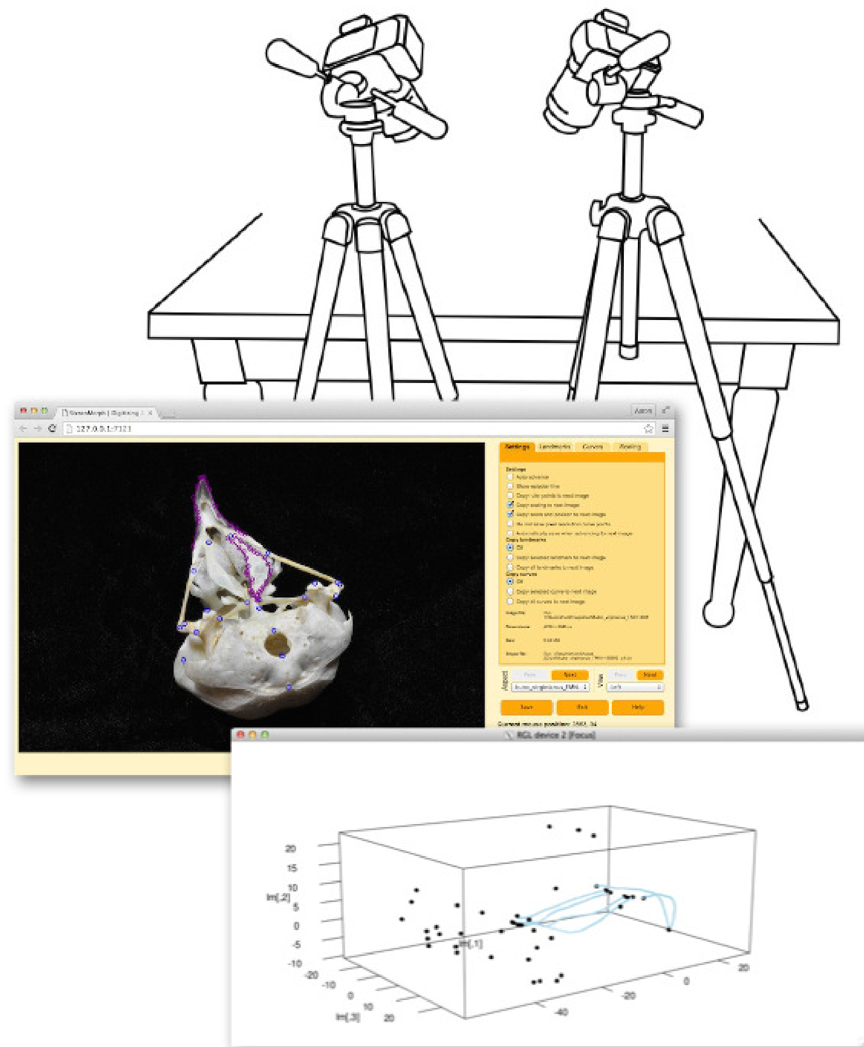
StereoMorph User Guide

Collecting shape data using the R package 'StereoMorph'



March 2017

Version 1.6.1

Table of Contents

1 Introduction **4**

2 Getting started **5**
2.1 Installing StereoMorph 5
2.2 Installing ffmpeg (only if using video) 6

3 Choosing cameras **9**
3.1 Cameras for stereo photography 9
3.2 Cameras for stereo videography 10

4 Creating a checkerboard **11**
4.1 Determining an appropriate checkerboard size 11
4.2 Creating the checkerboard 12
4.3 Measuring square size using DPI and scaling 15
4.4 Precision measurement of square size using a ruler 16
4.5 Creating a checkerboard stand 22

5 Arranging the cameras **25**
5.1 Arrangement for stereo photography 25

6 Calibrating stereo cameras **32**
6.1 General calibration steps and parameters 33
6.2 Calibrating with photographs 34
6.3 Calibrating with videos 36
6.4 Estimating calibration coefficients 38
6.5 Determining the calibration accuracy 39

7 Photographing objects **44**

8 StereoMorph digitizing application **47**
8.1 Digitizing video frames 47
8.2 Opening the digitizing application 48
8.3 Digitizing landmarks 50
8.4 Digitizing curves 52
8.5 Moving between images 56
8.6 Keyboard shortcuts and cursor actions 56

9 3D Reconstruction **58**
9.1 Reconstructing landmarks 58
9.2 Measuring 3D lengths 59
9.3 Reconstructing landmarks and curves 60
9.4 Reading shape data 62

10 Visualizing shape data **64**

11 Reflecting missing bilateral landmarks **66**

12 Aligning bilateral landmarks **69**

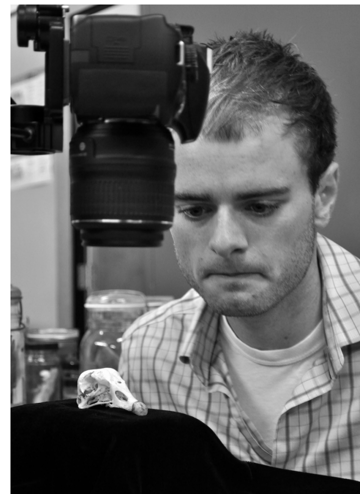
13 Additional features **72**
13.1 Extracting video frames 72
13.2 Extracting synchronized frames 73
13.3 Testing the accuracy using a second checkerboard 73

14 Citing StereoMorph **77**

15 Acknowledgements **77**

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.



These birds are easy photography subjects.

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 or more 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). StereoMorph also has the capability to calibrate stereo video cameras so that 3D shape data can be collected from video-captured subjects like moving animals.

This user guide demonstrates the features of StereoMorph. Each step is accompanied by example datasets so you can try out the code yourself. These example files 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 2017

2 Getting started

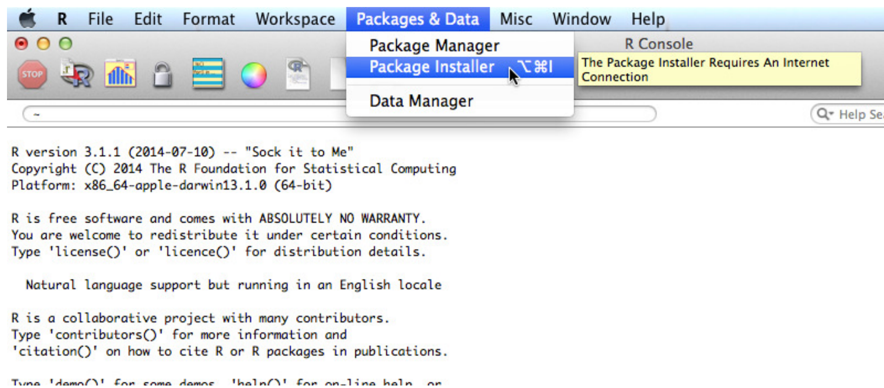
2.1 Installing StereoMorph

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.

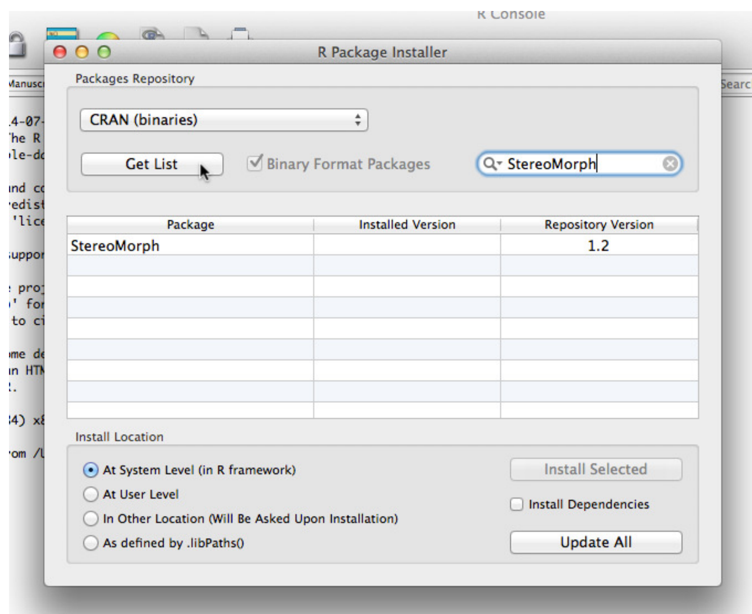
1. If you do not already have R installed on your computer, begin by installing R. R can be installed on Windows, Linux and Mac OS X.

2. Once installed, open R.

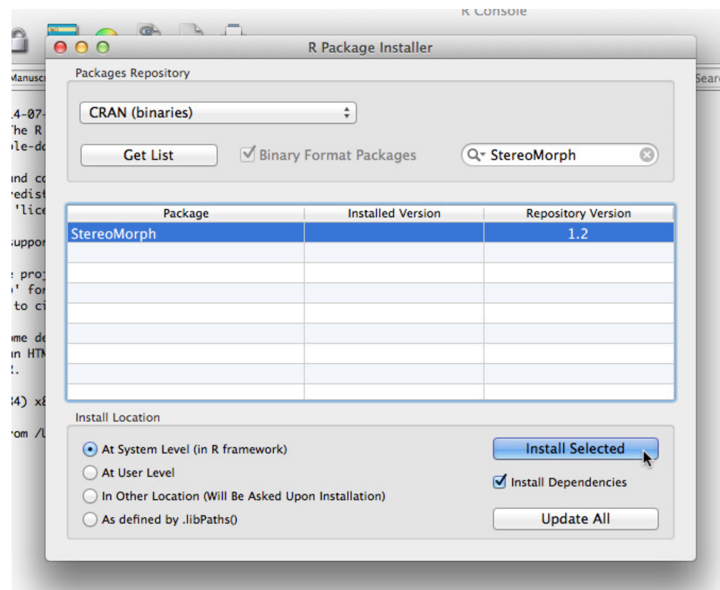
3. Go to *Packages & Data* → *Package Installer*



4. Find the StereoMorph package binary by typing "StereoMorph" into the *Package Search box* and clicking *Get List* (the repository version of StereoMorph may be greater than the version in the image below).



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. Throughout this tutorial I've included R code that you can use to reproduce examples. All R code is in the courier font style on a gray background as in the example below, with comments indicated in orange (and preceded by a '#') and function names in blue.

```
# Print 'Hello world!'
print('Hello world!')
```

To run each line of code, simply copy and paste the code into the R console.

7. Before calling any StereoMorph functions, load the StereoMorph package into the current R session using the library command.

```
# Load the StereoMorph package
library(StereoMorph)
```

2.2 Installing ffmpeg (only if using video)

If you are only using StereoMorph with photographs (not video) then this section can be skipped. As of version 1.6, StereoMorph has some basic video-handling capabilities. This includes calibrating two video cameras using video files, rather than using a set of photographs or already extracted frames from each video.

R does not have native support for reading frames from video files. Thus, a codec library must be installed in order for StereoMorph to extract frames from video files. Once the codec is installed and accessible from the command line, all video handling can be done

within R. This section will show you how to install the video codec library ffmpeg. Currently, these ffmpeg installation instructions are only suitable for Mac OS X.

1. Download the latest static build of ffmpeg for your operating system (less than 80 MB unzipped). On the ffmpeg download site you'll find links to the static builds under "Get the packages". For example, the most recent static build for Mac OS X can be downloaded here.

2. The downloaded static build should be a compressed folder. Unzip this folder. Inside you should see an executable named 'ffmpeg' (and possibly also 'ffprobe' and 'ffserver', which we do not need). Move the ffmpeg executable to the 'Downloads' folder.

3. We will now copy the ffmpeg file into '/usr/local/bin'. The easiest way to do this is through Terminal. Open the Terminal app (for entering command line codes). Note that all of the following commands are to be entered into Terminal (not into the R console).

In Terminal, navigate to the folder containing the ffmpeg file using the 'cd' (change directory) command. Type 'cd' followed by the file path to your Downloads folder. For example, the path on my system is '/Users/aaron/Downloads'.

```
cd /Users/aaron/Downloads
```

Make sure that there is a bin folder '/usr/local/bin/' using the 'mkdir' command (make directory). The 'sudo' before mkdir simply adds extra permissions to the commands so that the codec library will be accessible to your system (you may be prompted to type in your password).

```
sudo mkdir /usr/local/bin/
```

If this returns a line that includes "File exists", then this folder already exists on your system.

Copy the ffmpeg file into /usr/local/bin using the 'cp' command.

```
sudo cp ./ffmpeg /usr/local/bin
```

Set the access permissions to ffmpeg so that it can be called by R using the 'chmod' command. The numeric code '755' specifies a permission level that allows ffmpeg to be called from R.

```
sudo chmod 755 /usr/local/bin/ffmpeg
```

4. To make sure that system can find the ffmpeg command, we'll add /usr/local/bin to the $PATH variable. This is a set of folders that the system looks in when searching for a command. Use the 'cd' command (but without any path after it) to navigate to the home folder.

```
cd
```

Then use the 'open' command to open the bash_profile.

```
open -e .bash_profile
```

If the bash_profile doesn't exist, you can create it using the following 'touch' command and then use the open command.

```
touch .bash_profile
```

The bash_profile should open in TextEdit. Add a new line at the end of the document and paste in the following:

```
export PATH="/usr/local/bin:$PATH"
```

5. Save the bash_profile and then close and re-open Terminal (this will refresh Terminal so that it reflects the changes in the bash_profile). In Terminal use the 'echo' command to print the current $PATH variable.

```
echo $PATH
```

You should see '/usr/local/bin' between colons or at the beginning of the path followed by a colon.

We can now check that ffmpeg is properly installed by simply typing 'ffpmeg' in Terminal. If ffmpeg has been properly installed and can be located on the system, this should print the version number, copyright and other configuration details.

```
ffmpeg
```

3 Choosing cameras

When two or more cameras arranged in stereo are calibrated (using either photographs or video), the resulting calibration is specific to that particular arrangement of the cameras (their position and orientation). The cameras can move if they are rigidly attached to some support system and the support system itself is moved. However, the relative position and orientation of the cameras relative to one another must remain fixed to collect accurate 3D data. Additionally, the calibration is specific to the particular zoom and focus, so these must also remain fixed.

Keeping the cameras immobile and maintaining the same zoom and focus necessitates a few key technical specifications:

- A remote trigger mechanism (since touching the shutter button repeatedly causes the camera to move)
- Manual focus mode (i.e. ability to turn off auto-focus).
- Manual zoom mode (i.e. ability to turn off auto-zoom).

3.1 Cameras for stereo photography

For stereo reconstruction using photographs, the easiest setup consists of two digital cameras and two tripods. Although it is possible to use a mirror to split a single camera view into two views, the resulting photographs would have to be split into image pairs for calibration and digitizing.

Several DSLR (digital single-lens reflex camera) cameras satisfy the above specifications. Many come with a wireless remote shutter trigger, which removes the need to touch the camera at all when taking photographs. 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. The stereo photography examples in this guide were captured using a Nikon D5000.

An example of a DSLR camera (Nikon™ D5000). Image credit: Nikon.

