10 percent scaling for 0.25 inch squares)" is a checkerboard with 8 internal corners along one dimension, 6 internal corners along the other dimension and squares that are 180 pixels along each side. Note that the number of internal corners are not the number of squares but the number of intersections of black squares. The number of internal corners (rather than the number of squares) will be used for the automated detection steps since the code is actually detecting the internal corners (and returning their x,y coordinates).



Checkerboard with 8 x 6 internal corners and 180 pixel squares. If printed at 10% scaling and 72 dpi, squares should be 0.25 inches (6.35 mm).

Standard inkjet or laser printers (at least American printers) will print images at 72 dpi (dots per inch) by default. The DPI and scaling during printing determines how the size of an image in pixels will be converted into inches when the image is printed on paper. The size of the squares on paper can be calculated based on their pixel size, the DPI and scaling using the following formula:

$$\frac{Square\ size}{in\ pixels} * \frac{1}{DPI} * Scaling = \frac{Printed\ square}{size\ in\ in}$$

Or an additional conversion to mm can be added:

$$\frac{Square\ size}{in\ pixels} * \frac{1}{DPI} * Scaling * \frac{25.4\ mm}{inch} = \frac{Printed\ square}{size\ in\ mm}$$

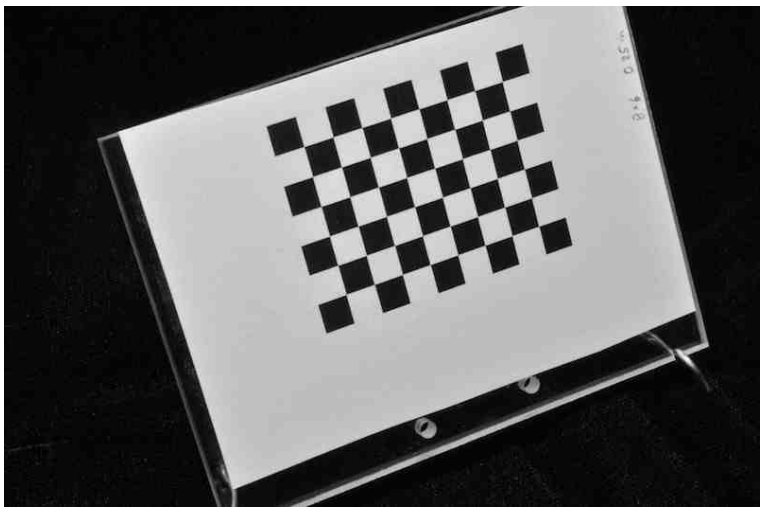
Note that throughout this tutorial "square size" refers to the length along any one side of the square (not, for instance, a diagonal distance or area). In my experience, even using fairly inexpensive desktop printers, this formula matches the printed dimensions well (within at least 100 microns). However, I have only tested this on a couple printers

and there is certainly a chance that other printers are not as accurate. If your setup requires exceptionally high accuracy, the next section (Measuring checkerboard size) will show you how to measure the printed checkerboard square size against an independent standard (e.g. ruler).

Using the above formula, the printed square size in mm of the 8 x 6 checkerboard should be 0.25 inches or 6.35 mm.

$$180\ px * \frac{1\ in}{72\ px} * 0.10 * \frac{25.4\ mm}{in} = 6.35\ mm$$

For the calibration you'll want to use a checkerboard that occupies about half of the image frame when you take the calibration images. This will allow you to place the checkerboard at different positions and angles within the calibration space while still being able to see the entire checkerboard at every position (the entire checkerboard must be visible for corner detection).



Example calibration image. The entire checkerboard is visible but there is enough room to place the checkerboard around the calibration volume at different positions and angles.

For the tutorial, the calibrated volume is about 60 mm x 80 mm x 100 mm. The 8 x 6 checkerboard in the "Checkerboards" folder works well for this. You might have to take a few test photographs of a ruler in your calibration setup to find a good checkerboard size.

3. If you don't find a checkerboard in the "Checkerboards" folder that is suitable for your setup you can use the drawCheckerboard() function. First specify a filename for the checkerboard (including file path if you'd like).

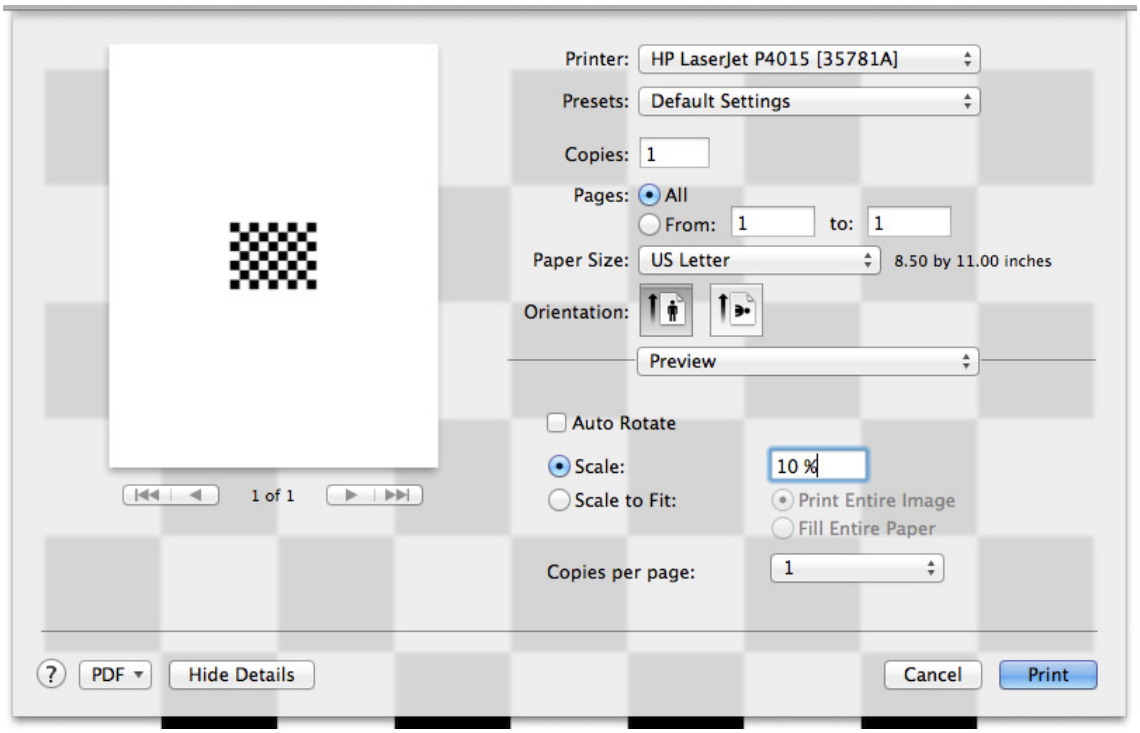
```
> filename <- 'Checkerboard 8x6 (180px).JPG'
```

4. Then call drawCheckerboard() with the following parameters: *nx* (number of internal corners horizontally), *ny* (number of internal corners vertically), *square.size* (size of the squares in the image in pixels), *filename* (where the image will be saved including the filename).

```
> drawCheckerboard(nx=8, ny=6, square.size=180, filename=filename)
```

Make sure that the checkerboard has a different number of internal corners along each dimension. This helps ensure that the corners are returned in the same order when the checkerboard is photographed in different orientations.

5. Once you have a checkerboard image, you are ready to print the checkerboard. I've been able to achieve great calibrations with inexpensive, desktop printers so any decent printer should do. Also, it is best to use a thicker paper such as cardstock. Once taped or glued to a hard surface, cardstock is less likely to get bubbles from moisture over time. For maximal accuracy you want the checkerboard to be as flat as possible. All of the checkerboards in the tutorial folder need to be printed with 10% scaling in order to achieve the dimensions indicated in the filename, so be sure to do this if you're using one of those. This can be done in the printer prompt.



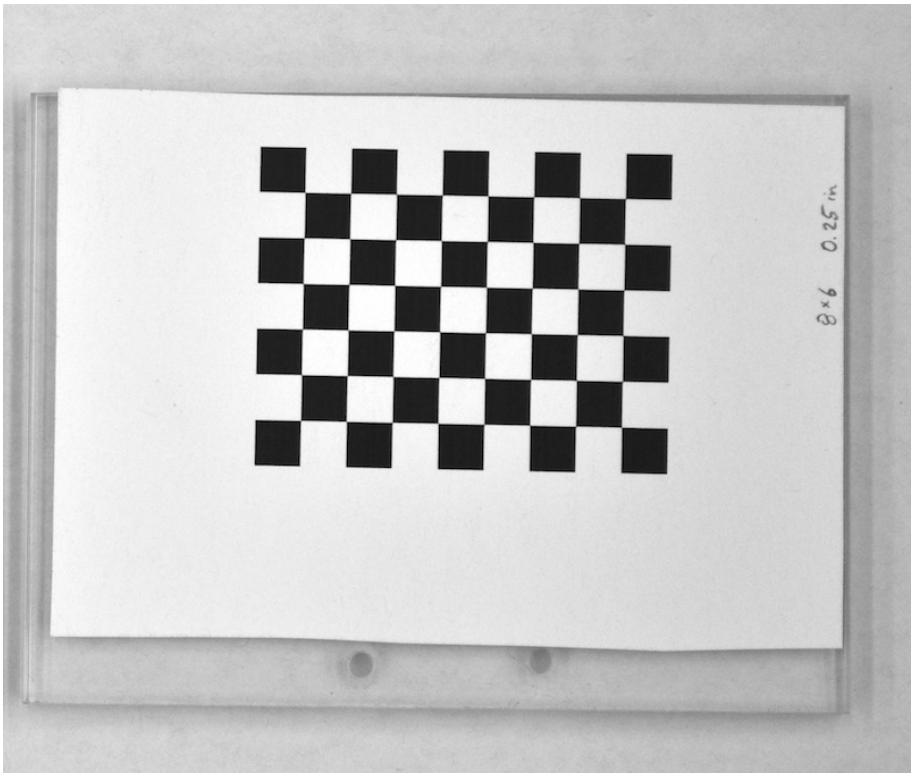
Printing an 8x6 checkerboard at 10% scaling.

6. It's good to write the size of the squares in pixels and inches on the front of the checkerboard lightly in small script with a pencil. This way you can read the square size directly from any photographs you take during the calibration.

7. Attach the checkerboard to a flat, hard surface using glue or tape. This way, the checkerboard can be more easily positioned at different angles without bending. If you don't require exceptionally high accuracy, a flat piece of wood or cardboard should be sufficient. If you want an exceptionally flat surface, I recommend plexiglass (at least 0.22 inches thick) or some other precision milled material.

Thick plexiglass (0.22") works well as a flat surface for high-accuracy applications.

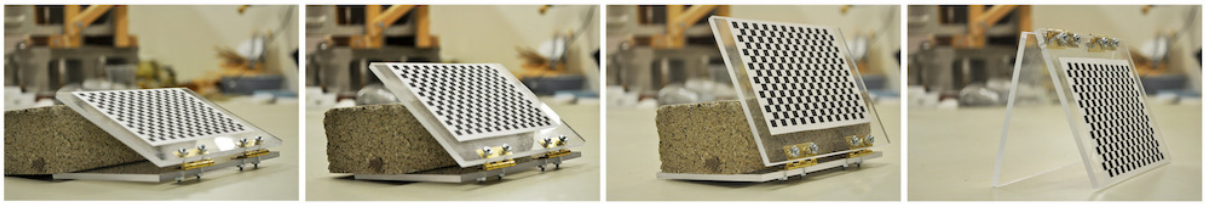
The checkerboard doesn't have to be aligned with the edge of the flat surface. For high-accuracy applications be sure to push out any bubbles between the paper and the surface. I use a spray adhesive to apply the checkerboard to be sure that it lays flat across the entire surface.



Checkerboard glued to plexiglass.

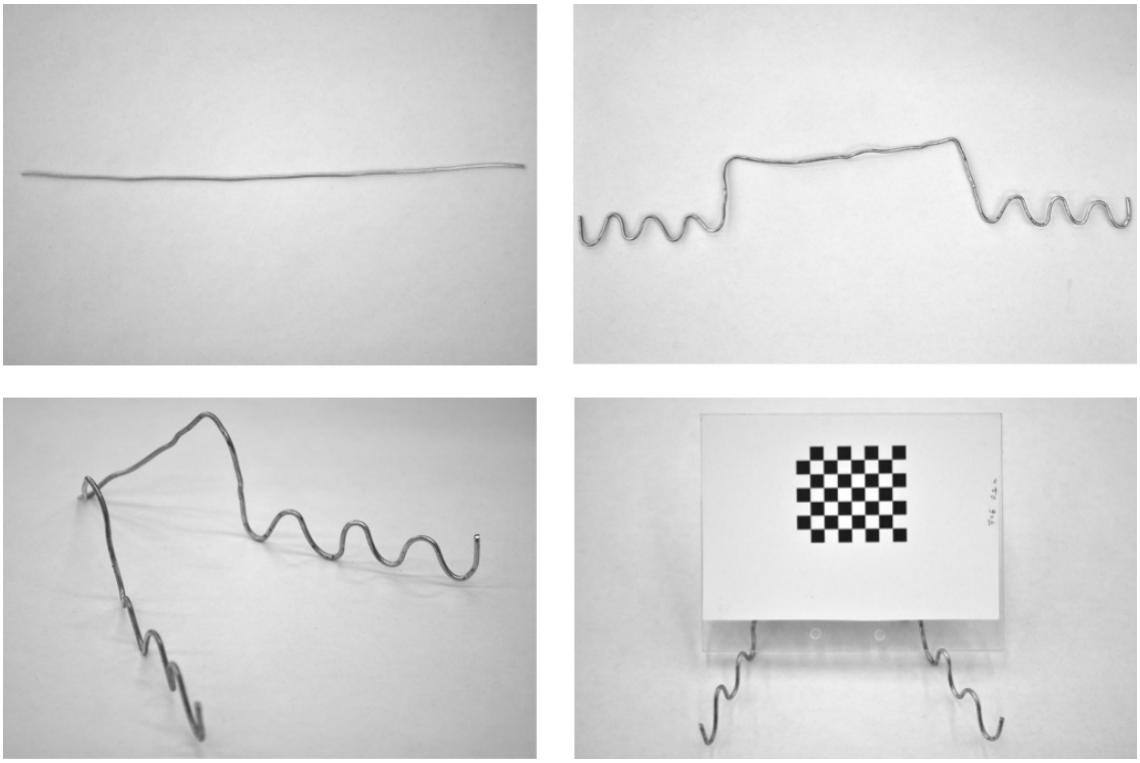
8. Lastly, you'll want to find a way to easily prop up the checkerboard in different positions and at several different angles. Exactly how you do this depends on the size of

your stereo setup. One way that works well for a large, heavy checkerboard is to attach the surface to another object via a hinge and use heavy objects to prop the checkerboard at different angles.



A hinged stand.

If your checkerboard isn't too heavy, you can also create a simple stand with some thick wire (around 16 gauge) and needle nose pliers.



Constructing a simple wire stand.

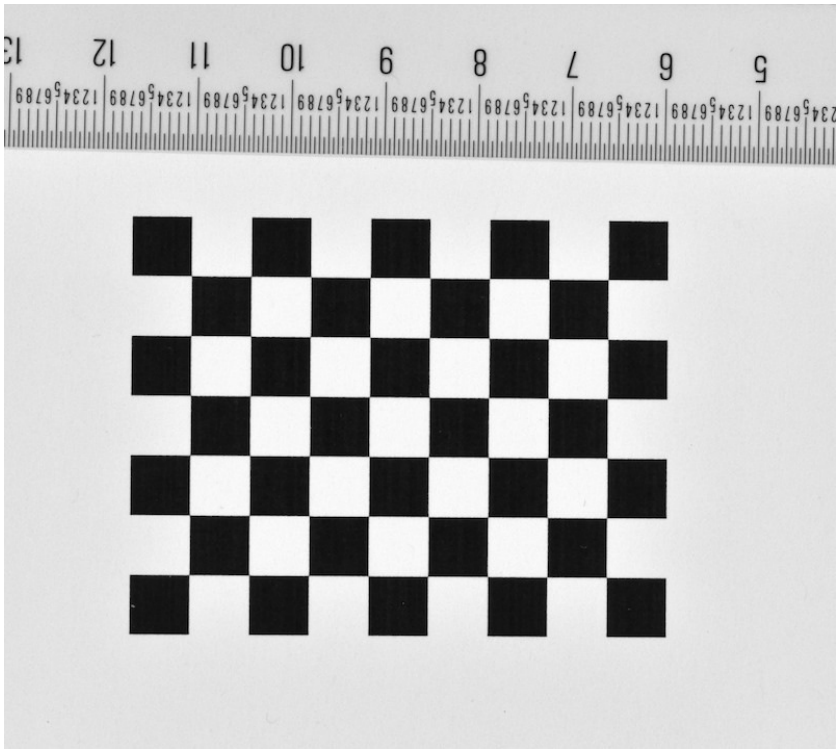
4 Measuring checkerboard square size

This section will show you how to measure the size of the squares in a printed checkerboard pattern against an independent standard such as a ruler. The previous section demonstrated how to calculate the size of printed squares based on the size of the squares in pixels, the DPI and the scaling. For most applications that should predict the size of the squares fairly well. But if your setup requires exceptionally high accuracy or you are unsure about the accuracy of your printer, you can follow the steps in this section.

1. Find a ruler. The required precision of the ruler will depend on the application. A standard office ruler should work well for most applications. If you require high accuracy, you can use a precision ruler. For this tutorial I used a 12" Single "A" - #46-IM precision rule by Schaedler (approximately $30, including tax and shipping), which has an accuracy tolerance of better than 0.00024".

A precision ruler.

2. Take a photograph of the ruler and the checkerboard pattern so that they are both visible in the image.



A photograph of a checkerboard pattern and a ruler. Nikon D5000 with AF-S DX Nikkor 18-55mm lens at 55mm, f/36.

There are few important points when taking the photograph:

- Make sure that the entire checkerboard pattern falls within the image.
- Use a camera lens with minimal lens distortion. For the above image, I used an 18-55mm zoom lens zoomed in all the way to 55mm. At 18mm the lens would have significant barrel distortion but zoomed in to 55mm the distortion is negligible.
- Position the camera and checkerboard so that the checkerboard is approximately coplanar with the image plane or the end of the camera lens (i.e. the long-axis of the lens should be at a right angle to the checkerboard). If the checkerboard is at an angle relative to the image plane, some squares will be closer to the image plane than others, resulting in a difference in size on the imaging plane (this is the perspective effect).
- For the same reason, the ruler should be in the same plane as the checkerboard pattern. If the ruler has some depth to it, raise the checkerboard so that it is coplanar with the points you'll be digitizing on the ruler.
- Position the ruler so that it is not at the very edge of the image. It is not possible to eliminate lens distortion entirely when taking a photograph and the edges will generally have the highest distortion. So it is best to keep everything being measured away from the edges.

3. Upload the photograph of the checkerboard and ruler to your computer. Be sure the image names only contain letters, numbers or underscores (no spaces, commas or periods). The tutorial folder "Measure square size" has photographs of the two checkerboards used in this tutorial with a ruler. Here we'll measure the square size for the two checkerboards used in the tutorial (7x6 and 8x6).

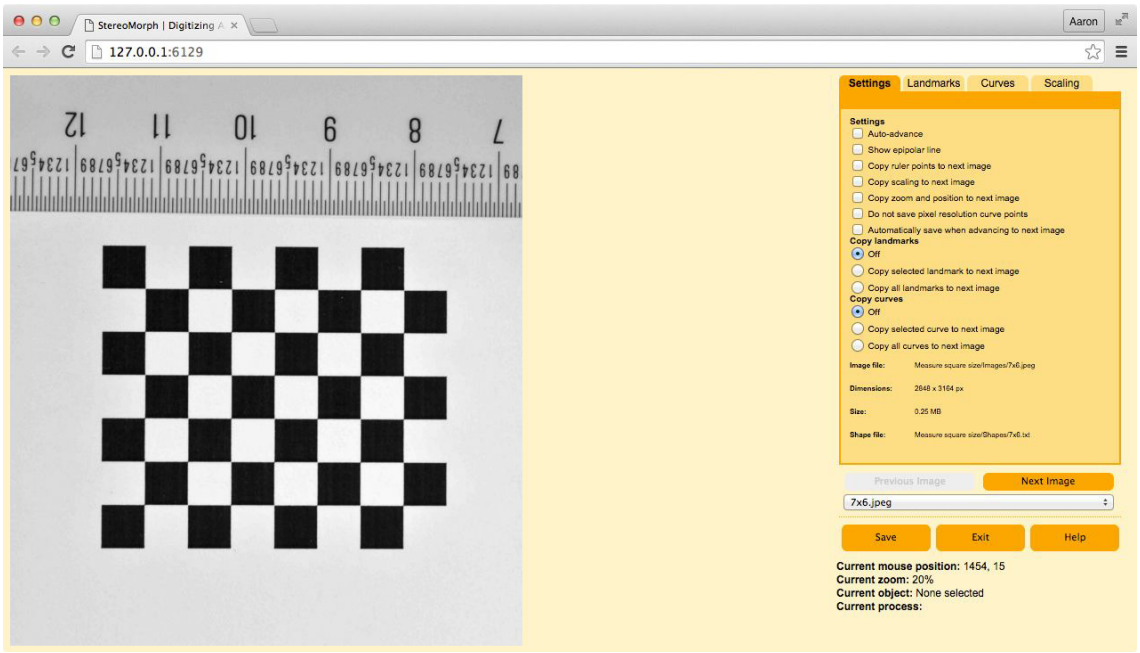
4. Load the StereoMorph library if it isn't already loaded and ensure that the tutorial folder is your current working directory.

```
> library(StereoMorph)
```

5. All of the remaining steps will be performed within the StereoMorph digitizing application. This is a web browser-based application for manually digitizing landmarks and curves in photographs.

The application is launched from R and opens in your default web browser; you do not need to be connected to the internet to launch the app (it runs on an internal server). To launch the app, use the function digitizeImages() with the following parameters: *image.file* (an image or folder of images to be digitized) and *shapes.file* (a file or folder where the digitized data will be saved). Using the tutorial project files the function call looks like this:

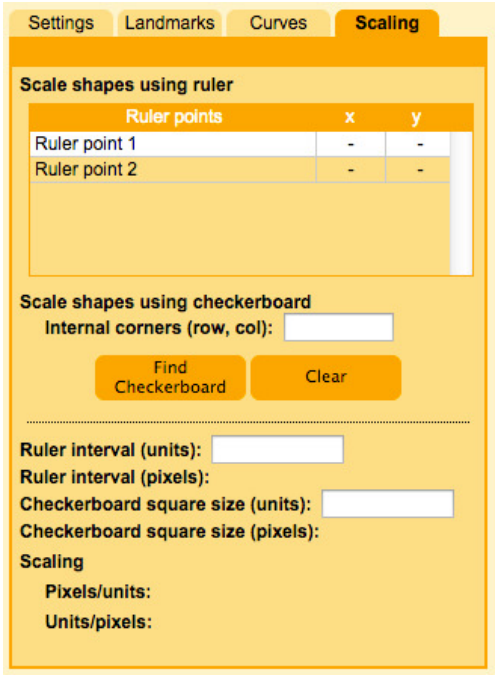
```
> digitizeImages(image.file='Measure square size/Images',
     shapes.file='Measure square size/Shapes')
```



The StereoMorph digitizing app.

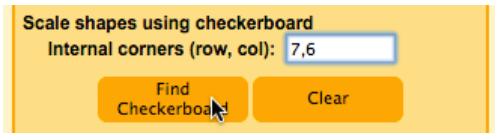
Once the app launches you should see the image on the left side of the window and a control panel on the right side. The Digitizing photographs section will explain the features of the app in more detail. For this section we will just measure the checkerboard square size and digitize points on the ruler.

6. Click on the "Scaling" tab in the right upper corner of the window.



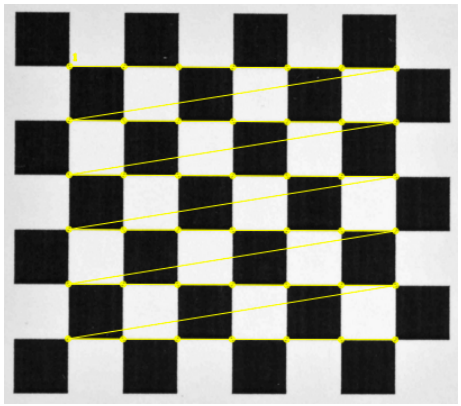
The Scaling panel of the digitizing app.

7. In the text field to the right of "Internal corners", enter the number of internal corners along each dimension of the checkerboard, separated by a comma, and click "Find Checkerboard".



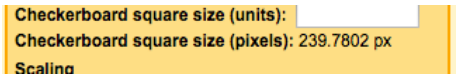
Entering the number of internal corners.

The StereoMorph function findCheckerboardCorners() will then automatically detect the internal corners of the checkerboard. This can take up to 20-30 seconds depending on the size of the image. Once these have been detected, yellow dots and lines will be displayed on top of the image with a small "1" indicating the first corner.

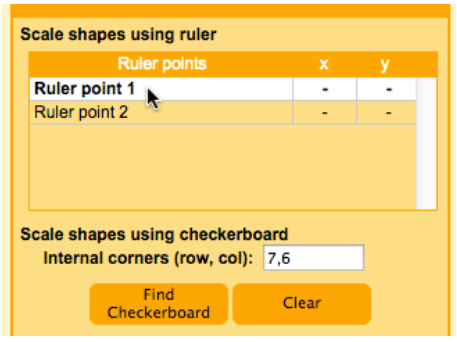


The detected corners displayed on top of the image.

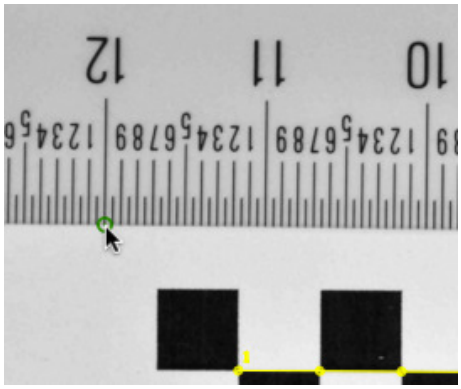
Note that for measuring the square size, the order in which the corners are detected doesn't matter. However, the order will become important later during the calibration step. Also, you'll see in the Scaling panel that the square size (in pixels) is measured once the corners are detected.



8. To get the square size in millimeters, all we need now is the conversion factor from pixels to millimeters (i.e. how many millimeters correspond to a single pixel in the image). Select "Ruler point 1" in the Scaling panel in order to set this as the current landmark.



9. Then move your cursor to the ruler mark at 12 cm in the image and double-click. This will create a landmark at that position.

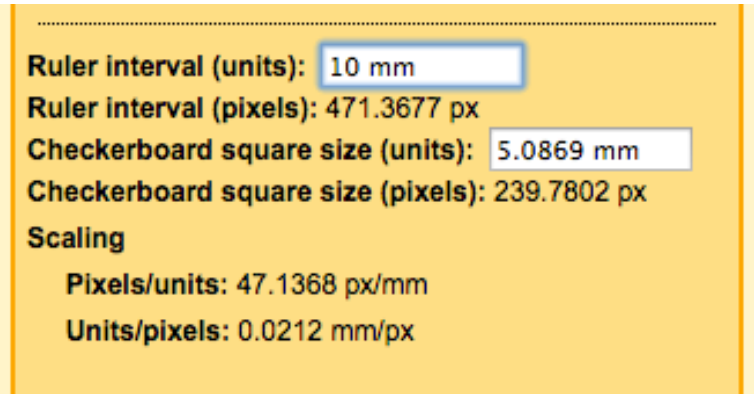


Double-click to add a landmark.

You can zoom in and out of the image by scrolling and you can move around the image by clicking and dragging the image with the mouse. If you want to change the position of the landmark, first make sure the landmark is selected by double-clicking on the landmark or clicking on the row in the Scaling panel (the landmark will turn from blue to green). Then click and drag the landmark with the mouse or use the arrow keys to move by a pixel at a time. To delete a landmark, first select the landmark and then press 'd'.

10. Digitize an addition 5 ruler points, selecting them in the Scaling panel and then placing them at the marks for 11, 10, 9, 8 and 7 cm. Every time you add a ruler point, the app will automatically create a new row in the ruler point table. The app will also continually update the corresponding ruler interval (in pixels) as you add or change ruler points.

11. In the Scaling panel, enter the distance between each consecutive ruler point with the units, in this case 10 mm. The app will calculate the checkerboard square size and this will be displayed to the right of "Checkerboard square size (units)".



A measured checkerboard square size of 5.087 mm.

In this case, the measured square size is 5.0870 mm (your results will likely differ a bit because you are unlikely to digitize the exact same pixel coordinates). Note that this differs by only 0.007 mm (7 microns) from what we expected based on the DPI and printer scaling. If you select a different ruler interval, select different points on the ruler, etc. you are likely to get a slightly different measurement but in general these measures should not differ by more than 0.2% of the square size. It's useful to write this on the front of the checkerboard in light pencil so that the square size can be read directly from images during the calibration step.

Note also that in addition to calculating the size of the checkerboard, the scaling panel tells you the size of each pixel in the units you specify. In this case, each pixel is about 0.021 mm wide (21 microns). You won't be able to measure the checkerboard square size much below this threshold since you can't digitize at a resolution smaller than a pixel. However, by sampling many points on the ruler it's possible to achieve a slightly higher resolution than the pixel resolution. We'll see later that the checkerboard corner detection also uses local image sampling around the corner to achieve a resolution greater than the pixel resolution.

12. Click "Save" to save the checkerboard corners, ruler points and scaling data to the file you specified using the *shapes.file* parameter. If you'd like to refer back to the square size measurement you can read either re-launch the digitizing app using digitizeImages() with the same input parameters (all of the saved data will be loaded in) or you can read the shape file directly. StereoMorph uses a custom XML-like format to save these data; you'll find the checkerboard square size saved within the "square.size" tag in the shapes file.

5 Arranging the cameras

The basic idea behind a stereo camera setup is to use information from two different views of the same object in order to reconstruct features visible in both views into 3D. A single view or image of an object has a information on the shape of features along the two dimensions of the image plane but not along the third dimension (going into and out of the image plane). Thus, by combining the information from two different views (and assuming that the cameras have been calibrated), the full three-dimensional shape of a feature can be reconstructed.

There are a number of ways to arrange cameras in stereo. This section will outline some general considerations to help you determine which camera arrangement will work best for your application. Once you've found a configuration that works well for your application, make sure you check its accuracy (either using the calibration images or a separate set of images) before collecting data.

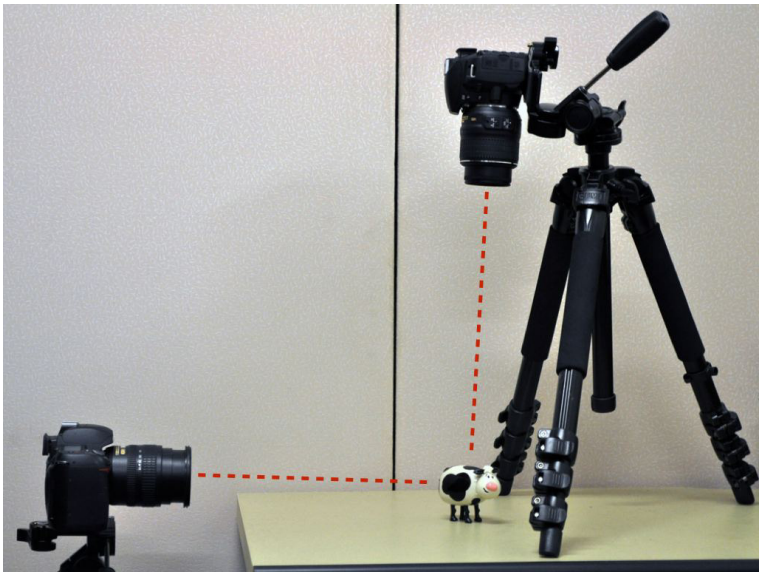
Materials needed for this section:

- 2 cameras (see Choosing your cameras)
- Camera remotes
- 2 sturdy tripods
- Tape

General considerations in arranging the cameras:

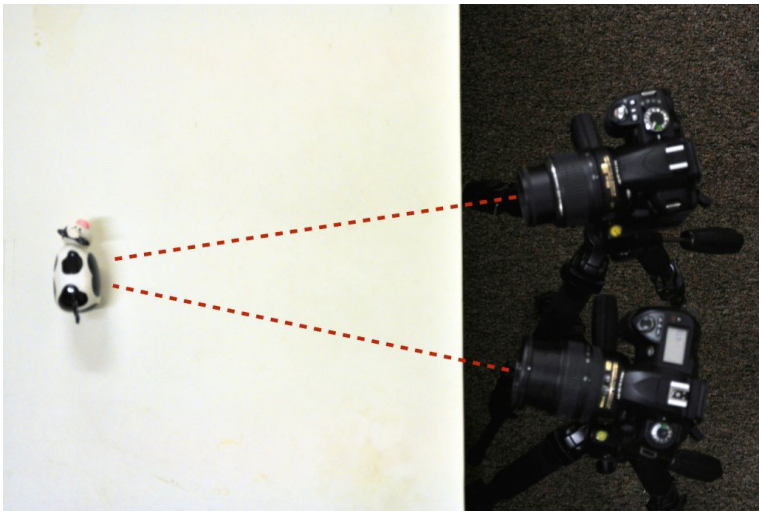
1. The views among the cameras must overlap such that the feature or features of interest are visible in both camera views. If you want to collect data on features around an entire object (for example the top, side and bottom of a skull) you can rotate the specimen and take two or more pairs of photographs of the same object. You'll end up with two or more separate sets of landmarks and/or curves. As long as there are three or more landmarks common among the different sets these can be aligned, or "unified", based on these common landmarks into a single set. StereoMorph has a function that will do this automatically, which will be detailed in Unification of reconstructed sets.