Additionally, lenses can be used with DSLRs that minimize image distortion. As of version 1.6, StereoMorph can account for standard types of lens distortion (e.g. pincushion and barrel distortion). But it's best if distortion can be avoided altogether simply by choice of lens. For example, an AF-S DX Nikkor 18-55mm lens (the basic lens that Niikon sells with their DSLR) zoomed in to 55mm has negligible distortion. 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.

3.2 Cameras for stereo videography

For users interested in collecting shape data from moving subjects, StereoMorph (as of version 1.6) can also calibrate video cameras using a checkerboard pattern. The video cameras must adhere to the same general specifications described above. An advantage of video cameras is that once the camera is triggered it will continue recording without having to continually press a shutter button, thereby minimizing the chance of moving the cameras.

For regular-speed applications (as opposed to high-speed), DSLR cameras or dedicated video cameras (e.g. GoPro™) work well. The stereo video examples in this guide were captured using GoPro™ Hero 3+ cameras. For underwater applications, GoPro cameras work well since they can be used with waterproof housing.

An example of a camera that can be used for stereo videography, the GoPro™ Hero 3+. Image credit: GoPro.

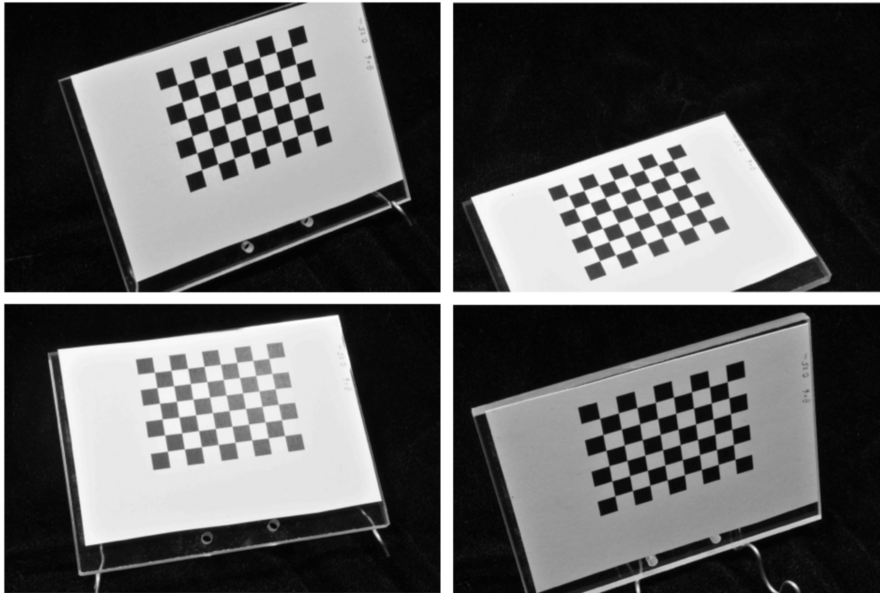
4 Creating a checkerboard

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 identify calibration points. A series of images or frames must be captured with the checkerboard in different positions and orientations throughout the space to be calibrated.

4.1 Determining an appropriate checkerboard size

The size of the whole checkerboard pattern will depend on the scale of your application. To determine the size you need, first estimate the volume of space that you want to calibrate - this is the volume of space that visible from two or more cameras. If the minimum dimension of this space is less than approximately 0.5 m, then the entire checkerboard pattern should be about half the minimum dimension of the volume.

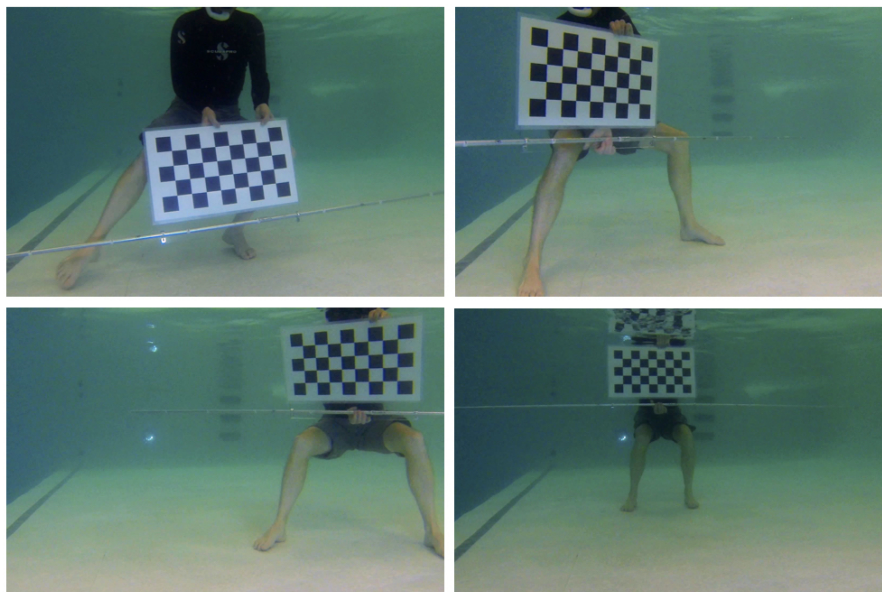
For example, for a calibrated volume with approximate dimensions of 0.3 m x 0.35 m x 0.4 m, a checkerboard pattern measuring 0.15 m along one dimension should be ideal. This allows the checkerboard pattern to be moved around the space and rotated while maintaining the entire pattern within the image frame.



For a calibrated volume with a minimum dimension less than 0.5 m, a checkerboard about half the frame size works well (only one camera view shown). This allows enough room to move the checkerboard to different positions and angles.

Alternatively, if you're calibrating a volume with minimum dimensions larger than 0.5 m it will become difficult to print and manipulate a sufficiently large checkerboard pattern.

For minimum dimensions larger than 1 m (for example, for a calibrated volume measuring 2 m x 3 m x 4 m), a checkerboard pattern measuring 0.5 m along one dimension should work well.



For a calibrated volume with a minimum dimension larger than 0.5 m, a checkerboard pattern that is about 0.5 m along one dimension works well but must be moved more around the volume (only one camera view shown). Image credit: Caine Delacy.

4.2 Creating the checkerboard

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

This section will show you how to create your own checkerboard using the StereoMorph function drawCheckerboard().

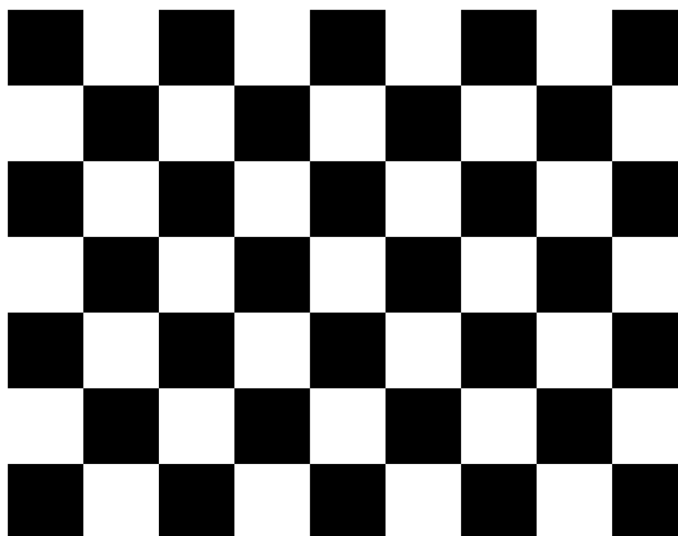
1. Be sure that you've installed the StereoMorph package and load the StereoMorph library into the current R session.

```
# Load the StereoMorph library
library(StereoMorph)
```

2. 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).

```
# Create checkerboard
drawCheckerboard(nx=8, ny=6, square.size=180, filename='Checkerboard 8x6 (180px).JPG')
```

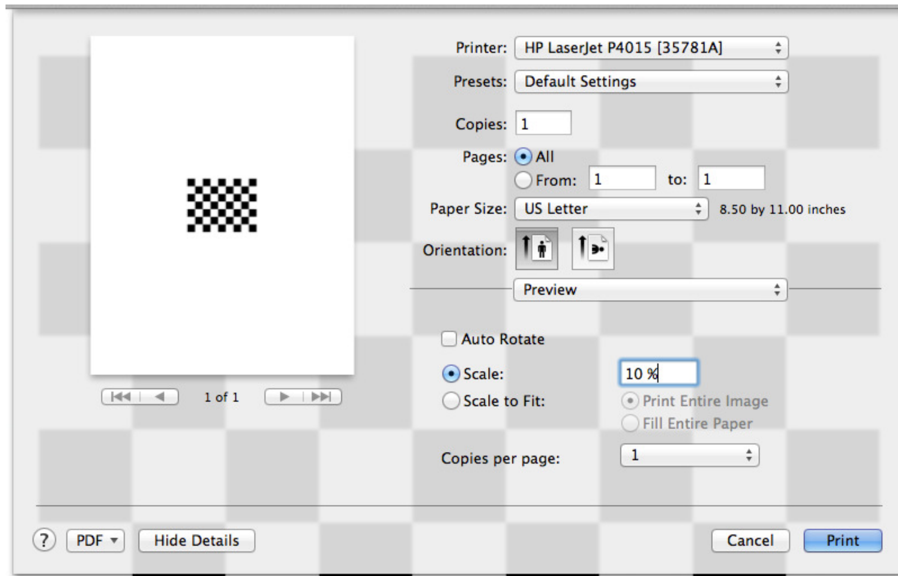
This will create a checkerboard with 8 internal corners along one dimension, 6 internal corners along the other dimension and squares that are 180 pixels along each side. Note that throughout this tutorial "square size" refers to the length along any one side of the square (not, for instance, a diagonal distance or area). Also, note that the number of internal corners are not the number of squares but the number of intersections of black squares. We care about the number of internal corners, rather than the number squares, because it's actually the internal corners that will be detected.



Checkerboard with 8 x 6 internal corners.

Also, note that the checkerboard has a different number of internal corners along each dimension. This is essential to ensure that the corners are returned in the same order when the checkerboard is photographed in different orientations.

3. You can now print the checkerboard. I've been able to achieve great calibrations with inexpensive, desktop printers so any decent printer should do. Even when attaching the paper to a hard surface it's 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. In the printer prompt, print the checkerboard at 10% scaling.

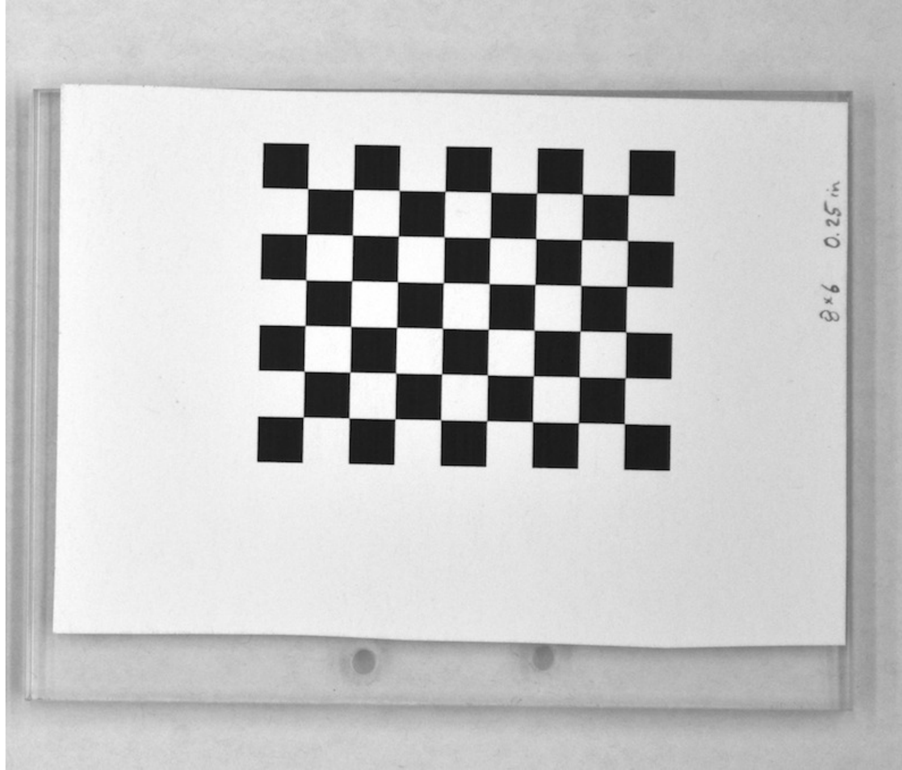


Printing an 8x6 checkerboard at 10% scaling.

4. 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).

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.

4.3 Measuring square size using DPI and scaling

We know that the squares were 180 pixels in the image file, but how large are the squares once they are printed on paper? We must know the size of the squares in real units (e.g. mm, inches) in order to properly scale our calibration. For a rough estimate, we can simply measure the length across the entire checkerboard and divide by the number of squares.

However, if we want to be more accurate we can calculate the real square size using known values from our printer scaling. 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. Thus, we can calculate the real square size using the square size in pixels, the printer scaling, and the printer DPI (dots per inch):

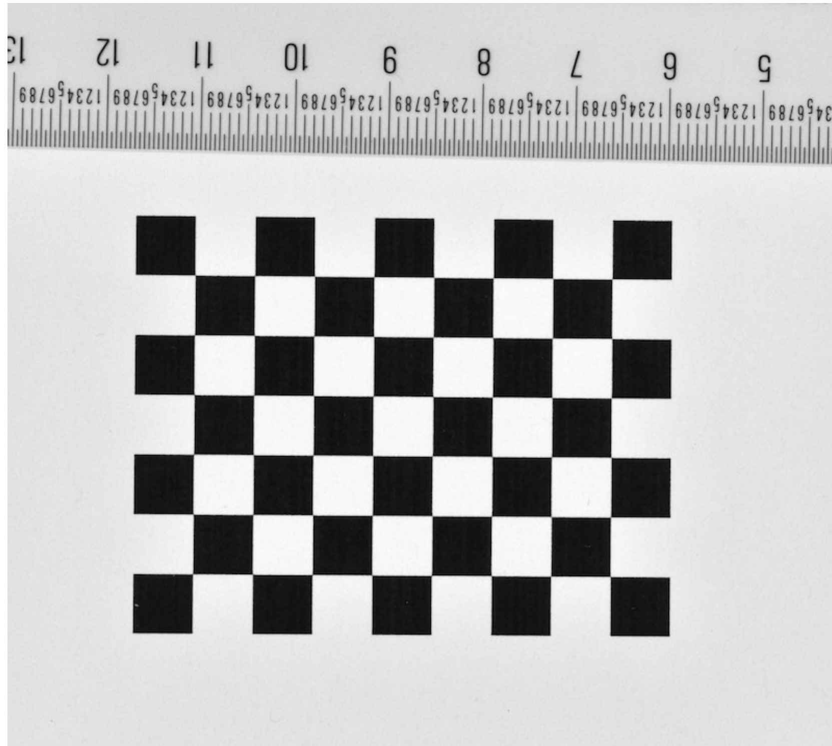
$$\begin{array}{c}\text{Square size in}\\ \text{pixels}\end{array} * \frac{1}{DPI} * Scaling = \begin{array}{c}\text{Printed square}\\ \text{size in inches}\end{array} \tag{1}$$

Or an additional conversion to mm can be added:

$$\begin{array}{c}\text{Square size in}\\ \text{pixels}\end{array} * \frac{1}{DPI} * Scaling * \frac{25.4\ mm}{inch} = \begin{array}{c}\text{Printed square}\\ \text{size in mm}\end{array} \tag{2}$$

The square size of 180 pixels and 10% scaling used in this example were chosen purposely to create printed squares that are 0.25 inches (6.35 mm) along each side when printed at

72 DPI. Once you've measured the square size, it's good to write it 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.