2. Theoretically, there is a trade-off between the ease of digitizing and reconstruction accuracy. For instance, if the angle between two camera views in a stereo setup is 90 degrees,



Two-camera stereo setup with the cameras at 90 degrees relative to one another.

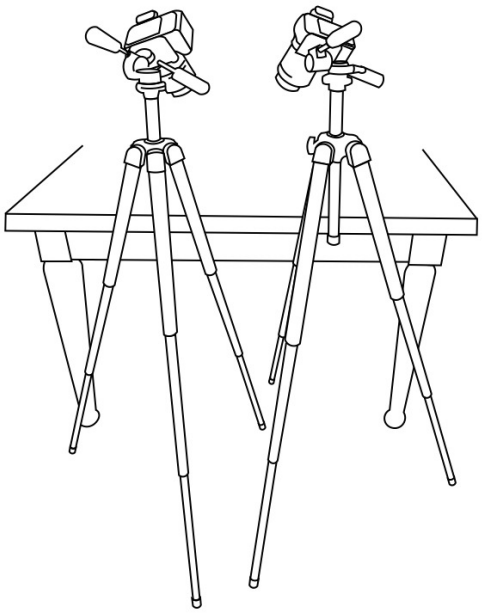
you will have high reconstruction accuracy (together, the two views give you full information on a point's position along all three axes) however the views will be so divergent that it will be difficult to identify the same point in both views. A point visible in one view may not even be visible in the other. If the angle between two cameras is reduced to around 20 degrees



Two-camera stereo setup with the cameras at 20 degrees relative to one another.

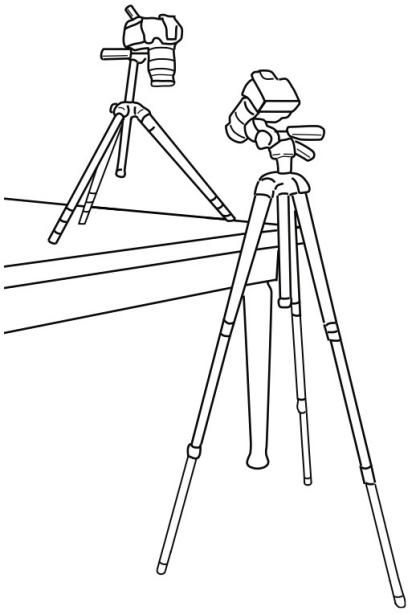
it's much easier to find the same point in both views (the views are nearly the same), however these slight differences in position are now the only information available on the point's position along the depth axis (orthogonal to the image planes). In practice, I've found that cameras positioned with a small angle relative to one another still provide high reconstruction accuracy but do not work as well for curve reconstruction. It's best to start with the cameras as close together as possible (more convergent views), test the accuracy and make the views more divergent if the accuracy is worse than what you're willing to accept.

The cameras were arranged as shown below for the accompanying example project.



One possible camera arrangement.

This is a nice setup because the orientation of both views is the same. By moving the cameras and changing the angle you can change the extent of divergence between the views. Also the table provides an easy place to set the objects and lights. An alternative setup is to have one camera on the tabletop and the other camera on the floor.



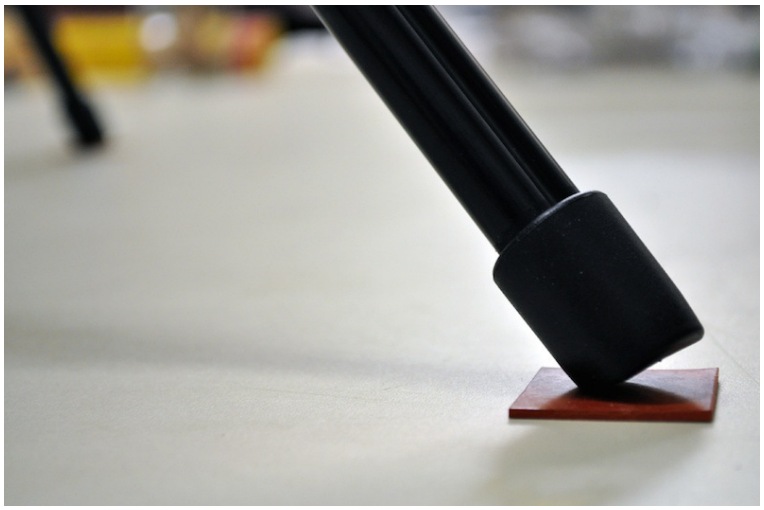
An alternative camera arrangement.

A disadvantage to this setup is that the view from one camera will be "upside-down" relative to the other. You can flip all of the images once they're loaded on your computer

to compensate for this.

3. It is *essential* that the cameras not move during the entire process of calibration and photographing specimens. The cameras can be calibrated before or after data collection but throughout and between these steps the cameras must remain in the exact same position. Because the camera is often positioned half a meter or more away from the object, even a small shift of the camera can translate into a large shift in the image frame, causing large inaccuracies.

If you position a tripod on a smooth surface, such as a table top, put small rubber squares under each tripod foot to keep the tripod from slipping.



4. Before you take any photographs, you can attach small pieces of tape to the tabletop or some other fixed surface within the calibration space and visible from both camera views.

By taking a photograph from both views before you begin and then after you have finished photographing all objects with a particular calibration you can compare the before

and after photographs to be sure that both cameras remained motionless throughout the photographing process.

5. Just as the cameras should not move during the entire process of calibration and object photographing, the camera settings themselves should not change. This includes the zoom (focal length) and the focus. The calibration is specific to a particular focal length and focus, so if either of these changes the calibration will no longer be accurate. Additionally, if your lens has vibration reduction (VR) you should turn this off. Vibration reduction uses a small gyroscope in the lens to compensate for camera motion and reduce blur. The spinning and stopping of the gyroscope can cause the image frame to shift randomly while taking photos.

Turn off auto-focus and vibration reduction, if applicable.

6. It's best to use a remote (wireless or cord) to release the shutter so you minimizing touching the shutter button on the cameras as much as possible.

Shutter remotes lessen the chances of the cameras moving during data collection.

I've found that pressing buttons on the camera lightly (such as for reviewing photos) doesn't cause significant movement of the cameras but pressing the shutter button requires more force and doing it repeatedly causes the cameras to move significantly over a series of photographs. A wireless remote is the best option since you can trigger both cameras with one remote (note that if the objects you are photographing are not moving the photographs do not have to be taken simultaneously).

7. Make sure that all connections and screws in the tripod and between the tripod and the camera are tight. This reduces the possibility of any motion of the cameras during data collection.

Ensure tight connections in the tripod and between the tripod and camera.

8. Set the cameras to the smallest aperture (this is the largest f-value).

A smaller aperture is ideal because it increases depth of field. Without increasing the lighting, the exposure time will increase.

The smaller the aperture, the greater the depth of field (i.e. the more things are in focus both close and far away from the camera). This is essential in a stereo camera setup because in order to digitize points accurately throughout the calibration volume they must be in focus.

9. If possible, set your camera to manual mode.

Manual mode on a Nikon.

This allows you to control both the aperture and shutter speed. Once you have the cameras arranged take some sample photos of the objects to find a good shutter speed (exposure) for your lighting. I've found that the automatic exposure on my camera is not always reliable and that it's better to simply have the same exposure throughout.

6 Calibrating stereo cameras

In order to perform stereo camera reconstruction we need a mathematical formula or model that relates particular combinations of 2D pixel coordinates from each view to 3D coordinates. The mathematical model used by StereoMorph is the DLT model (direct linear transformation; Abdel-Aziz & Karara 1971). With the DLT method, each calibrated camera has a set of 11 coefficients that relate each unique 3D coordinate in the calibration space to their corresponding (non-unique) 2D pixel coordinates in that particular camera view; modified forms of DLT use additional coefficients to account for lens distortion but StereoMorph uses just 11 (see Choosing lenses for a more thorough discussion of lens distortion).

Note that if you only have the coefficients for a single camera view you can only go one way: you can only project 3D points to pixel coordinates, not the other way around. This is because a point with particular pixel coordinates in an image can fall anywhere along a line in 3D space. But if you have the DLT coefficients from two or more camera views you can combine the pixel coordinates of a single point in both views from multiple camera views to find the corresponding 3D coordinate. You can think of this as finding the intersection of the two lines from each camera view in 3D space. Thus, the objective of the camera calibration step is to determine these 11 DLT coefficients for each camera.

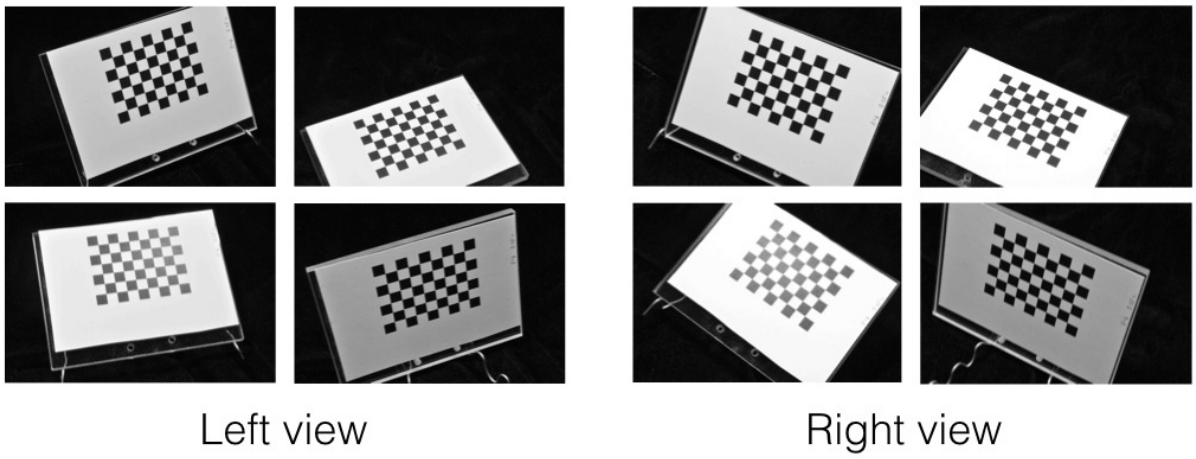
Just as there are multiple mathematical models for 3D reconstruction there are also multiple methods with which to determine the DLT coefficients. Typically, DLT coefficients are determined using what's referred to as a calibration or reference object. This is usually a 3D, cube-shaped structure filled with markers having known 3D positions relative to one another. The markers are digitized in each camera and the set of corresponding 2D and 3D coordinates are used to calculate the DLT coefficients.

Because of the difficulties in designing and building a 3D calibration, and the time-consuming task of digitizing the calibration cube markers, StereoMorph determines the DLT coefficients using the internal corners automatically detected from a checkerboard pattern. A checkerboard is photographed from both camera views at different positions and angles within the calibration space. Rather than estimate the DLT coefficients directly, StereoMorph estimates the six transformation parameters (3 translation, 3 rotation) required to transform the first checkerboard into each subsequent checkerboard in 3D space by minimizing the reconstruction error. These transformation parameters are then used to generate 3D coordinates (the equivalent of a calibration object) and calculate the DLT coefficients.

This section will show you how to use the function calibrateCameras() to calibrate stereo cameras using a checkerboard.

6.1 Photographing a checkerboard

Once you have arranged your cameras, take 8-10 photographs of the checkerboard at different positions and angles within the calibration space.

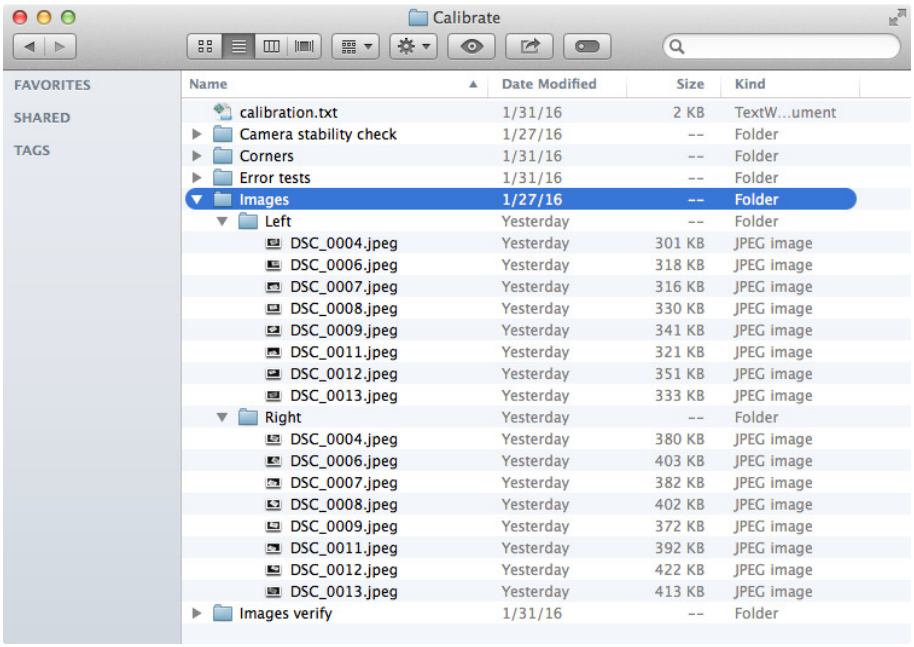


Examples of a checkerboard photographed from two views at different positions and angles within the calibration space.

Be sure that you position the checkerboard at **different positions throughout the calibration space and at different angles**. This provides a sampling of points throughout the space for the calibration. If you only photograph the checkerboard in one area of the calibration volume, reconstruction errors could be relatively higher in other areas. Similarly, if you only photograph the checkerboard at a particular angle (e.g. 45 degrees), you won't have good sampling of points along each dimension of the space (since a checkerboard is a flat surface it can only sample two dimensions at any one time). This can causes reconstruction errors to be higher along particular dimensions than along others.

Import the calibration images into a folder, separating the images from different views into two different folders (e.g. left and right, View 1 and View 2). Note that if you are also photographing objects and you're transferring images via a camera memory card it's best to wait until you've taken all the photographs before upload the calibration images - taking the memory card out of the camera risks moving the cameras in which case they'll no longer be properly calibrated. If you are using a cord to upload the images then you can upload the calibration images, check the calibration and proceed with photographing your objects.

In the tutorial project the calibration images are saved in "Run 1/Calibrate/Images".



Load the StereoMorph library into the current R session and ensure that the StereoMorph Tutorial folder is your current working directory.

```
> library(StereoMorph)
```

6.2 Estimating the calibration coefficients

Then call the calibrateCameras() function to perform the camera calibration. As of StereoMorph v1.5 all of the calibration steps can be performed within calibrateCameras(). These steps include: checkerboard corner detection, calibration coefficient estimation and calibration error assessment (based on all of the calibration images). The tutorial project function call looks like this:

```
> cal_cam <- calibrateCameras(img.dir='Run 1/Calibrate/Images',
    sq.size='6.35 mm', nx=8, ny=6,
    cal.file='Run 1/Calibrate/calibration.txt',
    corner.dir='Run 1/Calibrate/Corners',
    verify.dir='Run 1/Calibrate/Images verify')
```

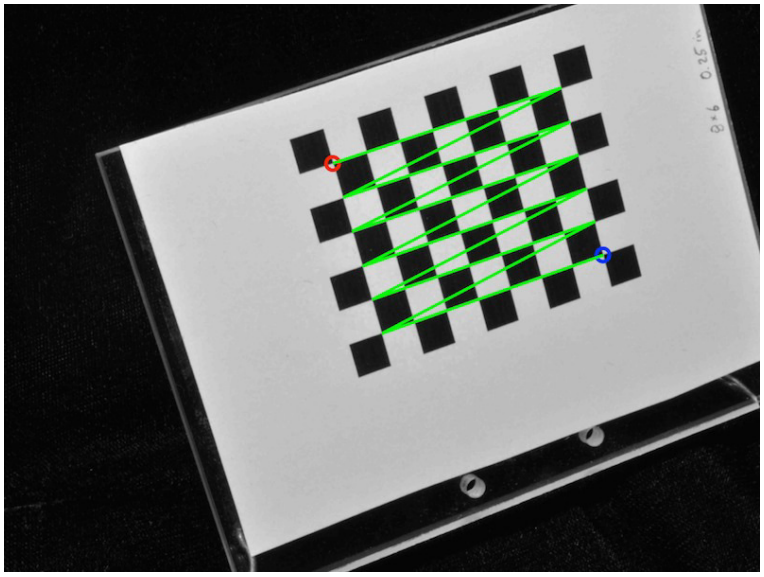
There are quite a few basic parameters here, let's unpack them.