An 8x6 checkerboard printed at 10% scaling and 72 DPI will create 0.25 inch (6.35 mm) squares.

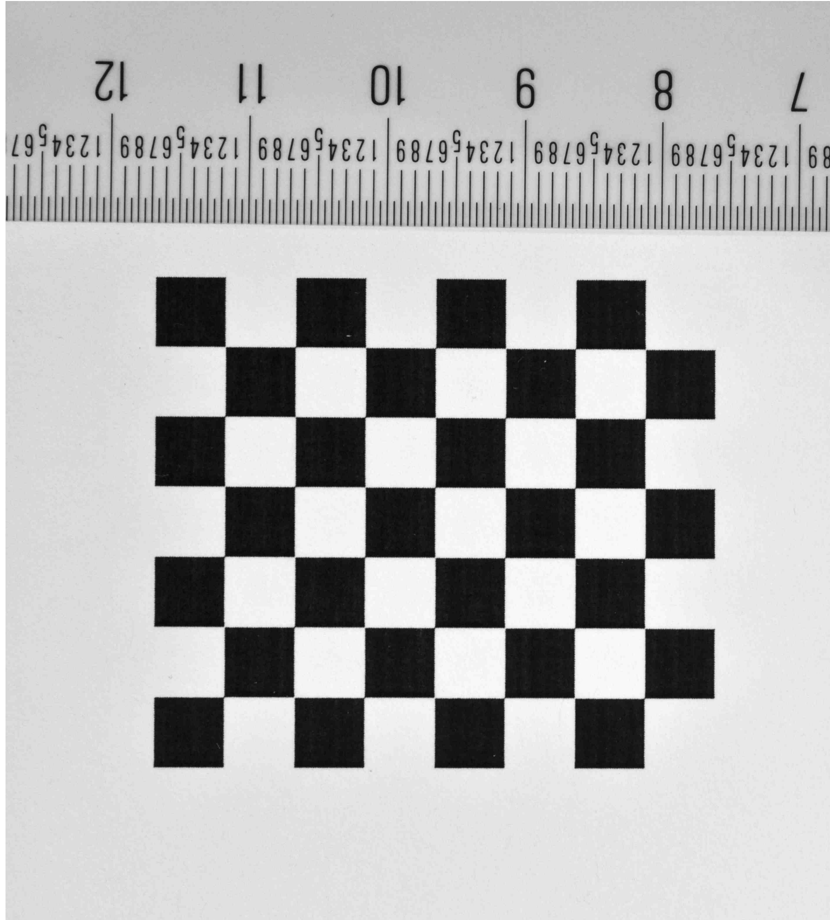
4.4 Precision measurement of square size using a ruler

This section will show you how to precisely measure the square sizes in a checkerboard pattern using 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 7x6 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 name does not contain commas or periods. If you'd like to work through the example below, you can download an example image, "7x6.jpg" here (0.3 MB). Note that this checkerboard is slightly different from the one we created earlier. It has 7x6 internal corners and the square size in the image was 144 pixels (printed at 10% scaling). Save this image to your R working directory.

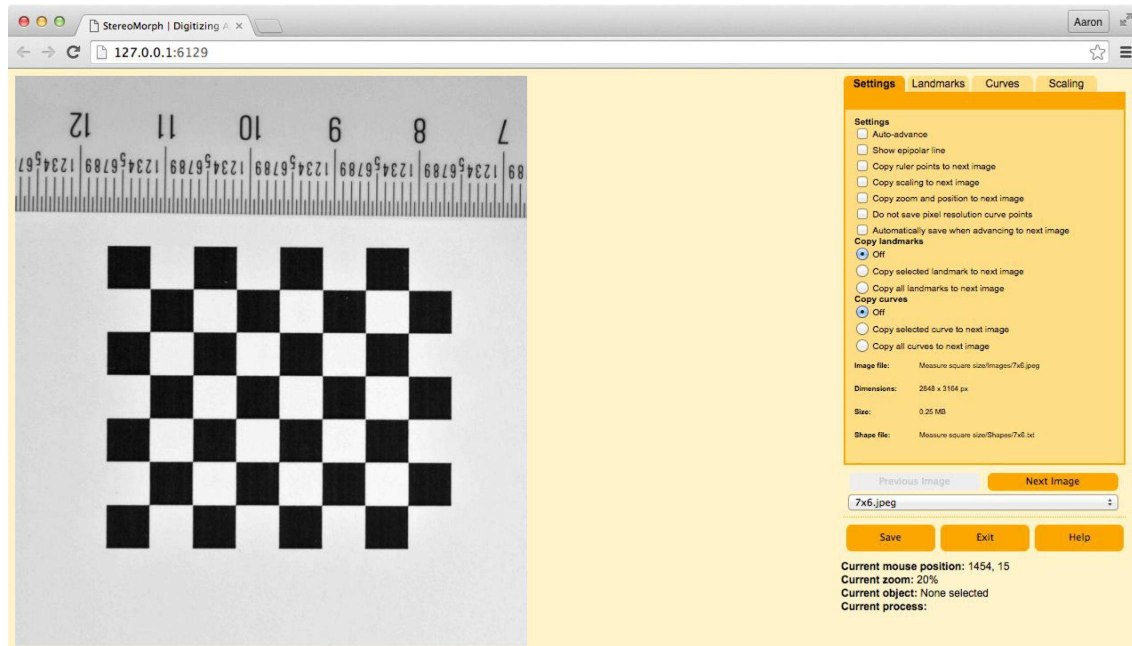
4. Load the StereoMorph library if it isn't already loaded.

```
# Load the StereoMorph library
library(StereoMorph)
```

5. All of the remaining steps will be performed within the StereoMorph digitizing application. This is a 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 example image, "7x6.jpeg", the function call looks like this:

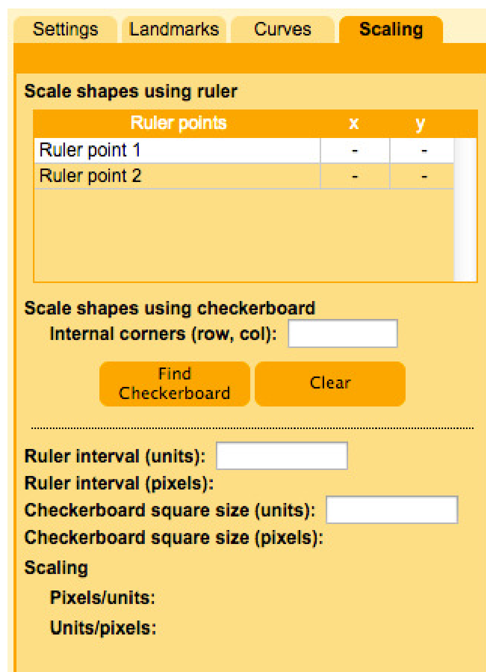
```
# Open the digitizing application
digitizeImages(image.file='7x6.jpg', shapes.file='7x6.txt')
```



The StereoMorph digitizing app.

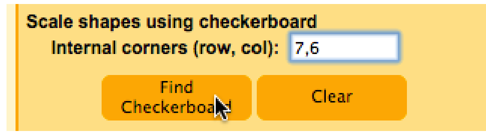
Once the app launches you should see the image on the left side of the window and a control panel on the right side. The Digitizing photographs section will explain the features of the app in more detail. For this section we will just measure the checkerboard square size and digitize points on the ruler.

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



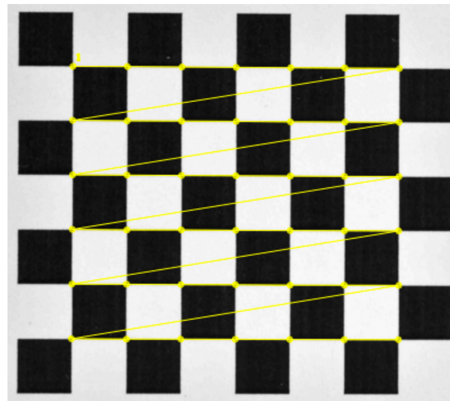
The Scaling panel of the digitizing app.

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”.



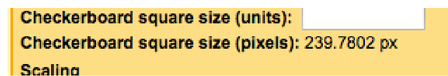
Entering the number of internal corners.

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.

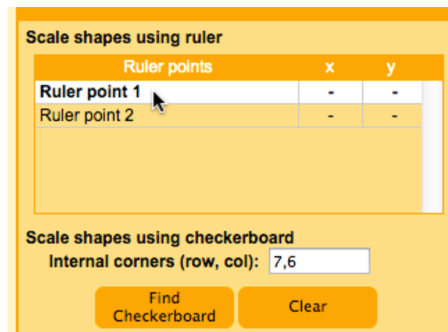


The detected corners displayed on top of the image.

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.



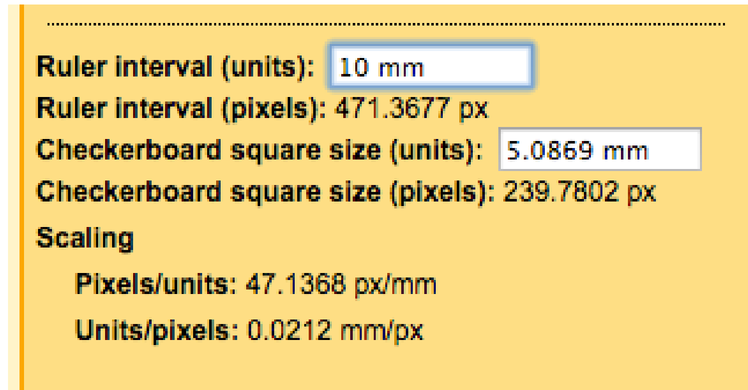
9. Then move your cursor to the ruler mark at 12 cm in the image and double-click. This will create a landmark at that position.

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). For this 7x6 checkerboard, the square size was 144 pixels and the checkerboard was printed at 10% scaling. Based on the equations for based on the DPI and printer scaling we would expect the printed square size to be 0.2 inches (5.08 mm). Note that our precision measurement differs by only 0.007 mm (7 microns) from what we expected. 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.

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.

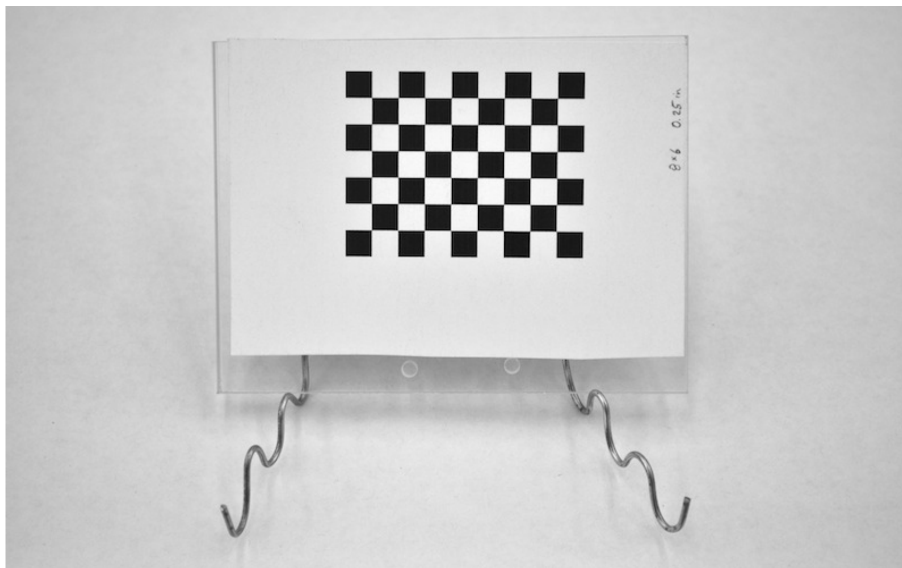
12. Click "Save" to save the checkerboard corners, ruler points and scaling data to the file you specified using the *shapes.file* parameter. If you'd like to refer back to the square size measurement you can read either re-launch the digitizing app using digitizeImages() with the same input parameters (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.

4.5 Creating a checkerboard stand

Materials needed for this section:

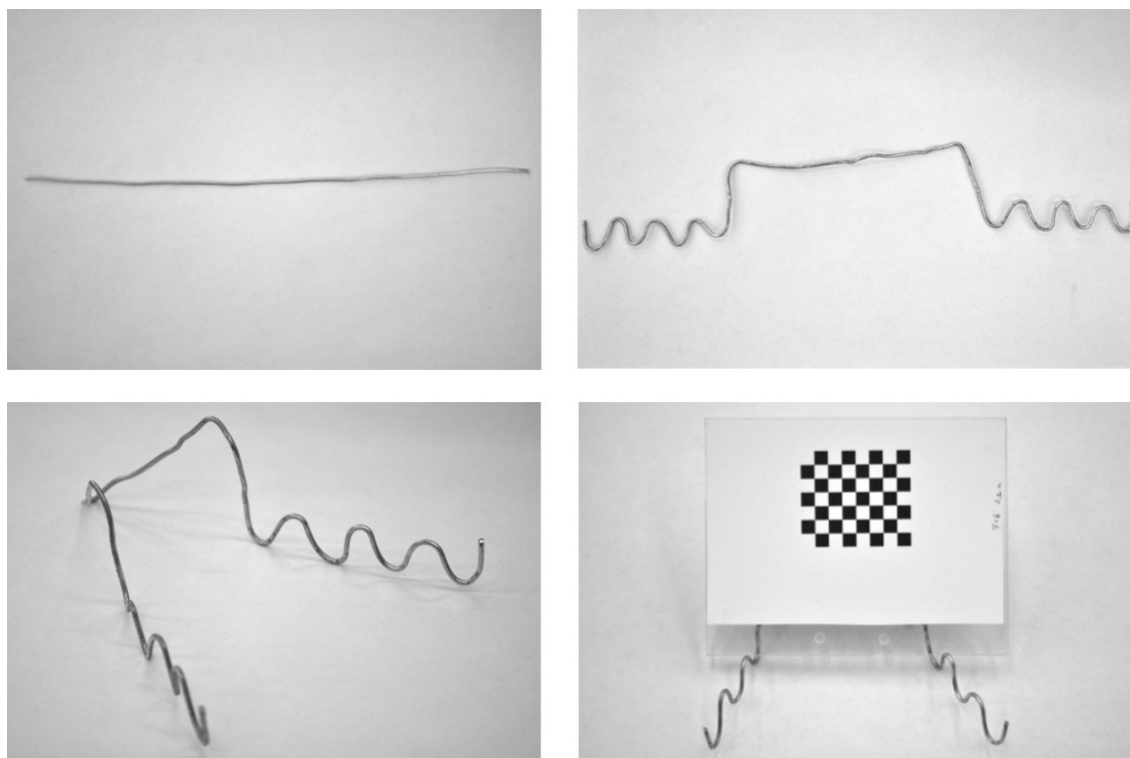
- Needle-nose pliers
- Thick wire (around 16 gauge)

If you are using StereoMorph for stereo photography, it's useful to have a stand that will hold the checkerboard in place while you take calibration photos with multiple cameras.



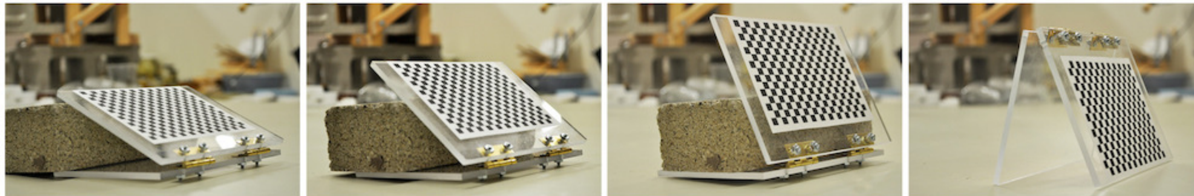
A simple checkerboard stand.

If you were to simply hold the checkerboard with your hand you would have to ensure the cameras took photographs at the exact same time. For stereo video, the checkerboard can be moved manually since the videos will have to be synchronized anyway.



Constructing a simple stand by bending a single piece of wire.

The type of stand you'll need depends on the size of your stereo setup. If your checkerboard isn't too heavy, you can create a simple stand with some wire and pliers as shown above. For a large, heavy checkerboard you may need to attach the hard surface to another object via a hinge and use heavy objects to prop the checkerboard at different angles.



A hinged stand.

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 it is possible to obtain three-dimensional information on position and shape.

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.

5.1 Arrangement for stereo photography

Materials needed for this section:

- 2 cameras (see Choosing cameras for stereo photography)
- Camera remotes
- 2 sturdy tripods
- Tape

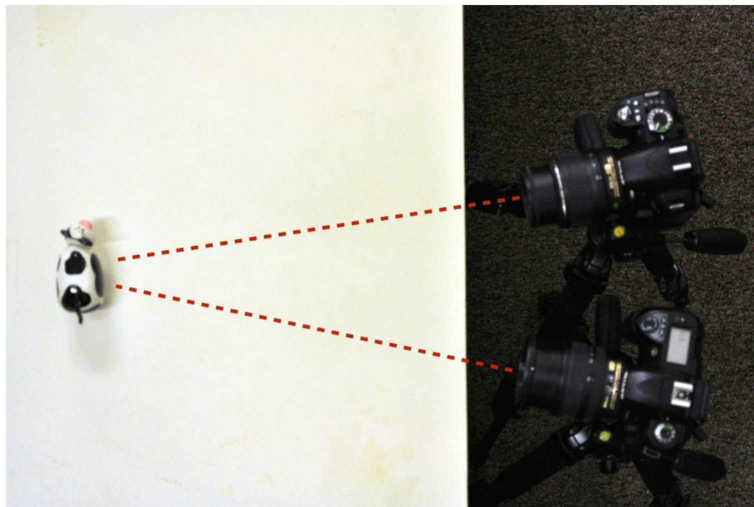
General considerations in arranging the cameras:

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.

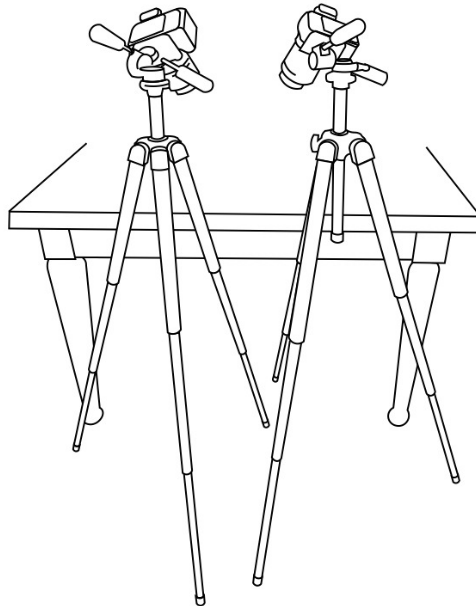
you will have high reconstruction accuracy (together, the two views give you full information on a point's position along all three axes) however the views will be so divergent that it will be difficult to identify the same point in both views. A point visible in one view may not even be visible in the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

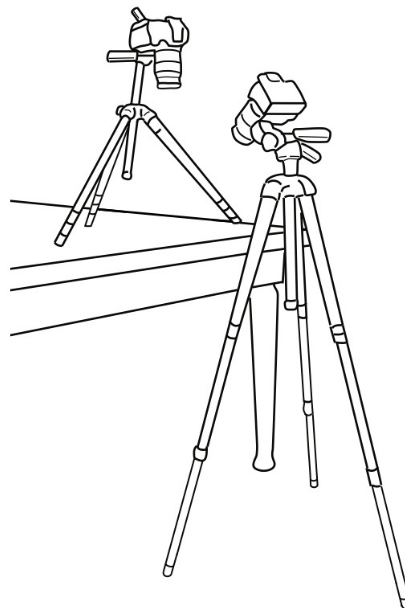
it's much easier to find the same point in both views (the views are nearly the same), however these slight differences in position are now the only information available on the point's position along the depth axis (orthogonal to the image planes). In practice, I've found that cameras positioned with 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.



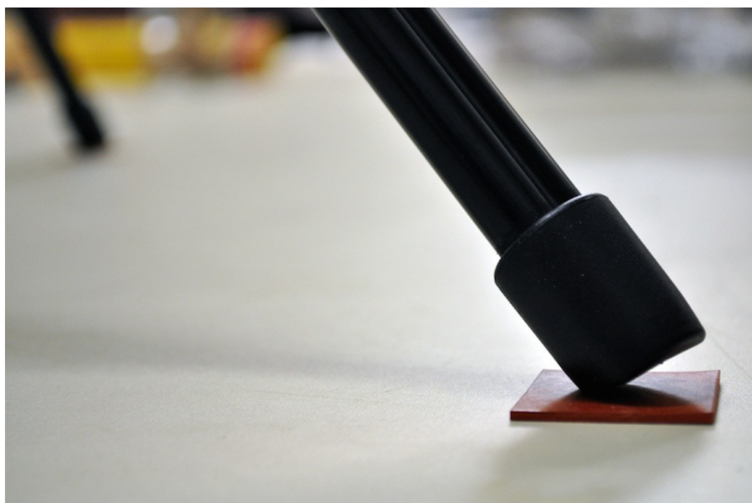
An alternative camera arrangement.

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.

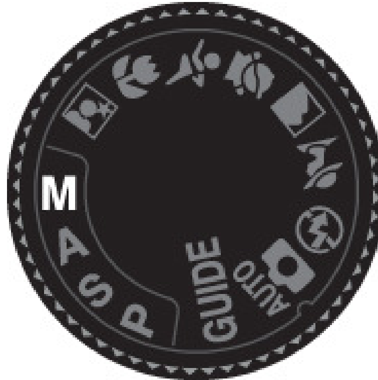
Ensure tight connections in the tripod and between the tripod and camera.

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.

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.

Whether you are using stereo photographs or video, 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.

6.1 General calibration steps and parameters

Calibration in StereoMorph is accomplished using a single function, calibrateCameras(). As of version 1.6, this function works with both photographs and synchronized video. While the parameters will differ depending on the camera arrangement and the type of input (i.e. photograph versus video), calibrateCameras() follows four general steps:

1. *Detect checkerboard corners*: The function will try to find the specified number of checkerboard corners in each input image.
2. *Estimate undistortion coefficients*: (Optional) If specified, the function will solve for coefficients that minimize standard lens distortion. This will not undistort the original photographs or video. But these coefficients will be used in the calibration and reconstruction process to minimize error due to lens distortion.
3. *Estimate DLT coefficients*: The function will estimate a set of optimized DLT coefficients that can be used to reconstruct points and curves from each view into 3D.
4. *Estimate calibration accuracy*: The function will estimate the calibration, using a separate set of images within the calibration set (if a sufficient number of images are supplied).

Because steps 1-3 are fairly time-consuming, the function will save the results of these steps. If you decide you would like to re-run any of these steps you can simply call calibrateCameras() again. Before running each of these three steps, you will be prompted whether you would like to keep the results from the previous run or re-run the step.

The calibrateCameras() function also has a general set of parameters used for both stereo photography and videography.

- *img.dir*: A folder containing the calibration images or videos.
- *sq.size*: The size of one checkerboard square (length along any one side) including the units (e.g. mm).
- *nx*: The number of internal corners along one dimension.
- *ny*: The number of internal corners along the other dimension (the choice of which is *nx* and *ny* is arbitrary but must be consistent throughout).
- *cal.file*: A file where the calibration results will be saved (the file will automatically be created if one doesn't already exist).

- *corner.dir*: A folder where the corners will be saved (the folder will be automatically be created if one doesn't already exist).
- *verify.dir*: (Optional) A folder where images will be saved that show the detected corners (the folder will be automatically be created if one doesn't already exist).
- *error.dir*: (Optional) A folder where several error diagnostic plots will be saved (the folder will be automatically be created if one doesn't already exist).
- *undistort*: A logical (TRUE or FALSE) indicating whether undistortion coefficients should be estimated. Currently, this is only recommended for video input.

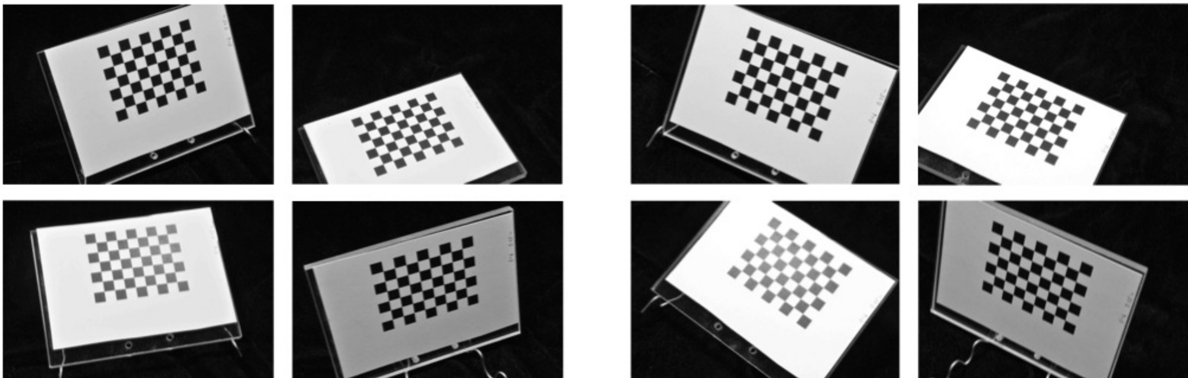
The following sections will show how to call calibrateCameras() using photographs and video. Before proceeding, make sure the StereoMorph library is loaded into the current R session.

```
# Load the StereoMorph package
library(StereoMorph)
```

6.2 Calibrating with photographs

The following will show you how to call calibrateCameras() using a set of photographs from two or more camera views.

1. Take 8-10 photographs from each camera of the checkerboard at different positions and angles within the calibration space.



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

2. Upload 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

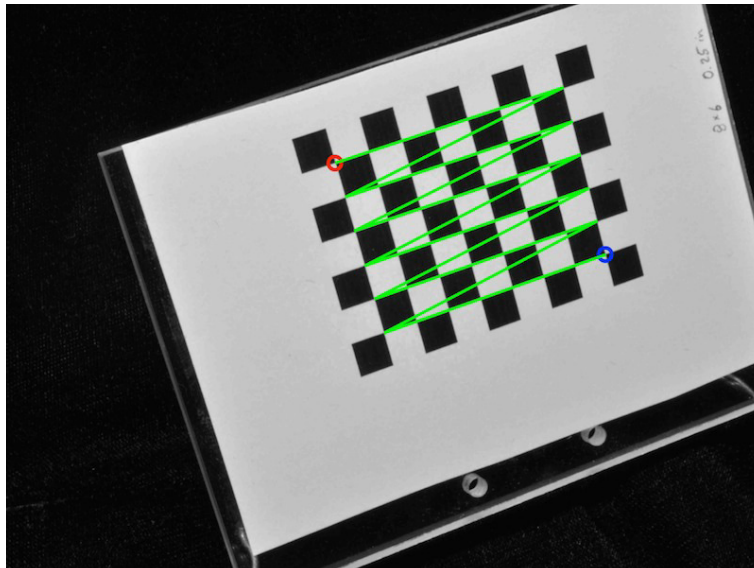
you are also photographing objects and you're transferring images via a camera memory card it's best to wait until you've taken all the photographs before upload the calibration images - taking the memory card out of the camera risks moving the cameras in which case they'll no longer be properly calibrated. 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.

If you'd like to work through the example below, you can download an example set of calibration images here (5 MB). Unzip this folder and move it to your R working directory.

3. Call the calibrateCameras() function. The call for the set of example images looks like this:

```
# Calibrate cameras from photographs
calibrateCameras(img.dir='Calibrate_images', sq.size='6.35 mm', nx=8, ny=6,
    cal.file='calibration.txt', corner.dir='Corners', verify.dir='Verify',
    error.dir='Errors')
```

The function will begin by trying to detect the corners in all of the calibration images. You can see the detected corners by looking in the "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 a 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 identified "first" corner (red circle) may not

be same corner across all the 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 example calibration images linked above, the calibration will not work since the corner order will not be consistent between the two views.

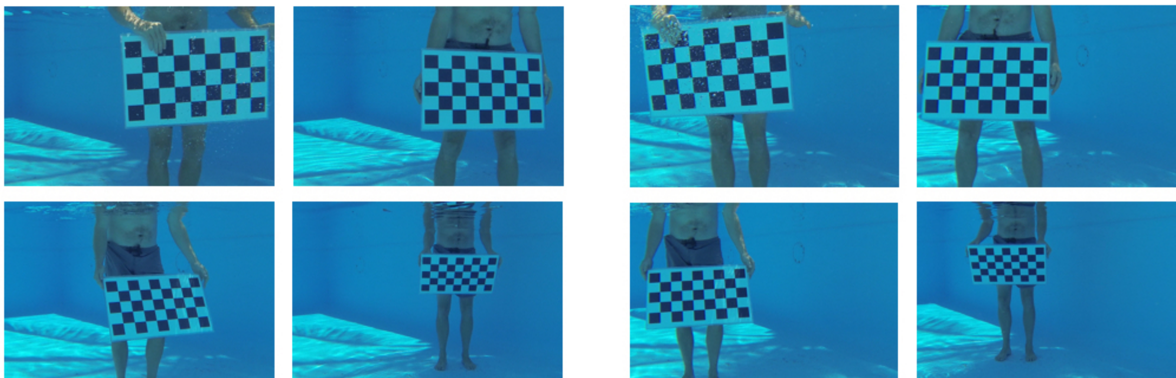
```
# Calibrate cameras from photographs when one view is 'upside-down'
calibrateCameras(img.dir='Calibrate_images', sq.size='6.35 mm', nx=8, ny=6,
    flip.view=TRUE, cal.file='calibration.txt', corner.dir='Corners',
    verify.dir='Verify', error.dir='Errors')
```

For the example set, the corners are detected in all the images. However, this is unusual. Usually, the detection will fail for a few images due to poor lighting or if portion of the checkerboard is cut off. These images will simply be ignored. It's a good idea to take at least 10 calibration images, so that if a couple images fail you will still have enough to get a good calibration.