- *img.dir*: The file path to the folder containing the calibration images, each view in a separate folder.
- *sq.size*: The size of the squares along with the units (length along any one side).
- *nx*: The number of internal corners along one dimension (the choice of which is *nx* and *ny* is arbitrary but must be consistent throughout).

- *ny*: The number of internal corners along the other dimension.
- *cal.file*: A file path to a '.txt' file where the calibration results will be saved. This file does not have to exist, a new file will automatically be created if one doesn't already exist.
- *corner.dir*: A file path to a folder where the corners will be saved. If this folder does not exist, a new folder will automatically be created.
- *verify.dir*: (Optional) A file path to a folder where images will be saved that show the detected corners. If this folder does not exist, a new folder will automatically be created.

If you run this for the tutorial project the function will access the existing "calibration.txt" file and prompt you sequentially whether you would like to repeat any of the steps of the calibration. You can simply enter 'n' at each prompt and the function will print the calibration errors using all of the calibration images.

To re-run all of the steps of the calibration (as if you were calibrating a new set of images) delete the "calibration.txt" file and the "Corners" folder in "Calibrate". Then call calibrateCameras() again (in this case the function will not issue any prompts). The function will begin detecting the corners in all of the calibration images. You can see the detected corners by looking in the "Images verify" folder.



Verify image to check the order of the detected corners. The red circle indicates the first corner and the blue circle indicates the last corner.

The corners are returned in particular order and this order is important for the calibration to work properly. In the verify image the first corner will be indicated by a red circle and the last by a blue circle. A green line connects these two circles showing how the corners are ordered between the first and the last (think RGB). The corner detection will always return the corners ordered along the *nx* direction first and the *ny* direction

second. However, the first corner (red circle) may not necessarily be the same between two images if the checkerboards are in very different orientations (especially if one is upside down relative to the other).

Be sure to **look through the verify images and check that the function is detecting the corners in the same order for all of the calibration images** (including across both views). If your cameras are arranged such that one view is upside-down relative to the other, set the *flip.view* parameter to TRUE in the calibrateCameras() call, as shown below. Note that if you set *flip.view* to TRUE and use the tutorial project calibration images the calibration will not work since the corner order will not be consistent between the two views.

```
> # For flipped views:
> cal_cam <- calibrateCameras(img.dir='Run 1/Calibrate/Images',
     sq.size='6.35 mm', nx=8, ny=6, flip.view=TRUE,
     cal.file='Run 1/Calibrate/calibration.txt',
     corner.dir='Run 1/Calibrate/Corners',
     verify.dir='Run 1/Calibrate/Images verify')
```

Based on the function read-out we can see that the checkerboard detection was successful for all the images:

```
Calibration checkerboard corner detection...
    Running automated checkerboard corner detection on calibration images...
        DSC_0004.jpeg
            View 1 : 48 corners found
            View 2 : 48 corners found
        DSC_0006.jpeg
            View 1 : 48 corners found
            View 2 : 48 corners found
        DSC_0007.jpeg
            View 1 : 48 corners found
            View 2 : 48 corners found
        DSC_0008.jpeg
            View 1 : 48 corners found
            View 2 : 48 corners found
        DSC_0009.jpeg
            View 1 : 48 corners found
            View 2 : 48 corners found
        DSC_0011.jpeg
            View 1 : 48 corners found
            View 2 : 48 corners found
        DSC_0012.jpeg
            View 1 : 48 corners found
            View 2 : 48 corners found
        DSC_0013.jpeg
            View 1 : 48 corners found
            View 2 : 48 corners found
```

The function will likely not detect the corners in all of your images. If the lighting is not ideal or a portion of the checkerboard is cut off, then the detection can fail. For this reason it is a good idea to take at least 10 calibration images, so that if a couple pairs

fail you will still have enough to get a good calibration.

Next, the function will begin the coefficient estimation. Because the estimation process does not always converge on the correct solution, the function divides up the detected corners into different sets and run separate calibrations for each set. It then saves the calibration with the lowest error. The number of sets that are created by default depends on the number of detected pairs. I'll refer to each pair of images as an aspect, since they are different aspects of the same checkerboard. If corners are detected in 8 or more aspects the function will randomly divide up these 8 images into 3 sets of 6 pairs. If corners are detected in 6 or 7 aspects, there will be 2 sets of 5 aspects each. And if the corners are detected in 5 aspects or less, the function will use a single set of all the available aspects.

The parameters that control this are:

- *num.sample.est*: the number of aspects overall that will be used in estimating the coefficients
- *num.sample.sets*: the number of unique sets that will be tried
- *num.aspects.sample*: the number of aspects in each set

The function prints these parameters before the estimation step. For the tutorial project that looks like this:

```
Calibration coefficient estimation...
    Number of aspects from total that will be sampled calibration coefficient estimation (num.sample.est): 8
    Number of unique sets of aspects to try (num.sample.sets): 3
    Number of aspects to sample for each set (num.aspects.sample): 6
    Aspects in each sets:
        1: DSC_0004, DSC_0007, DSC_0008, DSC_0009, DSC_0012, DSC_0013
        2: DSC_0004, DSC_0006, DSC_0007, DSC_0008, DSC_0009, DSC_0011
        3: DSC_0006, DSC_0007, DSC_0009, DSC_0011, DSC_0012, DSC_0013
    Use calibration aspects to determine the best calibration set?: TRUE
```

The default options should work well for most cases but you can set each of these parameters yourself as input parameters to calibrateCameras().

Before the coefficient estimation the function reduces the number of checkerboard corners to speed up the optimizations. This is done by fitting a perspective model to the corners and resampling to 9 corners (3x3). Once this is complete, the function begins the optimization step with each set of aspects. The optimization proceeds by sequentially adding and estimating the transformation of each aspect within the set. Because the number of parameters increases as the number of aspects increases, the optimization will take progressively more time up to the maximum number of aspects.

The optimization should generally converge on a value (reconstruction error) less than 1 for each minimization (indicated by the value to the right of "minimum"). This value is in pixels and is the root mean squared error that results when each corner is reconstructed

into 3D using the DLT coefficients and then projected back into pixel coordinates in each view. So regardless of the scale of your setup this error should be less than 1. If for some reason a particular set is not converging the function might stop running the minimization and switch to the next set. If the optimization fails to converge for all of the sets there is probably an error in the correspondence between the two corner sets. Most commonly this is due to corners that are in different orders between two views or between different aspects.

```
Selected aspect set (lowest reconstruction error): 3
    Aspects in set: DSC_0006, DSC_0007, DSC_0009, DSC_0011, DSC_0012, DSC_0013
Mean reconstruction RMS Error: 0.1013 px
DLT Coefficient RMS Error:
    Left   : 0.1937 px
    Right  : 0.222 px
```

The DLT coefficients corresponding to the optimization with the lowest error will then be saved in the enquotecalibration.txt file as an 11x2 matrix.

6.3 Determining the calibration accuracy

Once the best calibration is selected, the calibrateCameras() tests the calibration accuracy using all of the available calibration checkerboards. Note that these errors are measured using the same checkerboard that was used for the calibration. Therefore, they cannot be used to test whether the scaling of the calibration is correct. To test the accuracy of the calibration including scaling it's best to test the accuracy using an additional checkerboard with a different square size. The error diagnostics are the same in both cases so they will be described in this section.

After the coefficient estimation, calibrateCameras() prints a "dltTestCalibration Summary".

```
dltTestCalibration Summary
    Number of aspects: 8
    Number of views: 2
    Square size: 6.35 mm
    Number of points per aspect: 48
    Aligned ideal to reconstructed (AITR) point position errors:
        AITR RMS Errors (X, Y, Z): 0.01379392 mm, 0.01177767 mm, 0.02260367 mm
        Mean AITR Distance Error: 0.02573147 mm
        AITR Distance RMS Error: 0.02917404 mm
    Inter-point distance (IPD) errors:
        IPD RMS Error: 0.01778047 mm
        IPD Mean Absolute Error: 0.01412996 mm
        Mean IPD error: -0.001119889 mm
    Adjacent-pair distance errors:
        Mean adjacent-pair distance error: -0.001215635 mm
        Mean adjacent-pair absolute distance error: 0.01656301 mm
        SD of adjacent-pair distance error: 0.01934579 mm
    Epipolar errors:
        Epipolar RMS Error: 0.2463803 px
        Epipolar Mean Error: 0.2463803 px
        Epipolar Max Error: 1.522425 px
        SD of Epipolar Error: 0.2082847 px
```

This read-out summarizes four main error measurements:

- *Aligned ideal to reconstructed (AITR) errors*: The AITR error aligns an ideal (perfect) checkerboard to the reconstructed corners using least squares alignment. Then, the distance is measured between each ideal checkerboard corner and its corresponding reconstructed corner. If the corners were perfectly reconstructed, the ideal and reconstructed points would overlap perfectly. The "AITR RMS errors" are the alignment errors along each dimension (x,y,z) in the coordinate system of the calibration points. This coordinate system depends on the orientation of the first checkerboard so it is somewhat arbitrary. But if you have larger error along one dimension than another it will generally show up here. For the tutorial project the error is greatest along the z-axis but overall the mean errors are low (less than 25 microns along any dimension).

- *Inter-point distance (IPD) errors*: The IPD error summarizes distance rather than positional errors. For every reconstructed checkerboard, random pairs of points (without re-sampling) are chosen and the distance between them is compared to the distance on an ideal checkerboard. Unlike AITR error, this measure doesn't allow a comparison of the error along different dimensions. Additionally, while a range of different lengths are used to calculate the error the lengths can only be as large as the checkerboard. The IPD error should be about the same order of magnitude as the AITR error. For the tutorial project the IPD error is less than 20 microns. The reconstructed distances can be either shorter or longer than the actual distance. The "Mean IPD error" takes the simple mean of these errors: If there is no bias toward over- or underestimation of distance this should be near zero.

- *Adjacent-pair distance errors*: This is identical to IPD error except that the error is determined only using pairs of adjacent checkerboard corners. This means the ideal distances are uniform and the same size as the square size. Since the corners in each pair are uniformly close together, their mean position (the mid-point) can be used to look at how IPD error varies as a function of position in the calibration volume.

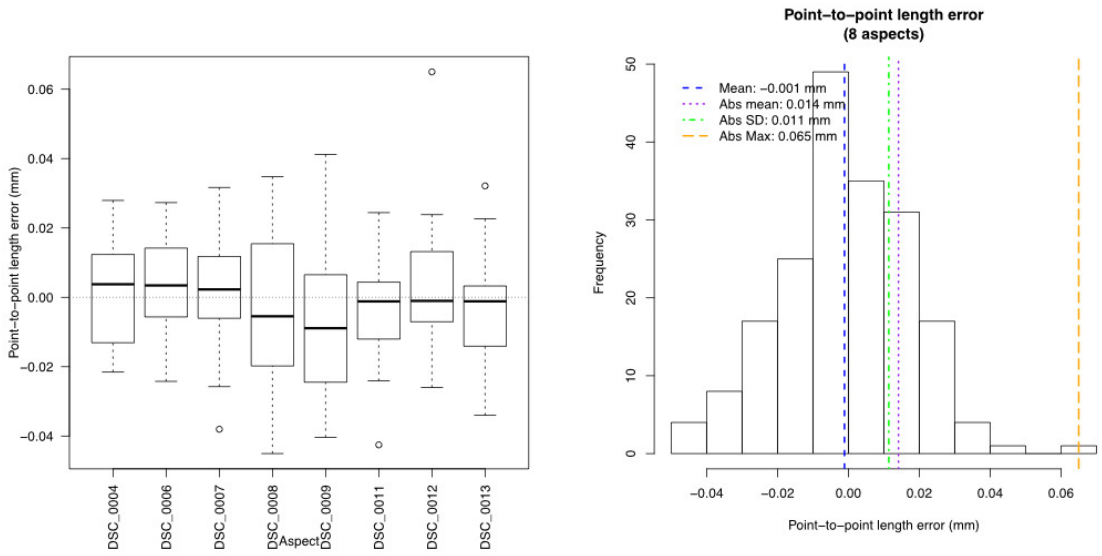
- *Epipolar errors*: For two cameras arranged in stereo a point in one camera view must fall along a line in a second camera view. This line is that point's epipolar line. The distance between a point's epipolar line and its corresponding point in that second camera view is the epipolar error. Since the error is a measure of distance between a line and point in the image plane the units are pixels.

It's important to consider the magnitude of these errors relative to the pixel resolution of the cameras and the size of the calibrated volume. For the tutorial project the image resolution is around 30 microns/pixel (using 12 MP cameras). Since digitized coordinates are limited to pixel resolution the reconstructed errors should always be greater 30 microns. Note that the calibration errors can be lower than this because the checkerboard corners are detected to subpixel resolution by sampling a small window of pixels around

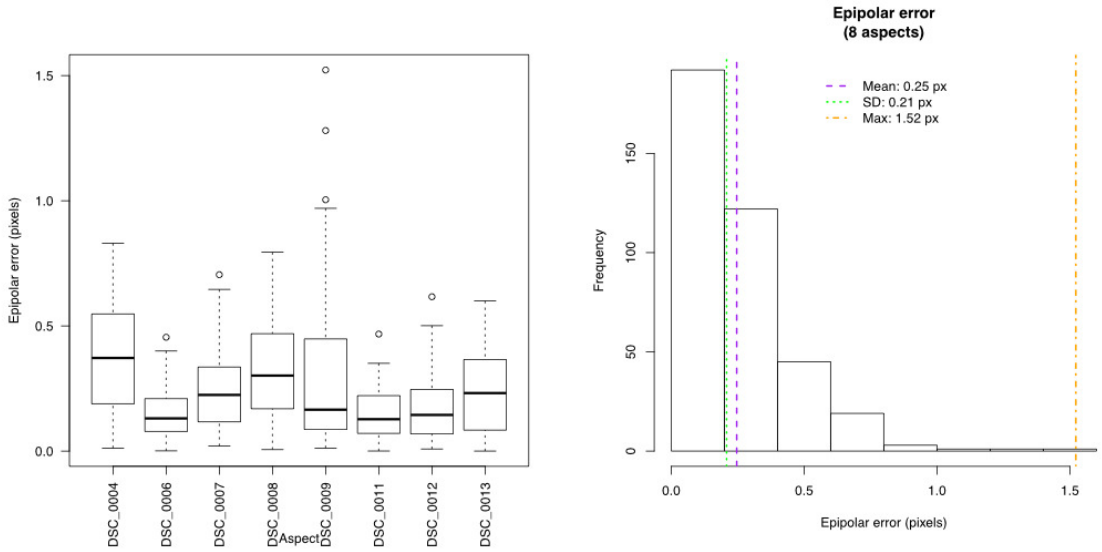
each internal corner and fitting a sub-pixel corner position. Also, the calibrated volume is approximately 60 mm x 80 mm x 100 mm. Thus, the mean distance (IPD) error is less than 0.03% of the length along any one dimension of the space.

In addition to the error summary read-out, calibrateCameras() creates several error diagnostic plots for a complete assessment of the error. These will be saved in a folder "Error tests" within the same folder as the "calibration.txt" file (if the "Error tests" folder doesn't exist before running calibrateCameras() one will be created). The plots for the tutorial calibration can be found in "Run 1/Calibrate/Error tests".

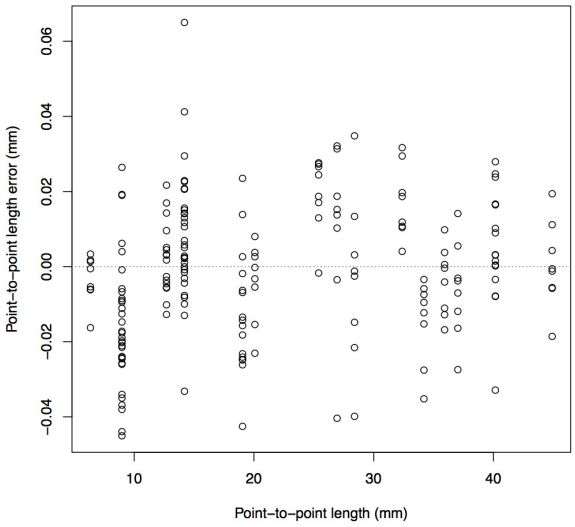
For example, the function creates a boxplot of the IPD errors, separated by aspect (left) and a histogram of the IPD errors (right) pooled from all aspects.



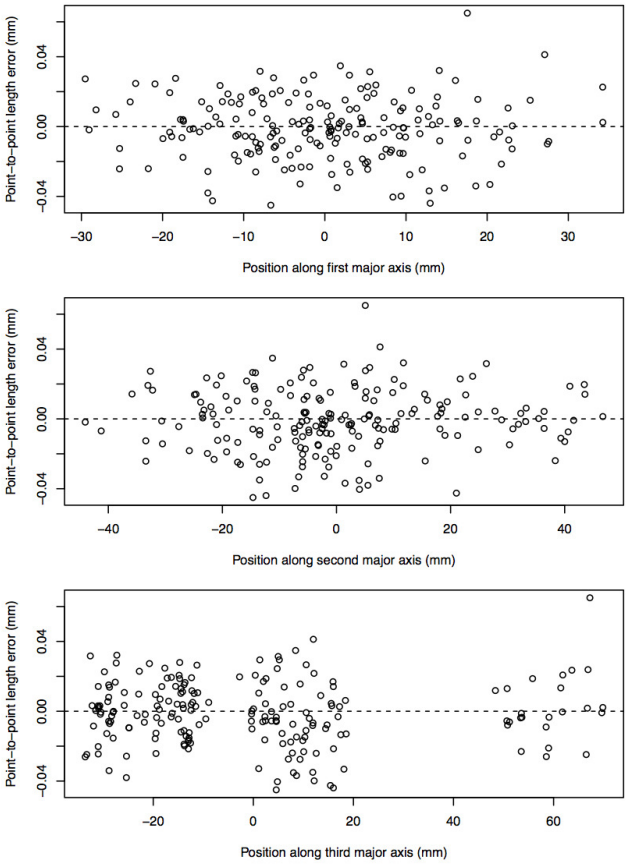
And the equivalent plots for epipolar error:



The function creates a plot of the IPD error as a function of the length between the two corners being measured to verify that error is not strongly correlated with the length being measured.



The function also uses all the reconstructed corners to identify the major axes through the calibration volume and plots the IPD error as a function of the position along each of these major axes. This is an easy way to get an idea of the size of the total calibrated volume and check whether the error is greater along one dimension than another.



7 Photographing objects

Now that your cameras are calibrated you can begin photographing objects. This is the step where the DLT method has an advantage over other methods for 3D data collection. Once the cameras are calibrated, the number of objects that can be photographed is only limited by the time it takes to position and photograph each object. Thus, you can essentially cycle the objects through the calibrated space. Note that you can calibrate the cameras after taking photographs of the objects - just so long as the cameras remain motionless throughout.
It is best to have a uniform background that provides good contrast to your specimen. First, this can decrease the photo size by as much as half (encoding a large black space takes up less space than a multi-colored, noisy background). Second, it's easier to discern points on the edge of the specimen when the edge is clearly distinguishable from the background. For light-colored specimens, black velvet works well. The cheapest material available at fabric stores works great and costs about $10 a yard.

Black velvet works great as a uniform black background.

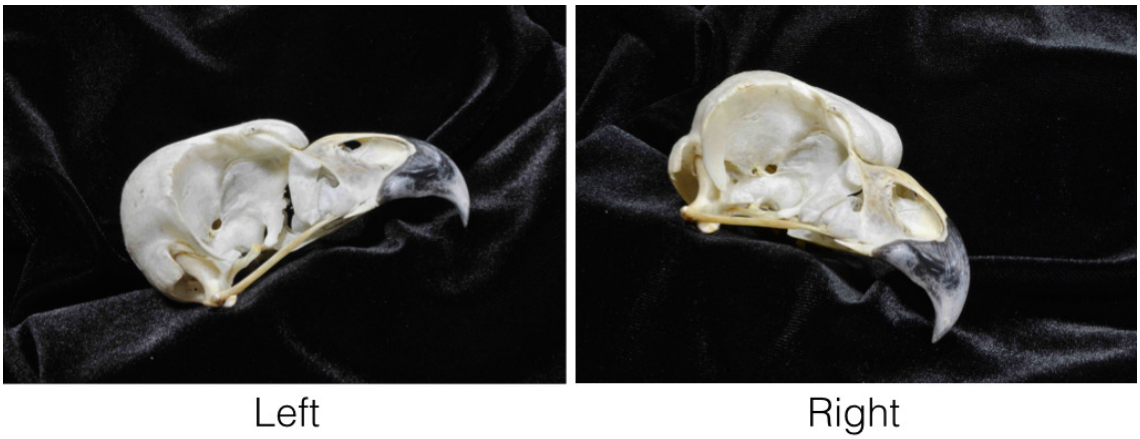
If you're collecting data on features around an entire object (for example the top, side and bottom of a skull) you can rotate the specimen and take two or more pairs of photographs of the same object (referred to here as "aspects"). You'll end up with two or more separate sets of landmarks and/or curves in different coordinate systems. These different sets can be aligned, or "unified", based on landmarks that are common between the sets to create a single set of landmarks and/or curves. StereoMorph has a function that will do this automatically, which will be detailed in Unification of reconstructed sets.

The tutorial project has photographs of the skulls of two different species of bird: a Great Horned Owl (*Bubo virginianus*) and an African Gray Parrot (*Psittacus erithacus*). These are specimens from the bird skeleton collection at the Field Museum of Natural History in Chicago. Within the tutorial folder project you'll find these in "Run 1/Reconstruct/Images". So that shape data can be collected from different areas around the skull there are 3 different aspects of each skull, with the filenames ending in "a1", "a2",

and “a3”.

The first aspect shows the area underneath the skull most clearly,

the second aspect shows the side of the skull most clearly,

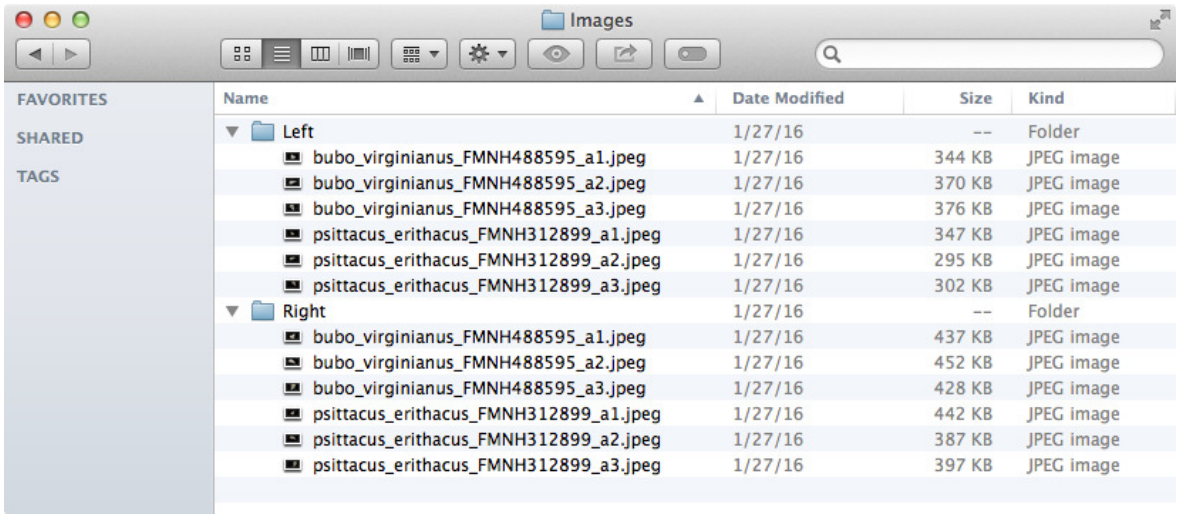


and the third aspect shows the area around the ear most clearly.

Depending on the data you want to collect you might be able to get away with a single pair of images of each specimen (a single aspect). If you do use multiple aspects, you’ll

need some overlap in landmarks among the images (at least three, preferably five to six) in order to combine all of the points into a single 3D point set.

Once you've photographed the specimens, upload them (as .jpeg/.jpg files) into a folder, separated by view. Be sure to use the same names for each view as in the calibration step.



If you have multiple aspects and want them to be unified into a single set per specimen you need to follow a particular naming convention: each file should end with an "a" and the aspect number as shown above. Also, use only letters, numbers and underscores when naming the image files (no spaces).

8 Digitizing photographs

Now that you have your object image pairs it's time to identify the landmarks and/or curves that you want to reconstruct into 3D. For this we'll use the StereoMorph's digitizing app (used previously to measure the checkerboard square size). The app is launched from R but runs in your default web browser (you do not need to be connected to the internet to launch the app since it runs on an internal server). The app is fully functional across Safari, Chrome, Firefox and Opera.

8.1 Launching the digitizing app

Load the StereoMorph library if it isn't already loaded and if you are following along with the tutorial ensure that the tutorial folder is your current working directory.

```
> library(StereoMorph)
```

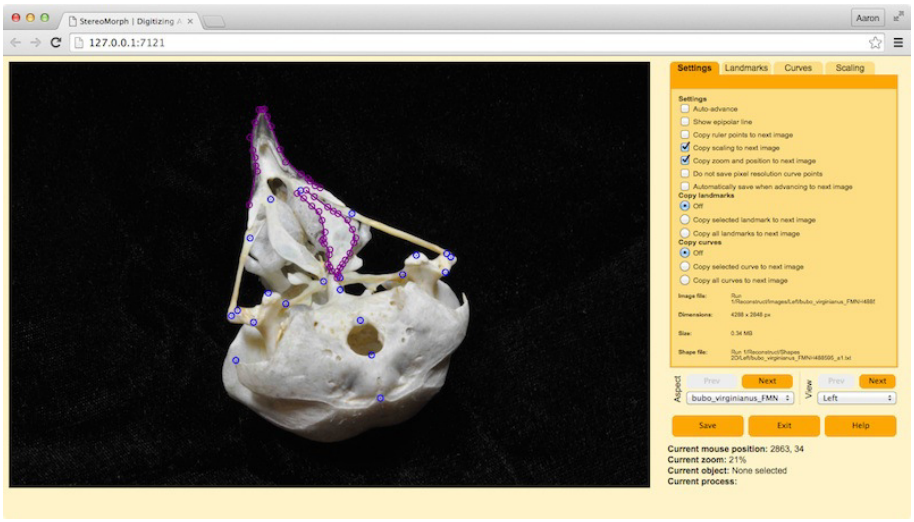
Launch the digitizing app from R using the digitizeImages() function. For the tutorial project the function call looks like this:

```
> digitizeImages(image.file='Run 1/Reconstruct/Images',
     shapes.file='Run 1/Reconstruct/Shapes 2D',
     landmarks.ref = 'Shapes ref/landmarks_ref.txt',
     curves.ref = 'Shapes ref/curves_ref.txt',
     cal.file='Run 1/Calibrate/calibration.txt')
```

These are the basic input parameters to digitizeImages() when using the app to digitize stereo image sets:

- *image.file*: File path to a folder containing the images to be digitized, separated by view into different folders.
- *shapes.file*: File path to a folder where the shape data will be saved. If this does not exist it will be created automatically with the same sub-folders as *image.file*.
- *landmarks.ref*: A .txt file or vector listing the names of the landmarks to be digitized. Landmark names should only contain letters, numbers and underscores (no spaces). If the input is a .txt file, each name should be on a separate line (refer to "Shapes ref/landmarks_ref.txt" in the tutorial project for an example).
- *curves.ref*: (Optional) A .txt file listing the names of the curves to be digitized with the start and end points of each curve. If not digitizing curves this can be omitted. These names should only contain letters, numbers and underscores (no spaces). Each curve should be on a separate line, with the curve name, start, and end point separated by tabs (refer to "Shapes ref/curves_ref.txt" in the tutorial project for an example).

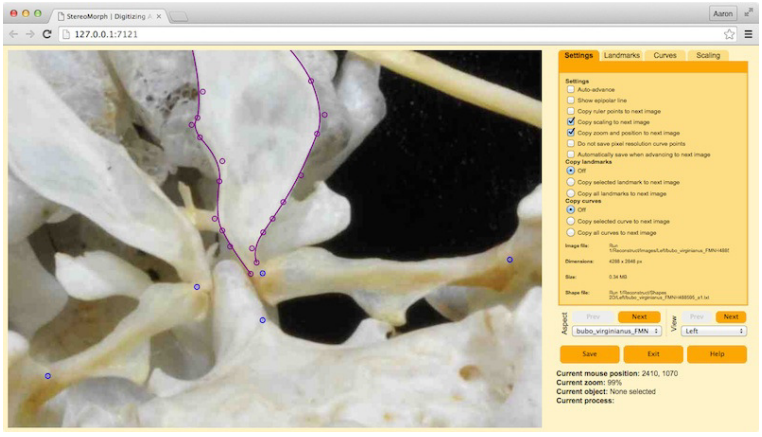
- *cal.file*: The calibration file created by calibrateCameras(). The DLT coefficients in this file will be used to draw the epipolar line onto the image when digitizing landmarks.



The StereoMorph digitizing app launched with a stereo image set.

The left two-thirds of the app are the image frame. This is where you can navigate around the image and add landmarks and curves using your mouse or trackpad. The right two-thirds are the control panel, for viewing and saving landmark/curve lists and navigating between images. There are four tabs in the control panel: Settings, Landmarks, Curves, and Scaling. The Settings tab contains different user-interaction options. These options will be saved using cookies so they do not have to be reset every time you open the digitizing app. The Landmarks and Curves tabs contain the pixel coordinates of all digitized landmarks and curve points. The Scaling can be ignored for stereo sets - it's used to scale coordinate data for 2D morphometrics.

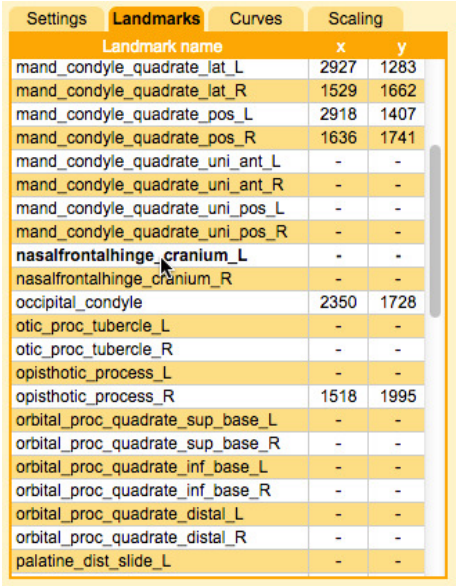
You can navigate around the image in the image frame using the same basic mouse actions as in Google Maps. You can zoom in and out by positioning your cursor over the image and scrolling. To move around the image click and drag the image.



Zoom in and out by scrolling over the image.

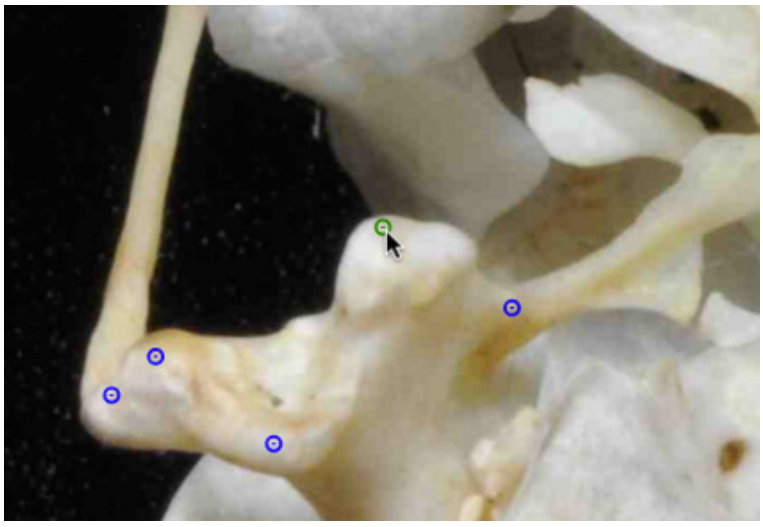
8.2 Digitizing landmarks

To digitize a landmark, click the corresponding row in the Landmarks tab. This will make that landmark the current object.



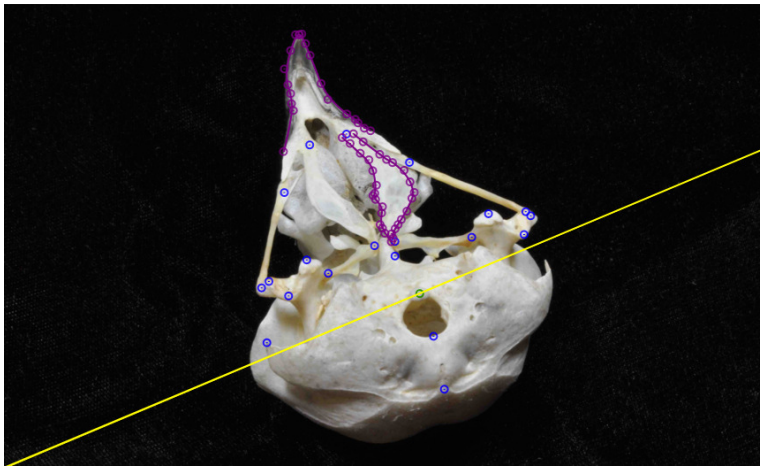
Move the cursor to where you want to add the landmark and double-click (keyboard shortcut: 'x'). If you've already placed a landmark you can also select it by placing your cursor over the landmark and double-clicking.

To move an already digitized landmark, first select it and then click and drag it with the mouse. You can also use the arrows on the keyboard to move in single pixel increments or hold shift to move in 10 pixel increments.



To delete a landmark, select the landmark and press 'd'. Landmarks with '-' values in the Landmarks panel will be ignored when saving.