6.3 Calibrating with videos

The following will show you how to call calibrateCameras() using a set of videos from two or more camera views. This differs only slightly from calibrating with photographs so much of the previous section will apply here as well. Your videos will need to be synchronized in order for the calibration to work properly (the first frame in all views must correspond to the same point in time). StereoMorph does not currently have tools to synchronize videos so this will have to be done using another program. Also, make sure that you have completed the steps in installing ffmpeg so that R can read the video files.

1. Record video of the checkerboard pattern being moved and rotated throughout the entire calibration space. It is essential that the checkerboard be moved throughout the entire space or else reconstruction errors will be higher in some areas than in others.



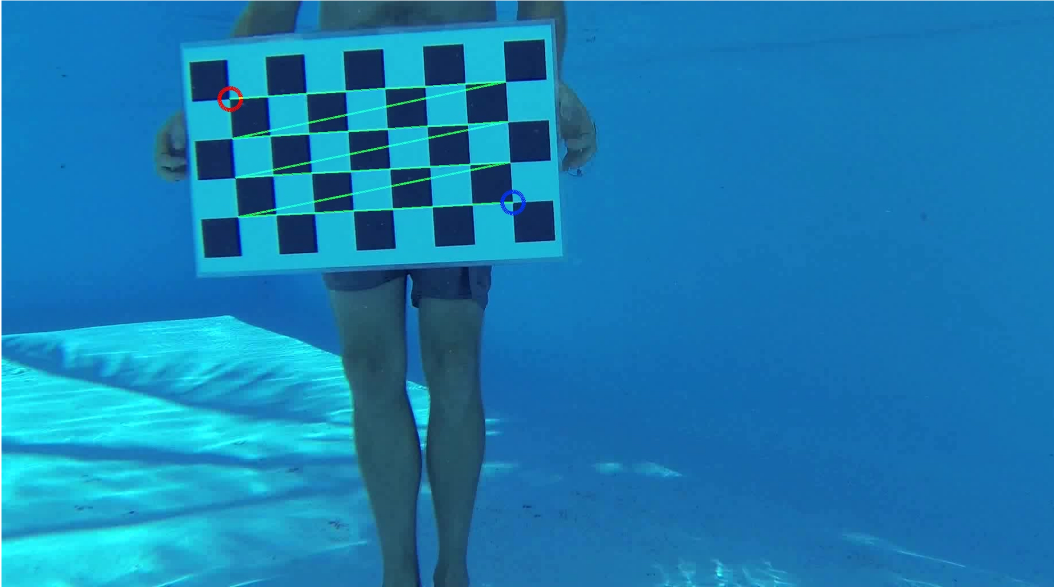
Four frames from video of a checkerboard at different positions and angles within the calibration space, recorded from left and right camera views. Image credit: Caine Delacy.

2. Upload the videos into a single folder, naming each video file with the corresponding camera view (e.g. "Left" and "Right" or "View 1" and "View 2"). If you'd like to work through the example below, you can download an example set of calibration videos here (19 MB). Unzip this folder and move it to your R working directory.

3. Call the calibrateCameras() function. The call for the set of example videos looks like this:

```
# Calibrate cameras from videos
calibrateCameras(img.dir='Calibrate_Videos', cal.file='calibration.txt',
    corner.dir='Corners', sq.size='63.42 mm', nx=8, ny=4,
    verify.dir='Verify', error.dir='Errors', undistort=TRUE,
    num.aspects.read=100, fit.min.break=2, nlm.calls.max=15,
    objective.min=0.8, max.sample.optim=30, num.sample.est=20,
    num.aspects.sample=8, num.sample.sets=3, objective.min.break=1.2)
```

The function will begin by trying to detect the corners in all of the calibration images. You can see the detected corners by looking in the "Verify" folder.



Detected corners in a video frame. The red circle indicates the first corner and the blue circle indicates the last corner. Image credit: Caine Delacy.

For the example set, the corners are detected for 83 frames in the left view and for 80 frames in the right view. However, there are only 68 frames for which the corners were detected in both views. Note that we can only use the corners detected in both views to estimate the calibration coefficients so we want to make sure that not only are the corners detected in enough frames but that they are detected in both views for a sufficient number of frames. Sixty-eight frames are more than sufficient for a good calibration.

6.4 Estimating calibration coefficients

After the checkerboard corners have been detected and (if specified) undistortion coefficients estimated, the function will begin estimating the DLT (calibration) coefficients. 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 sets. The parameters that control this are:

- *num.sample.est*: the number of aspects in the pool of images that will be used to estimate the coefficients
- *num.sample.sets*: the number of unique sets that will be tried
- *num.aspects.sample*: the number of aspects in each set that will actually be used to estimate the coefficients

The function prints these parameters before the estimation step. The parameter values used in the previous examples of calibration from photographs and from video should

work well for most cases but you can adjust these parameters if needed.

To estimate the DLT coefficients, calibrateCameras() uses an optimization routine that sequentially adds each checkerboard aspect (all views of a checkerboard in a particular position). As the number of parameters increases, the number of aspects increases and the optimization will take progressively more time up to the maximum number of aspects. For this reason, *num.aspects.sample* should not be greater than 9. Adding more aspects after 9 does not generally cause any further increase in accuracy.

6.5 Determining the calibration accuracy

The optimization should generally converge on a value (reconstruction error) less than 1 for each minimization. This value is in pixels 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. The DLT coefficients corresponding to the optimization with the lowest error will then be saved in the "calibration.txt" file as an 11x2 matrix.

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. This section will use the calibrating from photographs example to explain how to interpret the various calibration error tests.

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

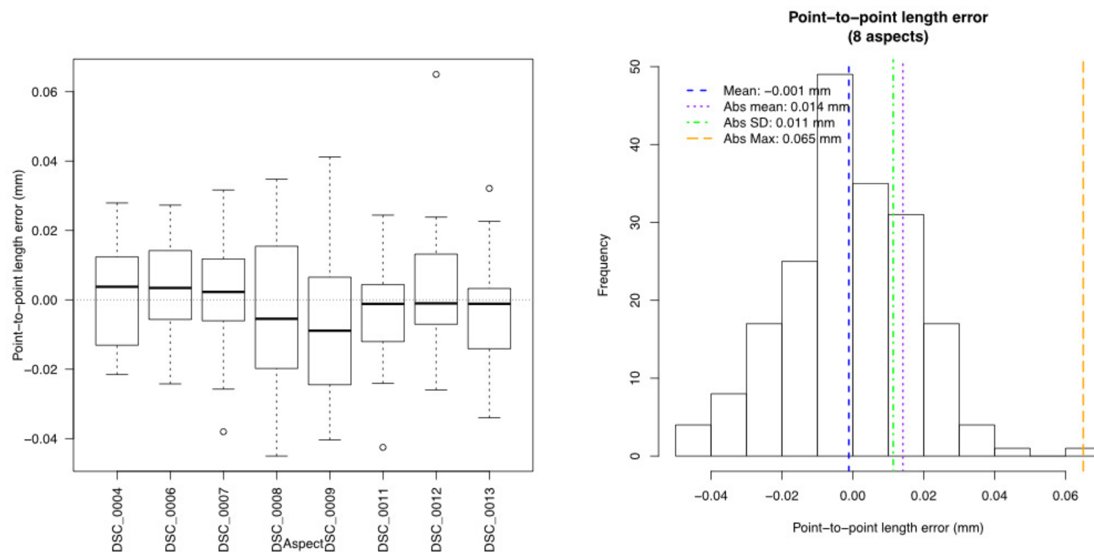
ideal distances are uniform and the same size as the square size. Since the corners in each pair are uniformly close together, their mean position (the mid-point) can be used to look at how IPD error varies as a function of position in the calibration volume.

- *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.

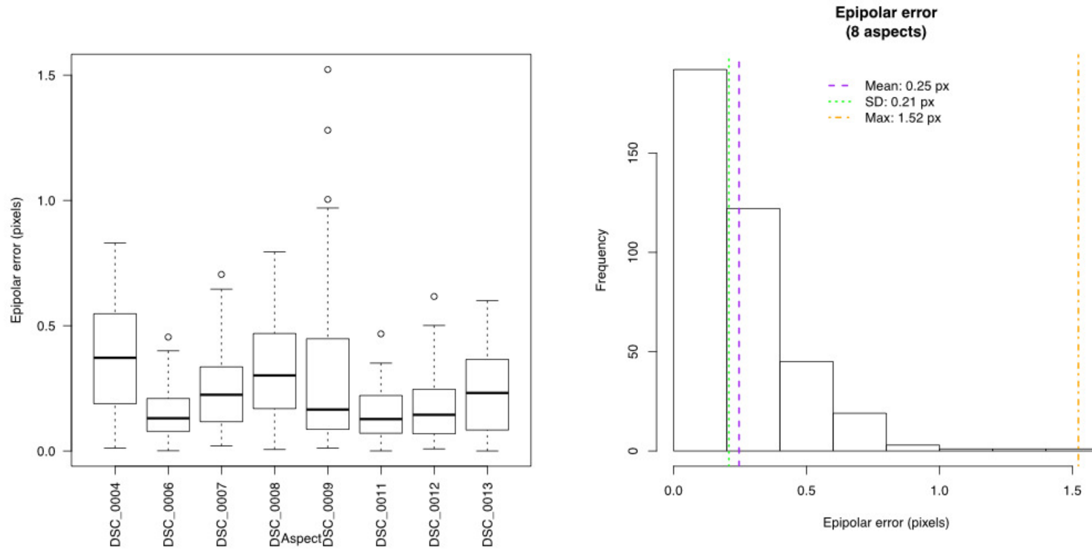
It's important to consider the magnitude of these errors relative to the pixel resolution of the cameras and the size of the calibrated volume. For the tutorial project the image resolution is around 30 microns/pixel (using 12 MP cameras). Since digitized coordinates are limited to pixel resolution the 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".

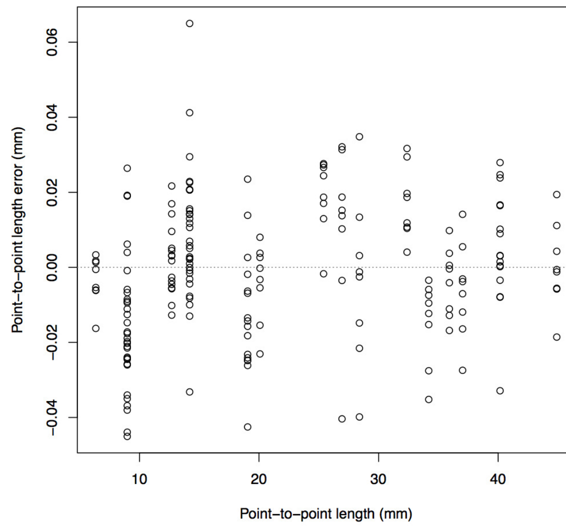
For example, the function creates a boxplot of the IPD errors, separated by aspect (left) and a histogram of the IPD errors (right) pooled from all aspects.



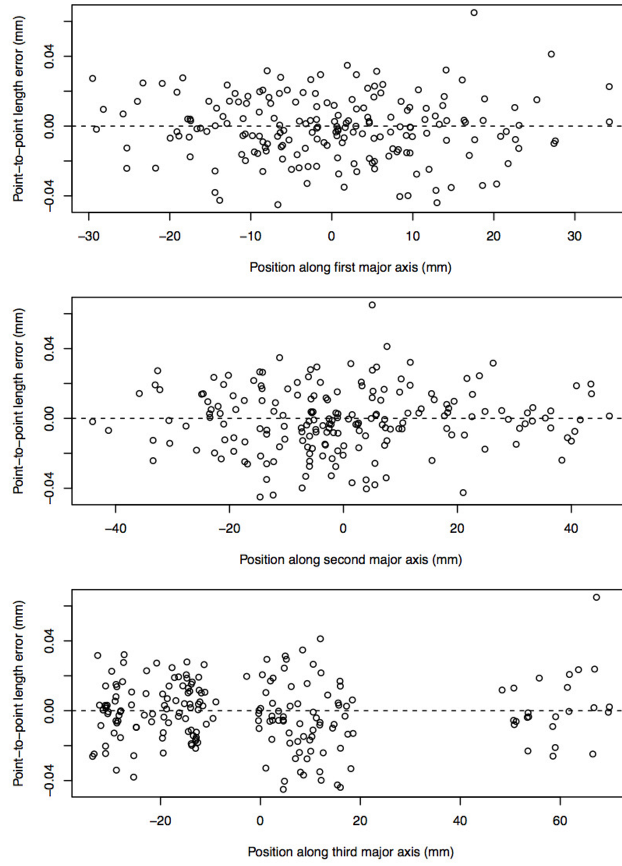
And the equivalent plots for epipolar error:



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.


Point–to–point length error (mm)
0.04
0.00
−0.04
−30
−20
−10
0
10
20
30
Position along first major axis (mm)
Point–to–point length error (mm)
0.04
0.00
−0.04
−40
−20
0
20
40
Position along second major axis (mm)
Point–to–point length error (mm)
0.04
0.00
−0.04
−20
0
20
40
60
Position along third major axis (mm)

7 Photographing objects

This section provides some extra details when using a stereo setup with photographs. Once the cameras are calibrated you can begin photographing objects. The number of objects that can be photographed is only limited by the time it takes to position and photograph each object. You can think of this as "cycling" the objects through the calibrated space. Note that it doesn't matter whether you calibrate the cameras before or after taking photographs of 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.

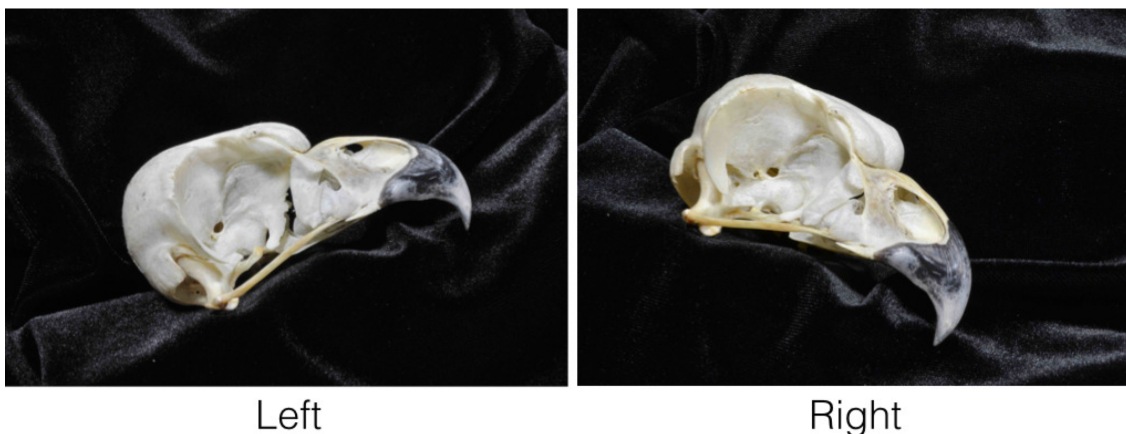
Black velvet works great as a uniform black background.

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.

If you'd like to see an example of this, download this stereo image set (4 MB). After you unzip the folder, you'll find an "Images" folder that contains left and right images of two different species of bird: a Great Horned Owl (*Bubo virginianus*) and an African Gray Parrot (*Psittacus erithacus*). For each species there are 3 different aspects of each skull, with the filenames ending in "a1", "a2", and "a3". The Great Horned Owl images are shown below.

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.

StereoMorph has a function that will automatically unify the landmark sets from different aspects, which will be detailed in Unification of reconstructed sets. But in order for the function to recognize different aspects, you'll need to follow a particular naming convention: each file must 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). Upload the photographs from each view (as .jpeg/.jpg files) into a separate folder. When you name each view folder be sure to use the same names that you used in the calibration step (e.g. "Left" and "Right" or "View 1" and "View 2").

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.

8 StereoMorph digitizing application

Once you have captured stereo images or video you can use the StereoMorph digitizing application to manually identify landmarks or curves that you want to reconstruct into 3D. Currently, StereoMorph doesn't have any automated feature recognition tools (except the checkerboard detection). But the digitizing application provides a user-friendly interface for identifying features yourself. This can be useful for features that would be otherwise difficult to identify or reconstruct automatically. The application 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 Digitizing video frames

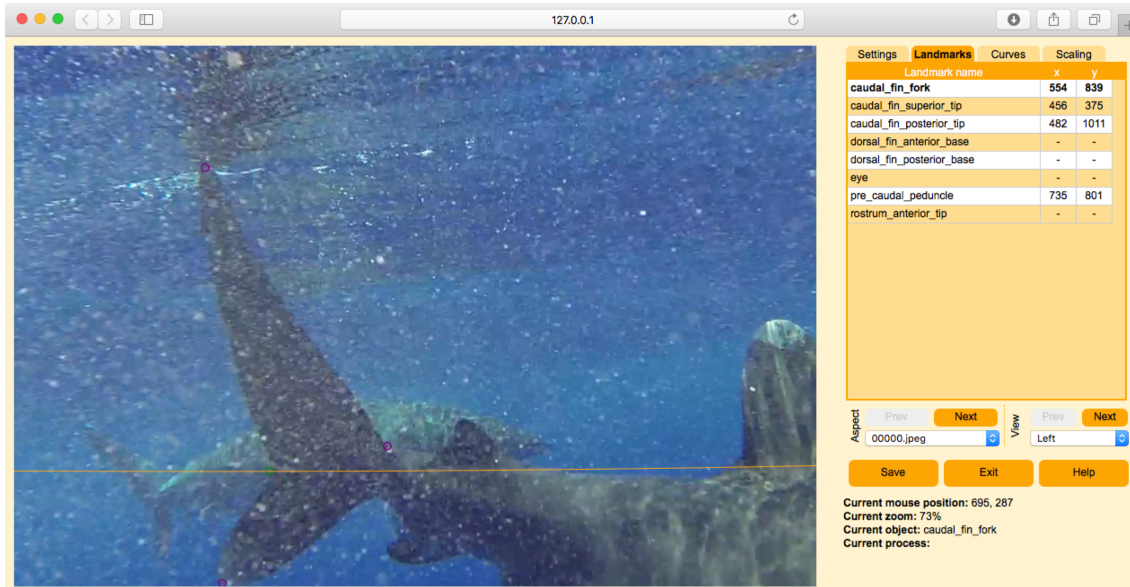
Currently, the StereoMorph digitizing application can only handle photographs. So in order to digitize videos you'll need to first extract the video frames as images and input these images to the digitizing application.

1. Load the StereoMorph library if it isn't already loaded.

```
# Load the StereoMorph package
library(StereoMorph)
```

2. Extract the frames you want to digitize using the StereoMorph function extractFrames(). You'll need to make sure that your video frames are synchronized. If your videos are not synchronized but you have a clear visual signal in the view of the cameras (e.g. turning on and off a light), you can use the extractFrames() function to extract "synchronized" frames. This will ensure that corresponding frames/images between different views have the same filename.

Once you have a set of images (i.e. frames) from both views this you can follow the same instructions as for photographs in the next section. If you'd like to work through the example below with stereo video frames, you can download an example set here (17 MB). This folder contains 50 frames from left and right video footage of freely swimming Oceanic white tip sharks (video courtesy of Caine Delacy and Mark Bond) along with a few extra files needed for the digitizing application (a calibration file, a list of points to identify, and a folder where the shape data will be saved). Unzip this folder and move the folder contents to your current R working directory.



Frame from stereo video frame example set (video courtesy of Caine Delacy and Mark Bond).

3. Launch the digitizing application using the digitizeImages() function.

```
# Launch the digitizing application
digitizeImages(image.file='Images', shapes.file='Shapes 2D',
   landmarks.ref='landmarks.txt', cal.file='calibration.txt')
```

For the remaining steps refer to the next section, starting with step 3. In the digitizing application you can skip to frame “00030.jpeg” to see some already digitized landmarks.

8.2 Opening the digitizing application

Once you have a set of images from two or more camera views you can use the StereoMorph digitizing application to manually identify the features that you want to reconstruct into 3D. We used this application in a previous section to precisely measure the square size of a checkerboard. If you’d like to work through the example below, you can download an example set of stereo images here (4 MB). Unzip this folder and move the folder contents to your current R working directory.

1. Load the StereoMorph library if it isn’t already loaded.

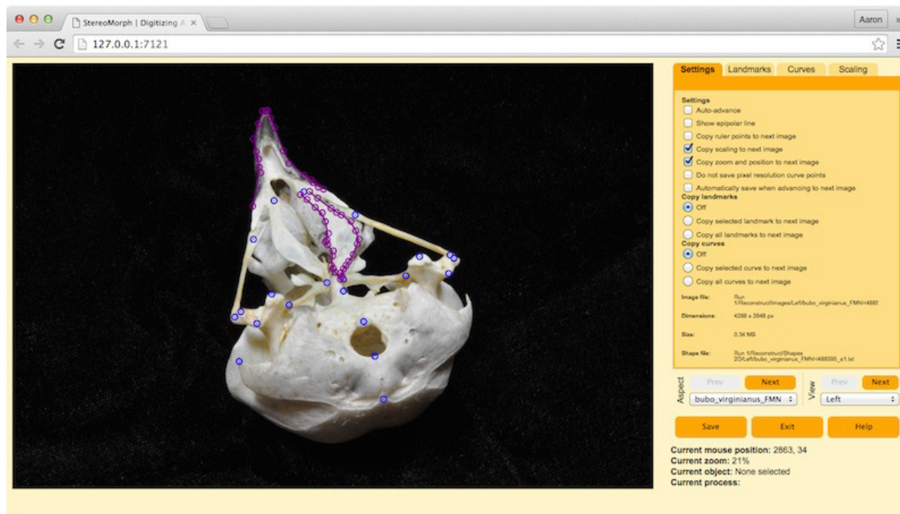
```
# Load the StereoMorph package
library(StereoMorph)
```

2. Launch the digitizing application using the digitizeImages() function.

```
# Launch the digitizing application
digitizeImages(image.file='Images', shapes.file='Shapes 2D',
   landmarks.ref='landmarks.txt', curves.ref = 'curves.txt',
   cal.file='calibration.txt')
```

3. These are the basic input parameters to digitizeImages() when using the app to digitize stereo image sets (refer to this stereo image set for an example):

- *image.file*: A folder containing the images to be digitized, separated into a folder for each view.
- *shapes.file*: 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 (see this example file).
- *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 (see this example file).
- *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.

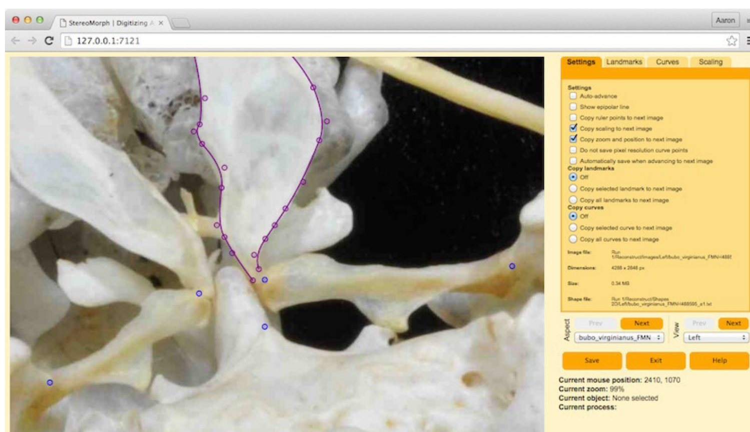


The StereoMorph digitizing app launched with a stereo image set.

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.

8.3 Digitizing landmarks

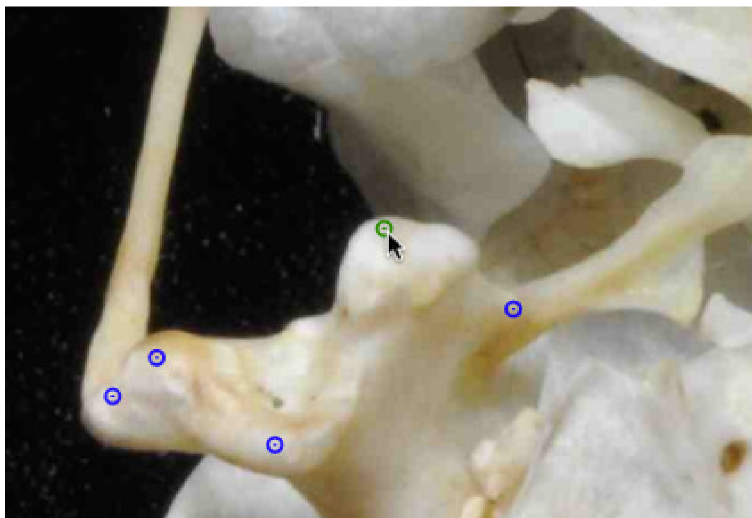
To digitize a landmark, click the corresponding row in the Landmarks tab. This will make that landmark the current object.

Settings | **Landmarks** | Curves | Scaling

Landmark name	x	y
mand_condyle_quadrate_lat_L	2927	1283
mand_condyle_quadrate_lat_R	1529	1662
mand_condyle_quadrate_pos_L	2918	1407
mand_condyle_quadrate_pos_R	1636	1741
mand_condyle_quadrate_uni_ant_L	-	-
mand_condyle_quadrate_uni_ant_R	-	-
mand_condyle_quadrate_uni_pos_L	-	-
mand_condyle_quadrate_uni_pos_R	-	-
nasalfrontalhinge_cranium_L	-	-
nasalfrontalhinge_cranium_R	-	-
occipital_condyle	2350	1728
otic_proc_tubercle_L	-	-
otic_proc_tubercle_R	-	-
opisthotic_process_L	-	-
opisthotic_process_R	1518	1995
orbital_proc_quadrate_sup_base_L	-	-
orbital_proc_quadrate_sup_base_R	-	-
orbital_proc_quadrate_inf_base_L	-	-
orbital_proc_quadrate_inf_base_R	-	-
orbital_proc_quadrate_distal_L	-	-
orbital_proc_quadrate_distal_R	-	-
palatine_dist_slide_L	-	-

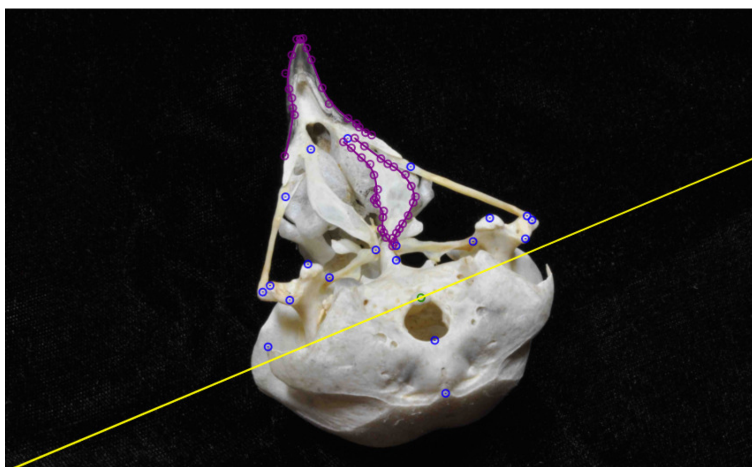
Move the cursor to where you want to add the landmark and double-click (keyboard shortcut: 'x'). If you've already placed a landmark you can also select it by placing your cursor over the landmark and double-clicking.

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.

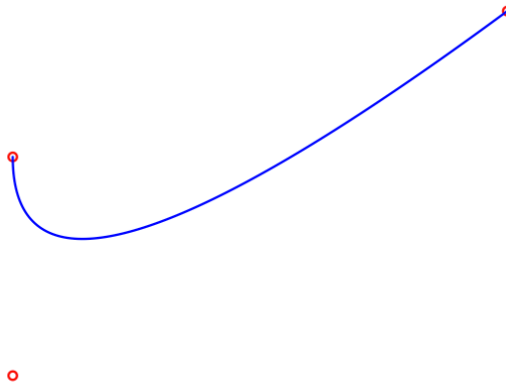
This line represents the line along which the landmark should fall if it corresponds exactly to the point digitized in the other view. Recall the epipolar error from the section on

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 properly, 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".

8.4 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.



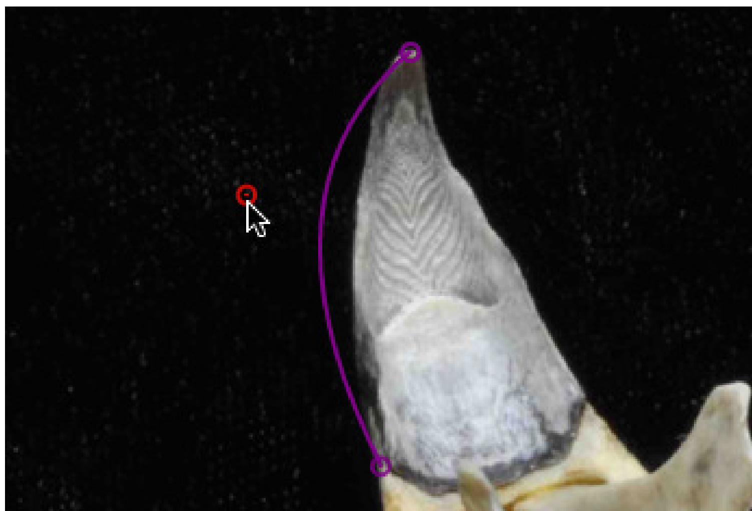
A 3-point Bézier curve with control points (red) and curve points (blue).