For 3D reconstruction, you'll need to digitize the same landmark in both views. Since the views have different perspectives of the object sometimes it's difficult to identify exactly corresponding points. To help with this you can turn on the epipolar line. In the Settings panel click the box next to "Show epipolar line". If you select a landmark, and *if that landmark has already been digitized in the other view*, the epipolar line will be projected across the image.



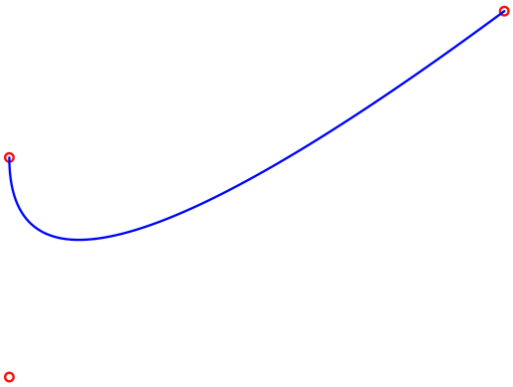
Epipolar line projected across the image for the landmark in green.

This line represents the line along which the landmark should fall if it corresponds exactly to the point digitized in the other view. Recall the epipolar error from the section on determining the calibration accuracy. The mean epipolar error using the checkerboard is typically less than 0.5 pixels and the maximum is typically less than 2 pixels. So assuming the calibration worked propertly, the epipolar line is a reliable aid in identifying corresponding points between views.

Once you've finished digitizing all of the shapes you'd like to digitize be sure to click "Save".

8.3 Digitizing curves

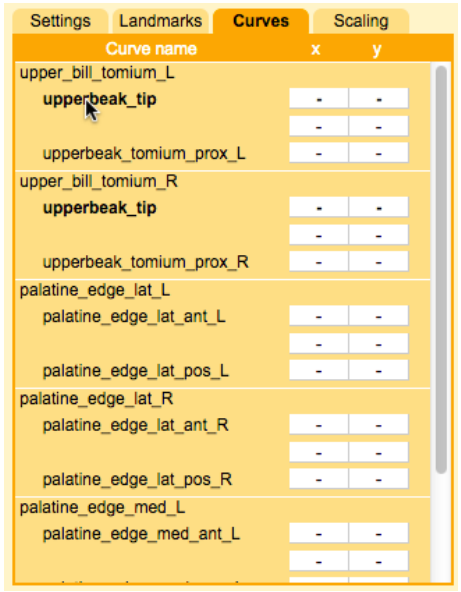
The digitizing app allows you to digitize curves using (Bézier curves. Bézier curves are constructed from a series of control points. Two control points at each end of the curve determine its start and end point. Control points in between control how the curve bends away from a straight line between the start and end point.



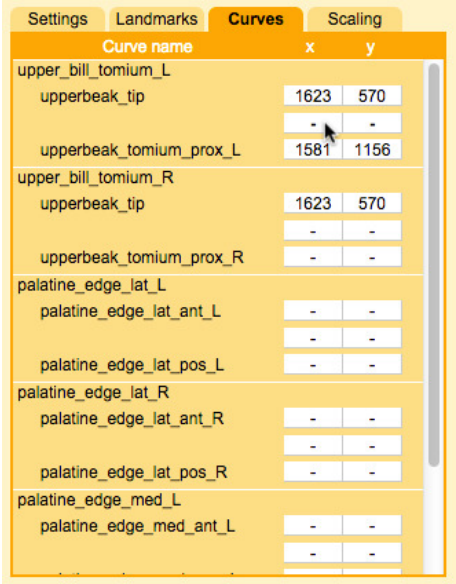
A 3-point Bézier curve with control points (red) and curve points (blue).

The digitizing app only uses quadratic Bézier curves (3 control points) but an unlimited number of these curves can be stringed together end-to-end as Bézier splines. This allows a user to quickly and accurately fit a curve to a feature in an image.

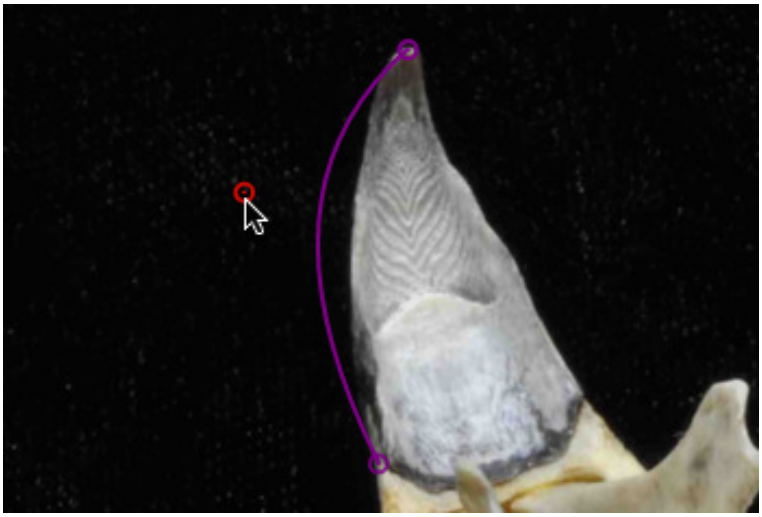
To add a curve, click on the Curves panel at the right. You'll see a list of all the curve names to the far right. Below each curve name are the start and end points specified in *curves.ref*. These start and end points are treated the same as landmarks in the digitizing app (they are added to the list in the Landmarks panel if not already listed in *landmarks.ref*). Select the first landmark of a curve by clicking on the corresponding row.



Position the curve start point on the image by double-clicking as described previously for Digitizing landmarks. Then select the curve end point and position this at the end of the curve. Now all that remains is to "fill in" a curve between these points. Select the empty row (with "-"'s) between the start and end point. These are the intermediate control points that will be used to define the Bézier spline.

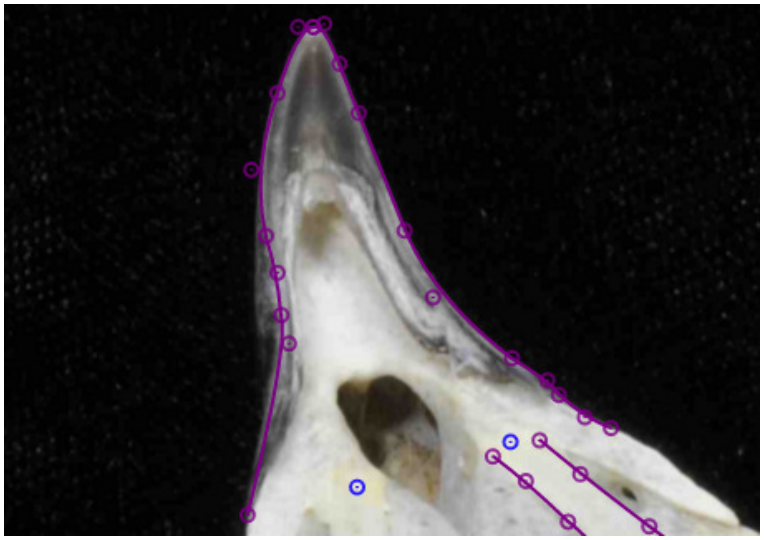


Add this point somewhere between the start and the end by double-clicking on the image. Note that the curve doesn't pass through this point, it only "reaches" toward it. You can change the shape of the curve by click-and-dragging this point.



Change the curve shape by clicking and dragging the control points.

For most curves a single intermediate point will probably not be enough to fit the shape well. Once you add an intermediate point a new empty row will open up below the current point in the Curves panel. Select this point and position it somewhere near the curve. The curve will end at that intermediate point so select the next empty row and add another intermediate point (for five points total, including the start and end). Note that as you add control points they alternate between points that the curve "reaches" to and points that the curve passes through; these are Bézier curves lined up one after another to form a Bézier spline.



Bézier splines digitized to fit the edge of the upper beak.

Repeat this, adding additional intermediate points until you have a sufficient number of points to fit the natural curve. To move back and forth between curve points on a particular curve without having to click the rows in the Curves panel you can use the keyboard shortcuts 'n' and 'p', for next and previous, respectively. Just make sure that the curves are always complete (the total number of control points must be odd). If the curve doesn't extend continuously between the start and end landmarks then you'll either need to add or remove one intermediate point. With a sufficient number of intermediate points you should be able to fit a spline to pretty much any natural curve.

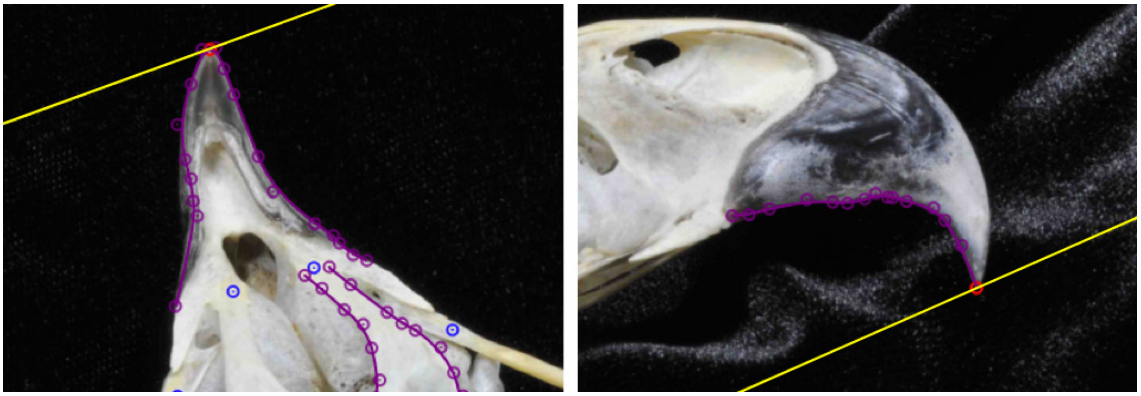
If you are collecting curve data there is an additional consideration that will effect the success of the curve reconstruction. The accuracy of the curve reconstruction can vary depending on the orientation of the object within the calibration volume. If you understand a bit about how the curve reconstruction relates to the epipolar line then you can find an orientation that will afford you the most accuracy.

Stereo reconstruction requires corresponding points between different views. Curves complicate this a bit because while the first and last point are clearly corresponding, how do the points along one curve correpond to those along the other? You might think it would be as simple as matching points that are at same relative position along each curve's length (i.e. a point halfway along one curve corresponds to a point halfway along the other). This turns out not to be the case because of how the different views distort the projection of the curve onto the image plane.

To solve this problem, StereoMorph identifies corresponding points by taking a point along the curve in one view and finding a point along the curve in the other view that intersects with the epipolar line of the first point. Basically, using information from the calibration to identify the corresponding points. This works best where the epipolar line is perpendicular to the curve because there is a clear, single point of intersection. At the other extreme, if the epipolar line is parallel to the curve there can be several points on

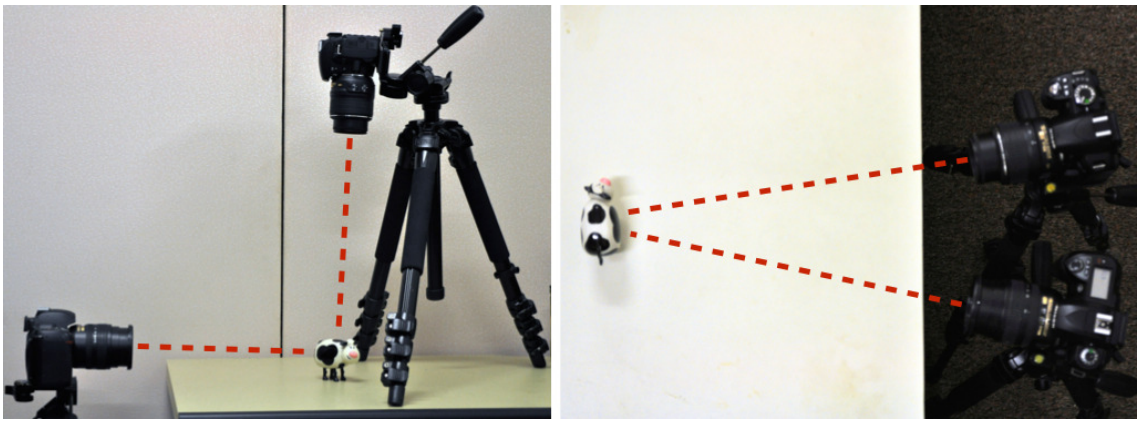
the curve at a similar distance from the epipolar line.

For this reason, if you are collecting curves it is best to avoid orienting the object so that the epipolar line is less than 10 degrees relative to the curve. This may not be avoidable in some cases but it can at least be minimized. The images below show the same curve digitized for two different orientations of the specimen. The left is a better orientation of the specimen for this curve because the epipolar line is nearly 90 degrees to the curve along its entire length. In contrast, with the orientation on the right the epipolar line will be nearly tangent to some portions of the curve.



The same curve digitized for different orientations of the specimen. Left is a better orientation for this curve since the epipolar line is less tangent to any one portion of the curve.

You can easily predict the orientation of the epipolar line for a particular camera arrangement. Imagine a line extending straight out from each camera as in the two arrangements below. Then imagine what each of these lines would look like in the other camera view (these are the epipolar lines for the center of the image in each view). These lines represent generally how the epipolar line will be oriented within each view.



Once you've finished digitizing all of the shapes you'd like to digitize be sure to click "Save".

8.4 Moving between images

Once you've digitized shape data in one image you can move to another image using the buttons or drop-down menus at the bottom of the control panel.

Changing the aspect will move to another image within the same view (here "aspect" refers all of the images from a particular camera view). Changing the view will stay within the same aspect and just change the view.

8.5 Keyboard shortcuts and cursor actions

Below is a guide to the mouse actions and keyboard shortcuts for the digitizing app:

- *click-and-drag over image*: Move image
- *click-and-drag over marker*: Move marker (i.e. landmark, ruler point, curve control point)
- *arrow*: Move marker or image (if no marker is selected) in one pixel increments up/down/right/left
- *shift* + *arrow*: Move marker or image in 10 pixel increments up/down/right/left
- *shift* + *cmnd* + *arrow*: Move image or marker in 100 pixel increments up/down/right/left
- *scroll over image*: Zoom in/out of image
- *double-click*: Add a marker, re-position a marker, select/unselect marker
- *Auto-advance*: This is an option in the Settings panel that will automatically advance to the next marker in sequence after the current marker is digitized.
- *x*: Same action as double-click (add a marker, re-position a marker, select/unselect marker)
- *n*: Select the next marker in sequence (the sequence for curve control points is start, end, and then intermediate points)
- *p*: Select the previous marker in sequence
- *d*: Delete the selected marker
- *shift* + *s*: Save shapes
- *shift* + <: Move to previous image (in Aspects)

- *shift* + >: Move to next image (in Aspects)
- *refresh*: Restore original shape data from when app was initially loaded (does not change saved files)

9 3D Reconstruction and unification

Once you have shape data digitized in two camera views you are ready to reconstruct the landmarks and/or curves into 3D. If you've photographed the same object in different orientations (different aspects) reconstruction of these shapes will produce separate 3D sets that are in different coordinate systems. These can be aligned with one another into a single set based on shared landmarks, a process referred to here as unification.

As of StereoMorph v1.5, there is a single function, reconstructStereoSets(), that will perform all of the steps of reconstruction and unification of landmarks and curves. This section will demonstrate how to use this function with the 2D shape data in the tutorial project folder ("Run 1/Reconstruct/Shapes 2D").

Load the StereoMorph library if it isn't already loaded and if you are following along with the tutorial ensure that the tutorial folder is your current working directory.

```
> library(StereoMorph)
```

Then call the reconstructStereoSets(). Here's what the basic function call looks like for the tutorial project.

```
> rss <- reconstructStereoSets(
     shapes.2d='Run 1/Reconstruct/Shapes 2D',
     shapes.3d='Run 1/Reconstruct/Shapes 3D',
     cal.file='Run 1/Calibrate/calibration.txt',
     even.spacing='Shapes ref/even_spacing.txt')
```

There are a number of other parameters that can be passed to this function but we'll start with quick descriptions of the basic parameters.

- *shapes.2d*: A file path to a folder containing digitized (2D) shape data, separated into different folders by view.
- *shapes.3d*: A file path to a folder where the reconstructed data should be saved. If this folder doesn't exist, it will be created.
- *cal.file*: The calibration file produced by calibrateCameras().
- *even.spacing*: (Only required for curves) A file path to a '.txt' file listing how many curve points you'd like each reconstructed curve to have. On each line of the '.txt' file, list the curve name and the number of points, separated by a tab (For an example, refer to "Shapes ref/even_spacing.txt" in the tutorial project folder). This input can also be a single number if you want to reconstruct each curve with the same number of points.

By default, the function performs both reconstruction and unification. The function recognizes different aspects of the same object by "_a#" at the end of each filename (e.g. "_a1", "_a2", etc.). So if each of your stereo pairs is a different object and you don't have these aspect numbers at the end of your filenames the function won't perform any unification and will simply perform reconstruction. If the input parameter *print.progress* is TRUE (the default), the function prints each of the steps along with the associated errors:

```
> rss <- reconstructStereoSets(
+         shapes.2d='Run 1/Reconstruct/Shapes 2D',
+         shapes.3d='Run 1/Reconstruct/Shapes 3D',
+         cal.file='Run 1/Calibrate/calibration.txt',
+         even.spacing='Shapes ref/even_spacing.txt')
reconstructStereoSets
     bubo_virginianus_FMNH488595
          Aspect 1
               Landmark Reconstruction RMS Error: 0.7215199 px
               Curve point matching and reconstruction
                    palatine_edge_lat_L
                         Max Reconstruction error: 4.655858 px
                    palatine_edge_med_L
                         Max Reconstruction error: 4.774051 px
                    upper_bill_tomium_L
                         Max Reconstruction error: 0.9674658 px
                    upper_bill_tomium_R
                         Max Reconstruction error: 4.989233 px
          Aspect 2
               Landmark Reconstruction RMS Error: 0.8148567 px
          Aspect 3
               Landmark Reconstruction RMS Error: 0.6173243 px
          Unify landmarks
               Unification RMS Error: 0.590829, 1.030958
     psittacus_erithacus_FMNH312899
          Aspect 1
               Landmark Reconstruction RMS Error: 0.6284269 px
               Curve point matching and reconstruction
                    upper_bill_tomium_L
                         Max Reconstruction error: 1.284035 px
          Aspect 2
               Landmark Reconstruction RMS Error: 0.4803887 px
          Unify landmarks
               Number of common points less than min.common (3). No unification performed.
>
```

Print-out of reconstructStereoSets() run.

From the print-out you can see that the first specimen (bubo_virginianus_FMNH488595) has three different aspects. Landmark reconstruction is performed for each of these three aspects. Plus, the first aspect has 4 curves which are also reconstructed. Your mean landmark reconstruction errors should be near or less than 1 pixel. The curve reconstruction errors can range from 1-10 pixels. Then all three aspects are unified. Since there are three aspects, there are two separate unifications: two aspects are unified, then the third is unified with this unified set. The two errors after "Unification RMS Error" are the errors for each of these unifications in the same units as the calibration (here, mm).

The second specimen (psittacus_erithacus_FMNH312899) has landmarks digitized in two aspects but I deliberately chose landmarks such that there is only 1 common landmark between the two sets. For this reason the function reports that there are not enough common points for unification. In this case, the function will save each aspect as a separate 3D set (retaining the "_a#"). To unify 3D sets you theoretically need at least 3, non-colinear points in common between the sets. However, in practice more common

points are required (at least 5-6) to get a good alignment between the two sets.

As mentioned previously, if you have a single image pair per specimen (single aspect) you don't have to put an aspect number at the end of the filename. There are no aspects to unify so the function will just perform reconstruction. For example, here is what the function call and result would look for a single image pair per specimen:

```
> # Run for single image pair per specimen
> rss <- reconstructStereoSets(
     shapes.2d='Run 1/Reconstruct/Shapes 2D (single aspect)',
     shapes.3d='Run 1/Reconstruct/Shapes 3D',
     cal.file='Run 1/Calibrate/calibration.txt',
     even.spacing='Shapes ref/even_spacing.txt')
```

```
reconstructStereoSets
    bubo_virginianus_FMNH488595
        Landmark Reconstruction RMS Error: 0.7215199 px
        Curve point matching and reconstruction
            palatine_edge_lat_L
                Max Reconstruction error: 4.655858 px
            palatine_edge_med_L
                Max Reconstruction error: 4.774051 px
            upper_bill_tomium_L
                Max Reconstruction error: 0.9674658 px
            upper_bill_tomium_R
                Max Reconstruction error: 4.989233 px
    psittacus_erithacus_FMNH312899
        Landmark Reconstruction RMS Error: 0.6284269 px
        Curve point matching and reconstruction
            upper_bill_tomium_L
                Max Reconstruction error: 1.284035 px
```

Example of reconstruction with 1 image pair per specimen.

There are several optional input parameters to reconstructStereoSets() that can be helpful. For instance, if you only want to reconstruct/unify one or more particular specimens, you can specify these as a vector using the input parameter *set.names*.

```
> # Only run for particular file(s) using set.names
> rss <- reconstructStereoSets(
     shapes.2d='Run 1/Reconstruct/Shapes 2D',
     shapes.3d='Run 1/Reconstruct/Shapes 3D',
     cal.file='Run 1/Calibrate/calibration.txt',
     even.spacing='Shapes ref/even_spacing.txt',
     set.names=c('psittacus_erithacus_FMNH312899'))
```

If you want to run the function only for specimens where the 2D shape files have changed, you can set *update.only* to TRUE (default is FALSE).

```
> # Only run on modified files using update.only
> rss <- reconstructStereoSets(
      shapes.2d='Run 1/Reconstruct/Shapes 2D',
      shapes.3d='Run 1/Reconstruct/Shapes 3D',
      cal.file='Run 1/Calibrate/calibration.txt',
      even.spacing='Shapes ref/even_spacing.txt',
      update.only=TRUE)
```

This saves time in running the function so that as you are digitizing specimens you can run reconstructStereoSets() only for those files that actually need to be updated.

If you want the function to print more details as it runs, you can set the input parameter *verbose* to TRUE (default is FALSE). For instance, this will print the reconstruction error for each of the landmarks. This is helpful for identifying incorrectly digitized markers.

```
> # Print more error details using verbose
> rss <- reconstructStereoSets(
      shapes.2d='Run 1/Reconstruct/Shapes 2D',
      shapes.3d='Run 1/Reconstruct/Shapes 3D',
      cal.file='Run 1/Calibrate/calibration.txt',
      even.spacing='Shapes ref/even_spacing.txt',
      verbose=TRUE)
```

Here are some additional input parameters that might be helpful:

- *reconstruct.curves*: (default TRUE) Make FALSE to only reconstruct landmarks.
- *unify*: (default TRUE) Make FALSE to reconstruct all aspects as separate sets (no unification).
- *min.common*: (default 3) This is the minimum number of common points between sets that are required for unification. It is helpful to set this higher than 3 (e.g. 5) so that you can easily identify when you need more common points between different aspects.

10 Reading and visualizing shape data

Now that we have 3D shape data it would be useful to read those data into a common format for analysis and visualization. This section will demonstrate how to read shape data from a StereoMorph shape file and how to plot these shapes using the R package 'rgl'.

10.1 Reading shape data

Load the StereoMorph library if it isn't already loaded and if you are following along with the tutorial ensure that the tutorial folder is your current working directory.

```
> library(StereoMorph)
```

We'll use the StereoMorph function readShapes() to read the 3D shape data we created in the previous section.

```
> file <- 'Run 1/Reconstruct/Shapes 3D/bubo_virginianus_FMNH488595.txt'
  shapes <- readShapes(file=file)
```

This reads all of the shape data from bubo_virginianus_FMNH488595.txt into the list structure `shapes` used by StereoMorph to organize different types of shape data. This structure has a custom print format that tells you all of the shapes in the list:

```
> shapes

Shapes
     landmarks
          Dimensions: 34 x 3
          Rownames: basioccipital_proc_ant, cranium_occipital, ...
     curves
          List names: palatine_edge_lat_L, palatine_edge_med_L, ...
```

The two shapes within `shapes` are landmarks and curves. The landmarks are in a 34 x 3 matrix, where the rows correspond to the landmarks and the columns to the 3D reconstructed coordinates. To access the landmarks matrix, use the '$' operator (as you would use to access any list element):

```
> shapes$landmarks
                                    x          y          z
basioccipital_proc_ant        -45.401700 -31.891966  10.424904
cranium_occipital             -74.634007 -31.841034   4.743550
foramen_magnum_sup            -66.019175 -37.620405   7.121629
...
```

You can then access any of the landmarks by using the standard matrix notation in R:

```
> shapes$landmarks['cranium_occipital', ]
        x         y         z
-74.63401 -31.84103   4.74355
```

The curve points require one extra step. Each curve can have a different number of points, so each is saved within a separate list, accessible by the curve name. To get a list of the curve names, use the '$' operator with 'curves' and the names() function:

```
> names(shapes$curves)
[1] "palatine_edge_lat_L" "palatine_edge_med_L" "upper_bill_tomium_L"
"upper_bill_tomium_R"
```

These are the 4 curves that have 3D coordinates. Use the '$' operator to return a particular curve, which yields a n x 3 matrix, where n is the number of points in the curve (specified previously through *even.spacing*).

```
> shapes$curves$upper_bill_tomium_L
                x         y         z
 [1,] 13.2310210 -25.35104 22.328762
 [2,] 13.3776717 -25.11830 22.070400
 [3,] 13.2665825 -24.71530 21.605985
 ...
```

The readShapes() function can also be used to read multiple shape files to create a landmark array or larger list structures of curves. Specify two files to be read by readShapes():

```
> file <- c('Run 1/Reconstruct/Shapes 3D/bubo_virginianus_FMNH488595.txt',
     'Run 1/Reconstruct/Shapes 3D/psittacus_erithacus_FMNH312899_a1.txt')
  shapes <- readShapes(file=file)
```

Shapes is still a list, but now the landmarks are an array of dimensions n x m x k, where n is the number of landmarks, m is the number of dimensions, and k is the number of files. Also, the curves have gained an additional level. The first level are the files that have been read, then each curve.

```
> shapes

Shapes
     landmarks
          Dimensions: 35 x 3 x 2
          Rownames: basioccipital_proc_ant, cranium_occipital, ...
          Matrix names: bubo_virginianus_FMNH488595, ...
     curves
          bubo_virginianus_FMNH488595
               List names: palatine_edge_lat_L, palatine_edge_med_L, ...
          psittacus_erithacus_FMNH312899_a1
               List names: upper_bill_tomium_L
```

Like before, the '$' operator can be used to access the landmarks, accessing, for example, a particular landmark across all of the files,

```
> shapes$landmarks['cranium_occipital', , ]
  bubo_virginianus_FMNH488595 psittacus_erithacus_FMNH312899_a1
x                   -74.63401                                NA
y                   -31.84103                                NA
z                     4.74355                                NA
```

or accessing all the landmarks for a particular file.

```
> shapes$landmarks[, , 'bubo_virginianus_FMNH488595']
                                      x         y         z
basioccipital_proc_ant        -45.401700 -31.891966  10.424904
cranium_occipital             -74.634007 -31.841034   4.743550
foramen_magnum_sup            -66.019175 -37.620405   7.121629
...
```

In addition to inputting a vector of files for readShapes(), you can also input a folder containing shape files. In that case readShapes() will read in all of the shape files in that folder:

```
> shapes <- readShapes(file='Run 1/Reconstruct/Shapes 3D')
```

10.2 Visualizing shape data

We can now plot the landmarks and curve points using the rgl function plot3d(). First make sure the rgl package is loaded.

```
> library(rgl)
```

Read in the shapes from a file in the tutorial project "Shapes 3D" folder.

```
> file <- 'Run 1/Reconstruct/Shapes 3D/bubo_virginianus_FMNH488595.txt'
  shapes <- readShapes(file=file)
```

Save the landmark matrix into a separate variable.

```
> lm <- shapes$landmarks
```

Define the ranges of x,y,z values of the plots as a variable. This will be used to set the aspect ratio of the plot box. Otherwise, plot3d() will plot the points in a box with equal lengths on all sides.

```
> r <- apply(lm, 2, 'max') - apply(lm, 2, 'min')
```

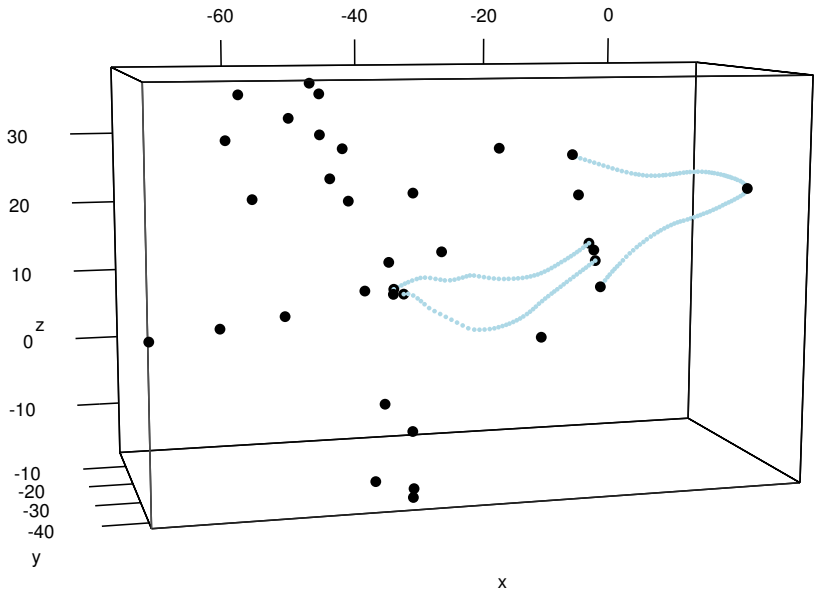
Use plot3d() to plot the landmarks within a bounding box.

```
> plot3d(lm, aspect=c(r/r[3]), size=7)
```

To plot the curve points, use lapply() to apply plot3d() to each element of the list 'shapes'.

```
> lapply(shapes$curves, plot3d, size=4, col="lightblue", add=TRUE)
```

The rgl plot opens in an interactive window that allows you to rotate the coordinates using the mouse.



Plot of reconstructed and unified landmarks and curve points using plot3d() in the rgl package.

11 Reflecting missing bilateral landmarks

When digitizing landmarks and curves you might not be able to digitize every point on both the left and right side. For objects that have bilateral symmetry you can use pairs of left and right points and points within the midline plane to define the midline plane and then project landmarks that are missing on one side across the midline plane. Even if you have data for both the left and right side you might want to average the sides to create a final shape that has perfect bilateral symmetry. This can be useful for reducing noise or for making subsequent analyses simpler. This section will show how to use the StereoMorph function reflectMissingShapes() to do both of these operations.

Load the StereoMorph library if it isn't already loaded and if you are following along with the tutorial ensure that the tutorial folder is your current working directory.

```
> library(StereoMorph)
```

In order for reflectMissingShapes() to recognize which landmarks are left, right, or on the midline, your landmark names will have to follow a particular convention. Landmark names that are right or left should end in "_" followed by either "R" or "L". Capitalization doesn't matter, so "r" or "l" will also work. These can be followed by numbers (e.g. for curve points) but should not be followed by other letters. All landmarks that don't have these endings will be treated as midline landmarks. For example, here are examples of acceptable landmark names to indicate a side:

```
jugal_upperbeak_L
jugal_upperbeak_r
jugal_upperbeak_R0202
```

The reflectMissingShapes() function takes two main inputs: *shapes* (the shape data to be reflected) and *file* (where the reflected shapes should be saved). But these two inputs allow quite a bit of flexibility. For instance, the function call to reflect missing landmarks for one 3D shape file looks like this (the paste0() function is used to paste together strings into a single string):

```
> specimen <- 'bubo_virginianus_FMNH488595.txt'
  shapes <- paste0('Run 1/Reconstruct/Shapes 3D/', specimen)
  file <- paste0('Run 1/Reconstruct/Shapes 3D reflected/', specimen)
  rms <- reflectMissingShapes(shapes=shapes, file=file, average=TRUE)
```

Note that *average* is set to TRUE (default is FALSE). This will average all of the bilateral landmarks such that all midline landmarks are within a single, midline plane and all left and right landmarks are reflected perfectly across the midline plane.

If the input to reflectMissingShapes() is a single set of shapes, the function will output the reflected shape data as a shape data structure. As shown in Reading shape data, you can access the reflected landmarks using the '$' operator:

```
> rms$landmarks
                                        x          y           z
basioccipital_proc_ant         -45.512223 -31.829321  10.9570590
cranium_occipital              -74.666742 -31.822480   4.9011635
foramen_magnum_sup             -66.069883 -37.591664   7.3657781
...
```

You can run reflectMissingShapes() over multiple files at once by making *shapes* and *file* vectors of corresponding file paths:

```
> specimen <- c('bubo_virginianus_FMNH488595.txt',
     'psittacus_erithacus_FMNH312899_a1.txt')
  shapes <- paste0('Run 1/Reconstruct/Shapes 3D/', specimen)
  file <- paste0('Run 1/Reconstruct/Shapes 3D reflected/', specimen)
  rms <- reflectMissingShapes(shapes=shapes, file=file, average=TRUE)
```

Currently, for this input type the function returns NULL.