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.

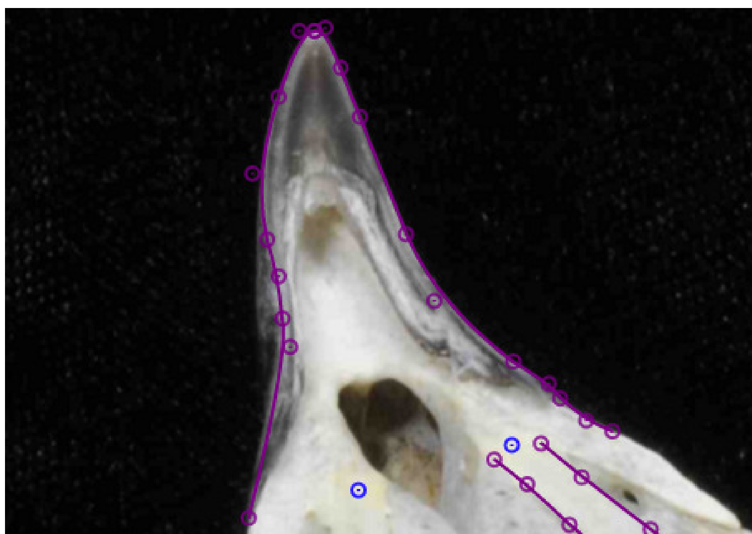
Position the curve start point on the image by double-clicking as described previously for Digitizing landmarks. Then select the curve end point and position this at the end of the curve. Now all that remains is to "fill in" a curve between these points. Select the empty row (with "-"'s) between the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

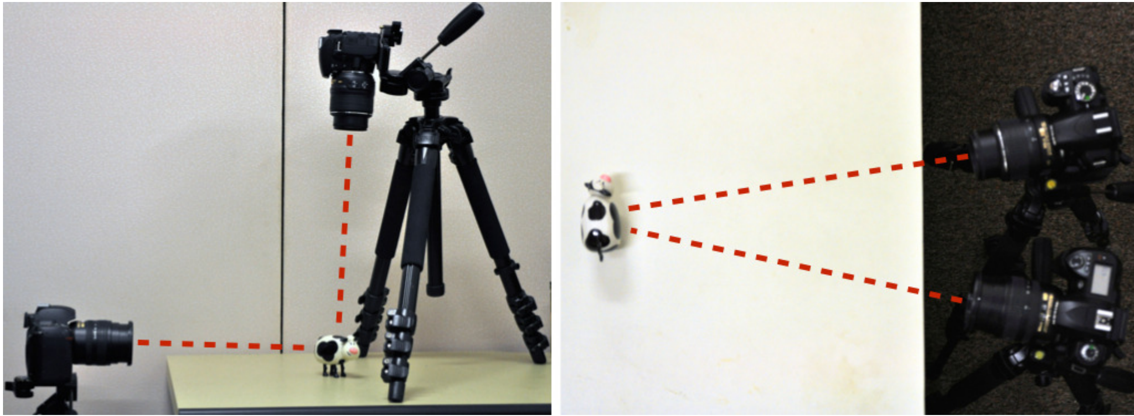
Stereo reconstruction requires corresponding points between different views. Curves complicate this a bit because while the first and last point are clearly corresponding, how do the points along one curve correpond to those along the other? You might think it would be as simple as matching points that 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".

8.5 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. If you're digitizing video frames, you can use the aspect buttons or drop-down menu to move among frames.

8.6 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
- <: Move to previous image (in Aspects)
- >: 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

Once you have landmarks or curves digitized in two or more views you can reconstruct them into 3D. This section will explain how to do this using the reconstructStereoSets() function. Before preceding, load the StereoMorph library, if it itsn't already loaded.

```
# Load the StereoMorph package
library(StereoMorph)
```

9.1 Reconstructing landmarks

If you've only digitized landmarks and do not have multiple aspects per object, you can use reconstructStereoSets() to reconstruct all the landmarks for a set of digitized images. If you'd like to try this with an example you can download this stereo landmark set (10 KB), previously used in the digitizing video frames section. Unzip the folder's contents into your current R working directory.

1. Call reconstructStereoSets() with the following three parameters.

```
# Reconstruct all digitized landmarks in Shapes 2D folder
reconstructStereoSets(shapes.2d='Shapes 2D', shapes.3d='Shapes 3D',
   cal.file='calibration.txt')
```

These parameters refer to:

- *shapes.2d*: A folder containing digitized (2D) shape data, separated into different folders by view. This is the same folder of shape files you used with the digitizing application (see digitizing photographs).
- *shapes.3d*: 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().

The reconstructStereoSets() function will reconstruct all the landmarks digitized in at least two views and save these into the "shapes.3d" folder using the same name as the corresponding 2D shape files. The reconstructed landmarks will be in the same units used in the calibration step (for the example above this is mm).

2. StereoMorph saves these files in a special xml-format that can be read using the StereoMorph function readShapes(). Use readShapes() to read these reconstructed landmarks from the 3D shapes folder.

```
# Read all digitized landmarks in Shapes 3D folder
shapes <- readShapes(file='Shapes 3D')

# Print all the recontructed landmarks as an array
shapes$landmarks
```

If there are multiple files in the "Shapes 3D" folder then the landmarks will be returned as an array in which the first dimension is the landmarks, the second dimension is the xyz-coordinates, and the third dimension is every image or frame. Note that when reading all landmarks from several files in this way, if a landmark is not digitized in every image it will be NA for those images. Thus, this landmark array will always be as long as all of the unique landmarks digitized in the entire image set.

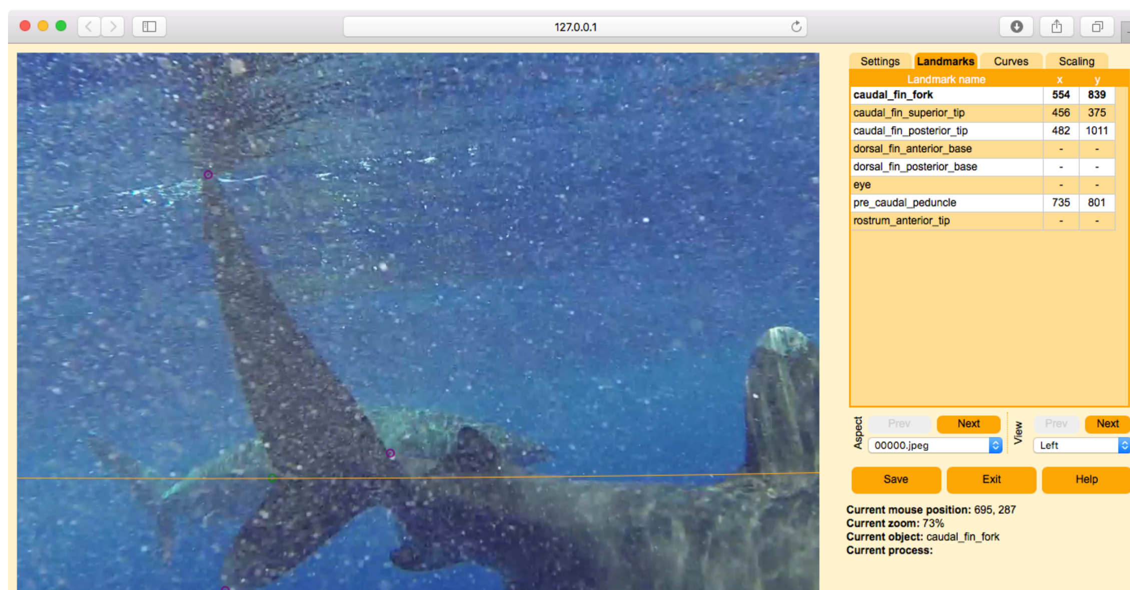
3. For example, you can use the following code to access different landmarks of interest.

```
# Access a landmark from the first image
shapes$landmarks['caudal_fin_fork', , 1]

# Access a landmark from all images
shapes$landmarks['caudal_fin_fork', , ]

# Access al landmarks from the first image
shapes$landmarks[ , , 1]
```

The reading shape data section has more examples of how to use the readShapes() function.

9.2 Measuring 3D lengths

Currently, StereoMorph does not have a tool to measure 3D lengths directly in the digitizing application. However, this can be easily done by using landmarks to define the two end points of the length you would like to measure. This adds flexibility as 3D lengths can then be measured among any identified landmarks.



3D lengths can be measured by using landmarks to define the end points of the length to be measured. 3D lengths can be measured among any of these identified 4 landmarks

(shown in small purple and green circles). Image is from stereo video frame example set (courtesy of Caine Delacy and Mark Bond).

1. Reconstruct and read in the 3D landmarks (see the preceding section, reconstructing landmarks).

```
# Reconstruct all digitized landmarks in Shapes 2D folder
reconstructStereoSets(shapes.2d='Shapes 2D', shapes.3d='Shapes 3D',
   cal.file='calibration.txt')

# Read all digitized landmarks in Shapes 3D folder
shapes <- readShapes(file='Shapes 3D')
```

2. Use the StereoMorph function distancePointToPoint() to find the distance between any two landmarks.

```
# Find distance between two landmarks in the first frame
distancePointToPoint(shapes$landmarks[c('caudal_fin_posterior_tip',
   'caudal_fin_superior_tip'), , 1])
```

9.3 Reconstructing landmarks and curves

If you've digitized both landmarks and curves you can use reconstructStereoSets() to reconstruct both at the same time. If you'd like to try this with an example you can download this stereo landmark and curve set (10 KB), previously used in the digitizing photographs section. Unzip the folder's contents into your current R working directory.

1. Call reconstructStereoSets().

```
# Reconstruct landmarks and curves
reconstructStereoSets(shapes.2d='Shapes 2D', shapes.3d='Shapes 3D',
   cal.file='calibration.txt', even.spacing='even_spacing.txt')
```

For curves, the function needs an additional parameter, *even.spacing* that specifies how many points will be reconstructed along each curve. This can be 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 (see this example file). You can also make *even.spacing* a single integer if you want each curve to have the same number of points.

If you've photographed the same object in different orientations (if you have multiple aspects) the landmarks and curves in each aspect will be in different coordinate systems. These can be aligned with one another based on shared landmarks between sets into a single set, which is referred to here as "unification". By default, the reconstructStereoSets() function performs both reconstruction and unification as long as aspect labels have been added to all of the image names (e.g. "_a1", "_a2", etc.).

If the input parameter *print.progress* is TRUE (the default), the function prints each of the steps along with the associated errors. If you try this with the example stereo shape set, you'll see that one of the objects (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) in the example stereo set 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.

2. 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
reconstructStereoSets(shapes.2d='Shapes 2D', shapes.3d='Shapes 3D',
   cal.file='calibration.txt', even.spacing='even_spacing.txt',
   set.names=c('psittacus_erithacus_FMNH312899'))
```

3. 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
reconstructStereoSets(shapes.2d='Shapes 2D', shapes.3d='Shapes 3D',
   cal.file='calibration.txt', even.spacing='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.

4. 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
reconstructStereoSets(shapes.2d='Shapes 2D', shapes.3d='Shapes 3D',
   cal.file='calibration.txt', even.spacing='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.

9.4 Reading shape data

The reconstructed landmarks and curve points are saved into the 3D shapes folder. StereoMorph saves these files in a special xml-format that can be read using the StereoMorph function readShapes(). If you'd like to try using readShapes() with an example dataset you can download this stereo landmark and curve set (10 KB), previously used in the digitizing photographs section. Unzip the folder's contents into your current R working directory.

1. Input a single file to readShapes to read all the shape data from that file.

```
# Read 3D landmarks and curves from a particular file (object)
shapes <- readShapes(file='Shapes 3D/bubo_virginianus_FMNH488595.txt')
```

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:

```
# Print the contents of shapes
shapes
```

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):

```
# Print the landmarks
shapes$landmarks
```

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

```
# Print the xyz-coordinates of the cranium_occipital landmark
shapes$landmarks['cranium_occipital', ]
```

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:

```
# Print the curve names
names(shapes$curves)
```

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*).

```
# Print the upper_bill_tomium_L curve points
shapes$curves$upper_bill_tomium_L
```

2. By inputting multiple filenames into readShapes(), you can read all the shape data from those files at once to create a landmark array or larger list structures of curves. Specify two files to be read by readShapes():

```
# Specify multiple shape files
file <- c('Shapes 3D/bubo_virginianus_FMNH488595.txt',
  'Shapes 3D/psittacus_erithacus_FMNH312899_a1.txt')

# Read shape files
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.

```
# Print the contents of shapes
shapes
```

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

```
# Print the xyz-coordinates of cranium_occipital across the files
shapes$landmarks['cranium_occipital', , ]
```

or accessing all the landmarks for a particular file.

```
# Print all the landmarks for bubo_virginianus_FMNH488595
shapes$landmarks[, , 'bubo_virginianus_FMNH488595']
```

3. 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:

```
# Read all shape files in folder Shapes 3D
shapes <- readShapes(file='Shapes 3D')
```

10 Visualizing shape data

This section will demonstrate how to plot landmarks and curves using the R package 'rgl'. If you'd like to try using readShapes() with an example dataset you can download this stereo landmark and curve set (10 KB), previously used in the 3D reconstruction section. Unzip the folder's contents into your current R working directory.

1. Begin by loading the StereoMorph and rgl packages.

```
# Load the StereoMorph and rgl packages
library(StereoMorph)
library(rgl)
```

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

```
# Read shapes
shapes <- readShapes(file='Shapes 3D/bubo_virginianus_FMNH488595.txt')

# Save the landmark matrix into a separate variable
lm <- shapes$landmarks
```

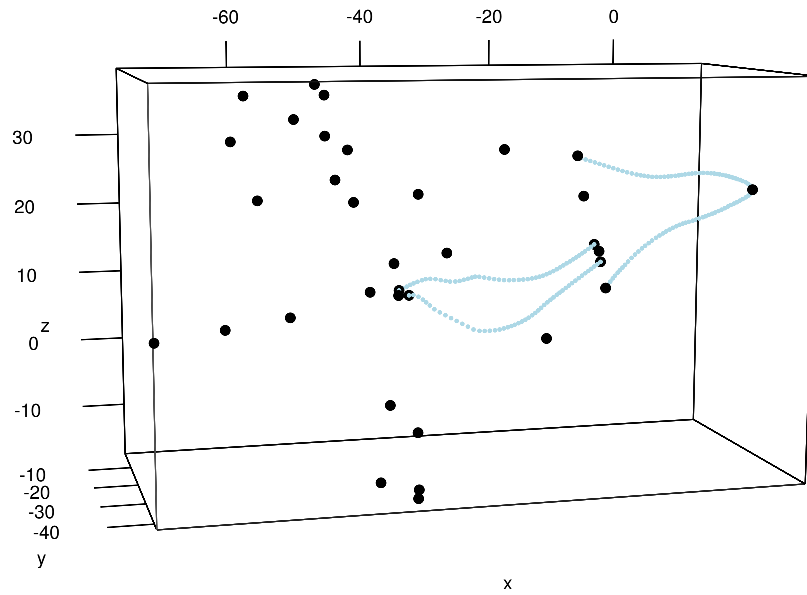
3. Define the ranges of x,y,z values of the plots as a variable. This will be used to set the aspect ratio of the plot box. Otherwise, plot3d() will plot the points in a box with equal lengths on all sides.

```
# Define range
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)
```

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

```
# Plot curve points
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.

If you'd like to try using reflectMissingShapes() with an example dataset you can download this stereo landmark and curve set (10 KB). Unzip the folder's contents into your current R working directory.

1. Begin by loading the StereoMorph library.