To run reflectMissingShapes() over all the files in a particular folder, use paths to folders as input rather than to particular files. If a folder input to *file* doesn't exist then one will be created.

```
> shapes <- 'Run 1/Reconstruct/Shapes 3D'
  file <- 'Run 1/Reconstruct/Shapes 3D reflected'
  rms <- reflectMissingShapes(shapes=shapes, file=file, average=TRUE)
```

You can also input a shape structure rather than a shape file.

```
> file <- 'Run 1/Reconstruct/Shapes 3D/bubo_virginianus_FMNH488595.txt'
  shapes <- readShapes(file=file)
  rms <- reflectMissingShapes(shapes=shapes, average=TRUE)
```

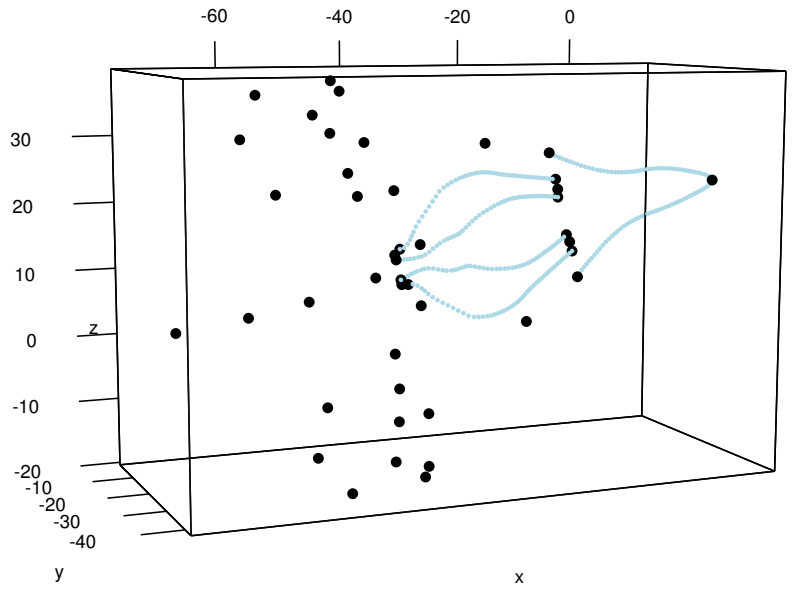
Note that *file* can be omitted, in which case the function will not create a file for the reflected shape data and will only return a shapes list. *file* can only be omitted for single shape set input to reflectMissingShapes().

Lastly, *print.progress* can be set to TRUE in order to view the reflection errors:

```
> rms <- reflectMissingShapes(shapes=shapes, average=TRUE,
   print.progress=TRUE)
```

To visualize the reflected shape data, you can use the plot3d() function as shown previously in Visualizing shape data.

```
> file <- 'Run 1/Reconstruct/Shapes 3D reflected/bubo_virginianus_FMNH488595.txt'
  shapes <- readShapes(file=file)
  lm <- shapes$landmarks
  r <- apply(lm, 2, 'max') - apply(lm, 2, 'min')
  plot3d(lm, aspect=c(r/r[3]), size=7)
  lapply(shapes$curves, plot3d, size=4, col="lightblue", add=TRUE)
```



Plot of reflected landmarks and curves using plot3d() in the rgl package.

12 Aligning bilateral landmarks

The 3D shape data produced so far have an entirely arbitrary orientation in 3D space. For subsequent analyses this shouldn't make a difference but for visualization purposes it can be useful to place the shape data in a consistent orientation. If you have bilateral shape data (left/right), you can align the shapes to the midline plane such that all points at the midline will have z-values of 0. This section will show you how to use the StereoMorph function alignShapesToMidline() to align landmarks and curves to the midline plane.

Load the StereoMorph library if it isn't already loaded and if you are following along with the tutorial ensure that the tutorial folder is your current working directory.

```
> library(StereoMorph)
```

As for reflecting missing landmarks, in order for alignShapesToMidline() to recognize which landmarks are left, right, or on the midline, your landmark names will have to follow a particular convention.Landmark names that are right or left should end in "_" followed by either "R" or "L". Capitalization doesn't matter, so "r" or "l" will also work. These can be followed by numbers (e.g. for curve points) but should not be followed by other letters. All landmarks that don't have these endings will be treated as midline landmarks. For example, here are examples of acceptable landmark names to indicate a side:

```
jugal_upperbeak_L
jugal_upperbeak_r
jugal_upperbeak_R0202
```

The alignShapesToMidline() function takes the same two main inputs as reflectMissingShapes(): *shapes* (the shape data to be aligned) and *file* (where the aligned shapes should be saved). These two inputs also allow quite a bit of flexibility. For instance, the function call to align shapes for one 3D shape file looks like this:

```
> specimen <- 'bubo_virginianus_FMNH488595.txt'
  shapes <- paste0('Run 1/Reconstruct/Shapes 3D reflected/', specimen)
  file <- paste0('Run 1/Reconstruct/Shapes 3D aligned/', specimen)
  asm <- alignShapesToMidline(shapes=shapes, file=file)
```

If the input to alignShapesToMidline() is a single set of shapes, the function will output the aligned shape data as a shape data structure. As shown in Reading shape data, you can access the reflected landmarks using the '$' operator:

```
> asm$landmarks
                                    [,1]        [,2]          [,3]
basioccipital_proc_ant        -28.8815899  -1.7153879 -4.529710e-14
cranium_occipital             -57.7443301   5.6059582  0.000000e+00
foramen_magnum_sup            -50.5257664  -2.2142802 -8.881784e-15
...
```

Note that if you want all of your midline landmarks to have 0 z-values, you'll need to run reflection with average set to TRUE.

You can run alignShapesToMidline() over multiple files at once by making *shapes* and *file* vectors of corresponding file paths:

```
> specimen <- c('bubo_virginianus_FMNH488595.txt',
     'psittacus_erithacus_FMNH312899_a1.txt')
  shapes <- paste0('Run 1/Reconstruct/Shapes 3D reflected/', specimen)
  file <- paste0('Run 1/Reconstruct/Shapes 3D aligned/', specimen)
  asm <- alignShapesToMidline(shapes=shapes, file=file)
```

Currently, for this input type the function returns NULL.

To run alignShapesToMidline() over all the files in a particular folder, use paths to folders as input rather than to particular files. If a folder input to *file* doesn't exist then one will be created.

```
> shapes <- 'Run 1/Reconstruct/Shapes 3D reflected'
  file <- 'Run 1/Reconstruct/Shapes 3D aligned'
  asm <- alignShapesToMidline(shapes=shapes, file=file)
```

You can also input a shape structure rather than a shape file.

```
> file <- 'Run 1/Reconstruct/Shapes 3D reflected/bubo_virginianus_FMNH488595.txt'
  shapes <- readShapes(file=file)
  asm <- alignShapesToMidline(shapes=shapes)
```

Note that *file* can be omitted, in which case the function will not create a file for the aligned shape data and will only return a shapes list. *file* can only be omitted for single shape set input to alignShapesToMidline().

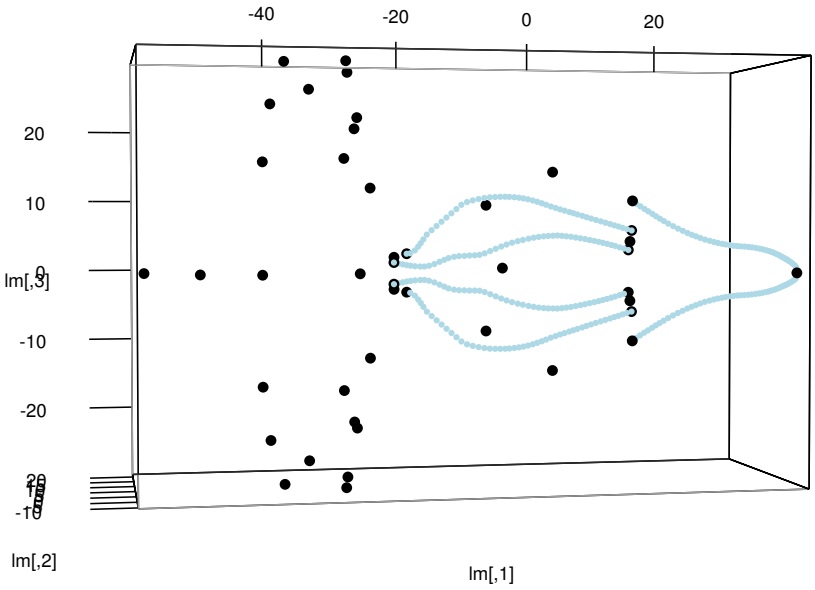
Lastly, *print.progress* can be set to TRUE in order to view the alignment errors:

```
> asm <- alignShapesToMidline(shapes=shapes, print.progress=TRUE)
```

The printed errors are the distance of the midline points from the midline. If you've previously reflected the shape data with average set to TRUE all of these errors will be 0 since all midline points will be aligned perfectly with the midline.

To visualize the aligned shape data, you can use the plot3d() function as shown previously in Visualizing shape data.

```
> file <- 'Run 1/Reconstruct/Shapes 3D aligned/bubo_virginianus_FMNH488595.txt'
  shapes <- readShapes(file=file)
  lm <- shapes$landmarks
  r <- apply(lm, 2, 'max') - apply(lm, 2, 'min')
  plot3d(lm, aspect=c(r/r[3]), size=7)
  lapply(shapes$curves, plot3d, size=4, col="lightblue", add=TRUE)
```



Plot of aligned landmarks and curves using plot3d() in the rgl package.

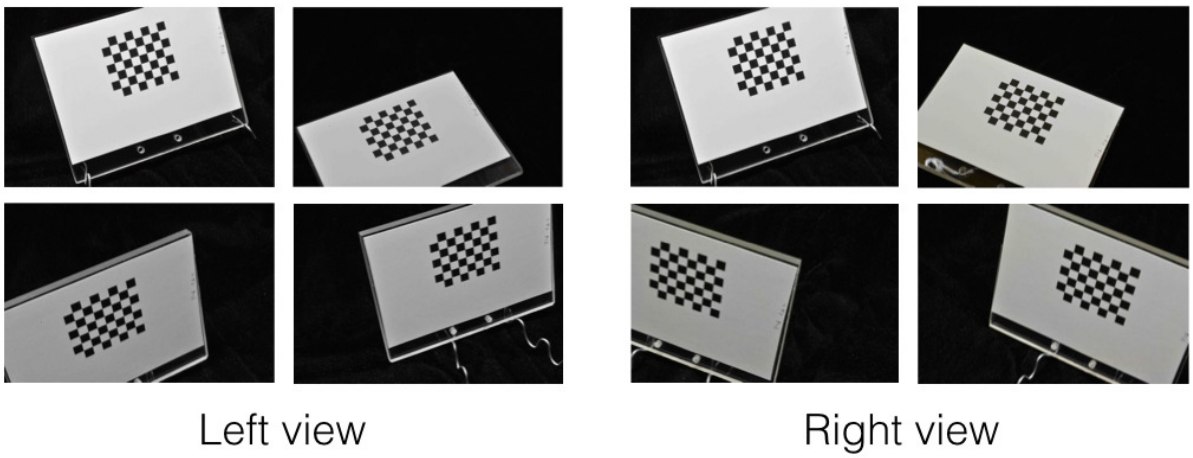
13 Additional tools and features

13.1 Testing the accuracy using a second checkerboard

The section Calibrating stereo cameras showed how to calibrate a stereo camera setup, by photographing a checkerboard pattern at different positions and angles in the calibration volume and using the StereoMorph function calibrateCameras() to estimate the calibration coefficients and test the calibration accuracy. Recall however that by determining the calibration accuracy using the same checkerboard used in the calibration it isn't possible to test whether the calibration has the correct scaling (the checkerboard square size is identical throughout).

This section will demonstrate how to use the StereoMorph function testCalibration() to test the calibration accuracy, using a checkerboard with a different square size. This serves as a more independent test of the calibration accuracy. This section will be brief since the steps for using testCalibration() are nearly identical to those for calibrateCameras() (refer to Calibrating stereo cameras for more details on calibrateCameras()).

Begin by photographing a checkerboard in the calibration volume in the same manner as for the calibration step. The test calibration images for the tutorial project can be found in the folder "Run 1/Test calibration/Images". Note that not only are the squares for this checkerboard a different size from the calibration checkerboard (5.080 mm versus 6.35 mm) but the number of internal corners in the pattern also differs (7x6 versus 8x6).



Examples of a test checkerboard photographed from two views at different positions and angles within the calibration space.

Upload the photographs into a folder, separating the different views into different folders. Be sure to use the same view names as you have used throughout.

Load the StereoMorph library if it isn't already loaded and if you are following along with the tutorial ensure that the tutorial folder is your current working directory.

```
> library(StereoMorph)
```

Then call the function testCalibration(). The basic input parameters are nearly identical to calibrateCameras(). The tutorial project function call looks like this:

```
> test_cal <- testCalibration(img.dir='Run 1/Test calibration/Images',
    cal.file='Run 1/Calibrate/calibration.txt',
    corner.dir='Run 1/Test calibration/Corners',
    sq.size='5.080 mm', nx=7, ny=6,
    plot.dir='Run 1/Test calibration/Error tests',
    verify.dir='Run 1/Test calibration/Images verify')
```

Here is a brief description of each parameter:

- *img.dir*: The file path to the folder containing the checkerboard images, each view in a separate folder.
- *cal.file*: A file path to the calibration file created by calibrateCameras().
- *corner.dir*: A file path to a folder where the corners will be saved. If this folder does not exist, a new folder will automatically be created.
- *sq.size*: The size of the squares along with the units (length along any one side).
- *nx*: The number of internal corners along one dimension (the choice of which is *nx* and *ny* is arbitrary but must be consistent throughout).
- *ny*: The number of internal corners along the other dimension.
- *plot.dir*: A folder in which to save the error diagnostics plots. Can be omitted to just print error summary in console.
- *verify.dir*: (Optional) A file path to a folder where images will be saved that show the detected corners. If this folder does not exist, a new folder will automatically be created.

If you run this for the tutorial project the function will access the saved corners and prompt you sequentially whether you would like to repeat corner detection. You can simply enter 'n' at each prompt and the function will print the test calibration errors using all of the image pairs in *img.dir*.

Just as in the calibration step, it's important to review the images in *verify.dir* and make sure that the corners are being returned in the same order for all of the images (the first corner is indicated by a red circle). If your cameras are arranged such that one view is upside-down relative to the other (if *flip.view* was TRUE for the calibration) you don't have to specify that as an input parameter. The function will detect this setting from the calibration file.

The function returns the accuracy test in the same way as the calibration step. Several error diagnostic plots are created if *plot.dir* is specified. Additionally, the function prints an error summary in the console:

```
dltTestCalibration Summary
     Number of aspects: 6
     Number of views: 2
     Square size: 5.08 mm
     Number of points per aspect: 42
     Aligned ideal to reconstructed (AITR) point position errors:
          AITR RMS Errors (X, Y, Z): 0.0148 mm, 0.0220 mm, 0.0212 mm
          Mean AITR Distance Error: 0.0309 mm
          AITR Distance RMS Error: 0.0346 mm
     Inter-point distance (IPD) errors:
          IPD RMS Error: 0.0316 mm
          IPD Mean Absolute Error: 0.0237 mm
          Mean IPD error: -0.0161 mm
     Adjacent-pair distance errors:
          Mean adjacent-pair distance error: 0.000650 mm
          Mean adjacent-pair absolute distance error: 0.0167 mm
          SD of adjacent-pair distance error: 0.0188 mm
     Epipolar errors:
          Epipolar RMS Error: 0.304 px
          Epipolar Mean Error: 0.304 px
          Epipolar Max Error: 0.970 px
          SD of Epipolar Error: 0.209 px
```

The errors in this printed summary are fairly close to those from the calibration step. One important error measure to note here is the "Mean IPD error". This is the mean error of all the inter-point distance (length measurements) among points on the checkerboard. Since this is not an absolute mean error, it should be close to zero because the error should not be biased (i.e. the calibration should not consistently under- or overestimate the lengths). Here it differs from 0 by 0.016 mm (16 microns). We can call this negligible since it is below the pixel resolution threshold for this setup (at least 30 microns/pixel).

14 Citing StereoMorph

The StereoMorph R package and its associated digitizing app are the result of several years of work. I am pleased to offer the app open-source and free of charge and hope that users find it useful and that it brings about interesting, fruitful and fun advances for biologists and non-biologists alike. I only ask that if you use the digitizing app and share your results that you cite StereoMorph. For peer-reviewed publications, please cite the following article:

Olsen, A.M. and M.W. Westneat. 2015. StereoMorph: an R package for the collection of 3D landmarks and curves using a stereo camera set-up. *Methods in Ecology and Evolution*. 6:351-356. DOI: 10.1111/2041-210X.12326.

15 Acknowledgements

I would like to thank several people whose helpful feedback and assistance over the years has greatly improved StereoMorph. These include, but are not limited to: Justin Lemberg, Brett Aiello, Andy Smith, Hannah Weller, Dallas Krentzel, Gavin Thomas, Chery Cherian, Sushma Reddy, Vincent Bonhomme, Stewart Edie, Yinan Hu, Ty Hedrick, Merrilee Guenther, Benjamin Rubin, and José Iriarte-Díaz. I would also like to thank Dave Willard, Ben Marks, and Mary Hennen at the Field Museum of Natural History for their assistance in the use of specimens from the bird skeleton collection. The development of StereoMorph was made possible by funding from the US National Science Foundation (DGE-1144082; DGE- 0903637).