```
# Load the StereoMorph package
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
```

2. 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:

```
# Reflect missing shape data
rms <- reflectMissingShapes(shapes='Shapes 3D/bubo_virginianus_FMNH488595.txt',
  file='Reflected/bubo_virginianus_FMNH488595.txt', average=TRUE)
```

If the directory in "file" doesn't exist, one will automatically be created. 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:

```
# Print reflected landmarks
rms$landmarks
```

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

```
# Set shape files to reflect
specimen <- c('bubo_virginianus_FMNH488595.txt',
   'psittacus_erithacus_FMNH312899_a1.txt')

# Reflect missing shape data for specified files
rms <- reflectMissingShapes(shapes=paste0('Shapes 3D/', specimen),
   file=paste0('Reflected/', specimen), average=TRUE)
```

Currently, for this input type the function returns NULL.

3. 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.

```
# Reflect missing shape data for all files in a folder
rms <- reflectMissingShapes(shapes='Shapes 3D', file='Reflected',
   average=TRUE)
```

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

```
# Read shape file
shapes <- readShapes(file='Shapes 3D/bubo_virginianus_FMNH488595.txt')

# Reflect missing shape data for shape data
rms <- reflectMissingShapes(shapes=shapes, average=TRUE)
```

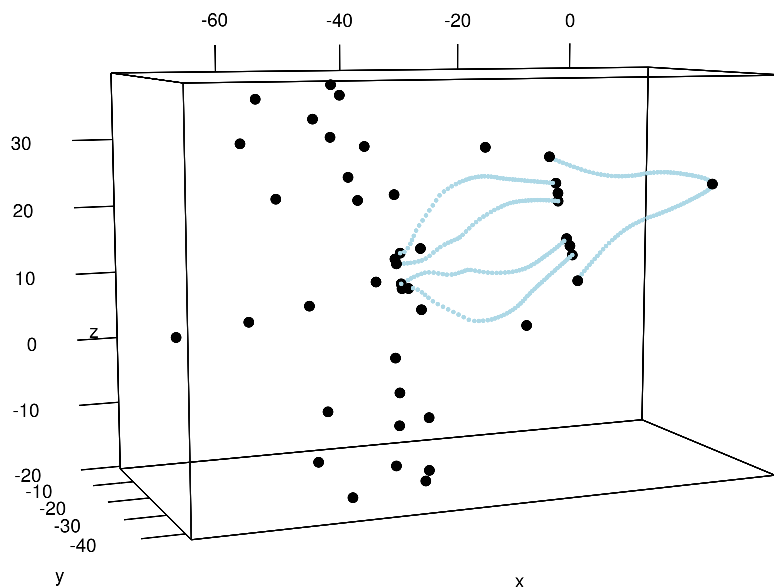
file can be omitted, in which case the function will not create a file for the reflected shape data and will only return a shapes list. *file* can only be omitted for single shape set input to reflectMissingShapes().

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

```
# Reflect missing shapes and print error summary
rms <- reflectMissingShapes(shapes=shapes, average=TRUE,
   print.progress=TRUE)
```

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

```
# Load the rgl package
library(rgl)

# Read shapes
shapes <- readShapes(file='Reflected/bubo_virginianus_FMNH488595.txt')

# Get landmarks
lm <- shapes$landmarks

# Set landmark range
r <- apply(lm, 2, 'max') - apply(lm, 2, 'min')

# Plot landmarks
plot3d(lm, aspect=c(r/r[3]), size=7)

# Plot curves
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.

If you'd like to try using alignShapesToMidline() with an example dataset you can download this stereo landmark and curve set (10 KB). Unzip the folder's contents into your current R working directory.

1. Begin by loading the StereoMorph library.

```
# Load the StereoMorph package
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
```

2. 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:

```
# Align shapes to midline
asm <- alignShapesToMidline(shapes='Reflected/bubo_virginianus_FMNH488595.txt',
  file='Aligned/bubo_virginianus_FMNH488595.txt')
```

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:

```
# Print landmarks
asm$landmarks
```

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.

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

```
# Specify multiple specimens
specimen <- c('bubo_virginianus_FMNH488595.txt',
     'psittacus_erithacus_FMNH312899_a1.txt')

# Align shapes to midline
asm <- alignShapesToMidline(shapes=paste0('Reflected/', specimen),
   file=paste0('Aligned/', specimen))
```

Currently, for this input type the function returns NULL.

4. 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.

```
# Align shapes to midline
asm <- alignShapesToMidline(shapes='Reflected', file='Aligned')
```

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

```
# Read shapes
shapes <- readShapes(file='Reflected/bubo_virginianus_FMNH488595.txt')

# Align shapes to midline
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().

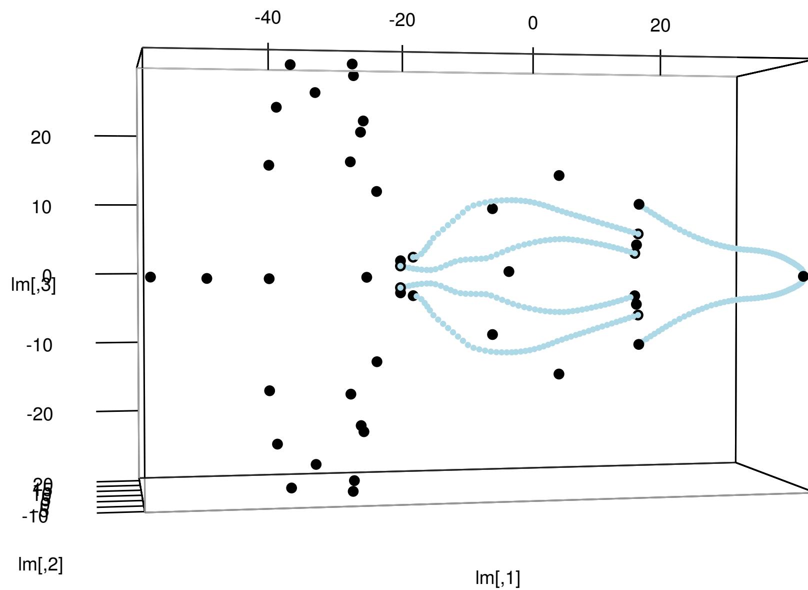
7. Lastly, *print.progress* can be set to TRUE in order to view the alignment errors:

```
# Align shapes to midline and print error summary
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.

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

```
# Load the rgl package
library(rgl)

# Read shapes
shapes <- readShapes(file='Aligned/bubo_virginianus_FMNH488595.txt')

# Get landmarks
lm <- shapes$landmarks

# Set landmark range
r <- apply(lm, 2, 'max') - apply(lm, 2, 'min')

# Plot landmarks
plot3d(lm, aspect=c(r/r[3]), size=7)

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



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

13 Additional features

13.1 Extracting video frames

Since the StereoMorph digitizing application cannot read videos directly, the frames must be extracted from the videos and input to the digitizing application as images. Frames can be extracted in StereoMorph using the function extractFrames(). Before using extractFrames() be sure that you have completed the steps in installing ffmpeg so that R can read the video files. If you'd like to work through the example below, you can download an example video file here (10 MB). Note that in Safari you may have to right-click and select "Download Video" rather than using File/Save As.

1. If you are unsure of how many frames the video has or which frames you would like to extract, you can call extractFrames() without any parameters.

```
# Extract frames from a video
extractFrames()
```

The function will prompt you to enter a video file that you want to extract frames from. Either type the video file path or simply click and drag the video file into the R console and the file path will be copied over.

2. Next, the function will prompt you to ask where you want to save the extracted frames. Either type a file path to a folder or simply click and drag a folder into the R console and the file path will be copied over.

3. The function will then tell you the number of total video frames and ask you to enter the frames that you want to extract. Note that the first frame is frame 0. The example video has 100 frames total, so you can enter any frames between 0 and 99. You can specify the frames you want to extract by entering a single number, using the ':' symbol or using the c() or seq() functions:

```
# To extract a single frame
2

# To extract all frames between 3 and 10 (including 3 and 10)
3:10

# To extract every third frame between 5 and 20 (including 5 and 20)
seq(5, 20, by=3)

# To extract a particular set of frames
c(0, 4, 5, 9)
```

By default, if the number of frames you set to extract is greater than 100, the function will list all the frames to be extracted and issue a second prompt to ask if you are sure (to avoid extracting thousands of frames by mistake). This warning can be turned off by

setting the *warn.min* parameter to any number larger than the total number of frames in the video (e.g. 1000000).

4. If you already know the input parameters in advance and want to use the function without any prompts, you can just set these parameters in the function call. Create a folder named "Frames" in your current R working directory.

5. Call extractFrames with all the input parameters.

```
# Extract the first 20 frames from the example video
extractFrames(file='Example_video.mov', save.to='Frames', frames=0:20)
```

13.2 Extracting synchronized frames

If you are extracting frames from two or more videos that are not synchronized you might want to make sure that the extracted frames have the same names. By default, extractFrames() will name the extracted frames the name of the corresponding frame (i.e. frame 50 of a video will be named "000050.jpeg"). extractFrames() adds enough zeros to the start of the name to maintain the same filename length for all frames.

However, if two videos are not synchronized but you know the time offset (e.g. 30 frames) you might want to extract frames 0-49 from one video and frames 30-79 from the second video. If you were to do this, the names of the extracted images from the first video would be "000000.jpeg" to "000049.jpeg" and those from the second video would be "000030.jpeg" to "000079.jpeg". In order to "synchronize" the video frames we can name both sets "000000.jpeg" to "000049.jpeg". To override the default names created by extractFrames(), you can use the *names* parameter.

For example, the following command will extract frames 30-79 and name them "000000.jpeg" to "000049.jpeg". The formatC() function is used to add leading zeros to the numbers 0:49 to create a string of width, *width*.

```
# Extract the first 20 frames from the example video
extractFrames(file='Example_video.mov', save.to='Frames', frames=30:79,
  names=formatC(0:49, width=6, format='d', flag='0'))
```

StereoMorph currently does not contain any special tools to determine the time/frame offset between two videos. However, one simple way to determine this offset is to turn on and off a light in view of the cameras. You can then use the extractFrames() function to narrow down and identify the on/off frame in each view and thereby determine the offset.

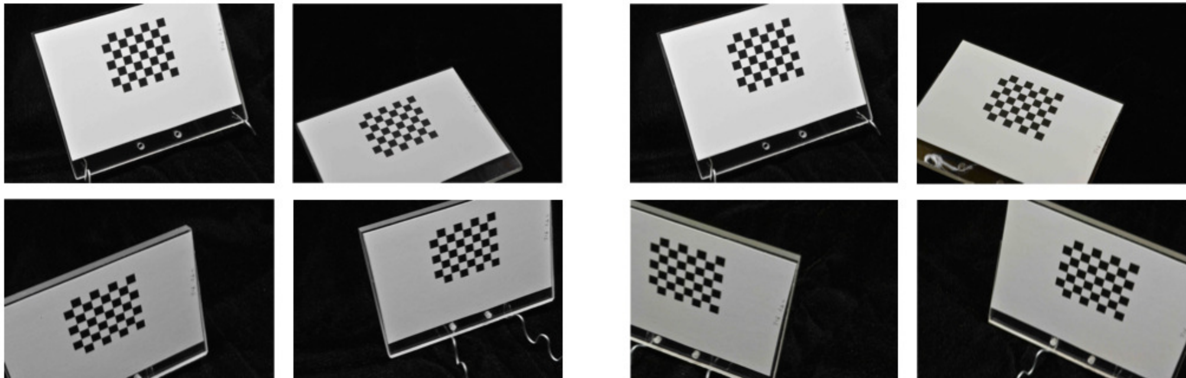
13.3 Testing the accuracy using a second checkerboard

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 cali-

bration 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()).

1. Begin by photographing a checkerboard in the calibration volume in the same manner as for the calibration step.



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

2. Upload the images into a folder, separating, the images from different views into two different folders (e.g. "Left" and "Right", "View 1" and "View 2") just like for the calibration step.

If you'd like to try out the following code with an example dataset you can download this test calibration image set (3 MB). This example set tests the scaling accuracy of the calibration performed in This example set uses a checkerboard that differs both in square size (5.080 mm versus 6.35 mm) and the number of internal corners (7x6 versus 8x6). Unzip the folder and move its contents into your current R working directory.

3. Make sure the StereoMorph library is loaded.

```
# Load the StereoMorph package
library(StereoMorph)
```

4. Call testCalibration().

```
# Test calibration against other checkerboard set
test_cal <- testCalibration(img.dir='Images', cal.file='calibration.txt',
   corner.dir='Corners', sq.size='5.080 mm', nx=7, ny=6, error.dir='Errors',
   verify.dir='Verify')
```

The basic input parameters are nearly identical to calibrateCameras(). Here is a brief description of each parameter:

- *img.dir*: The folder containing the checkerboard images, each view in a separate folder.
- *cal.file*: The calibration file previously created by calibrateCameras().
- *corner.dir*: 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.
- *error.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 folder where images will be saved that show the detected corners. If this folder does not exist, a new folder will automatically be created.

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 *error.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 software 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 StereoMorph and share your results that you include a citation. 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, Andrew George, Dallas Krentzel, Mark Westneat, Gavin Thomas, Chery Cherian, Sushma Reddy, Vincent Bonhomme, Stewart Edie, Yinan Hu, Stas Malavin, Ty Hedrick, Merrilee Guenther, Benjamin Rubin, and José Iriarte-Díaz.

I would like to extend my appreciation to Caine Delacy and Mark Bond, with whom I had the great opportunity to collaborate on calibrating an underwater GoPro setup using StereoMorph. This collaboration led to the addition of video reading tools and distortion correction to the StereoMorph package and produced the fantastic stereo video footage of free swimming Oceanic white tip sharks featured in this user guide. My sincere thanks to Caine and Mark for allowing me to include these videos here.

I would like to thank Dave Willard, Ben Marks, and Mary Hennen at the Field Museum of Natural History in Chicago, Illinois 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; DBI-1612230).