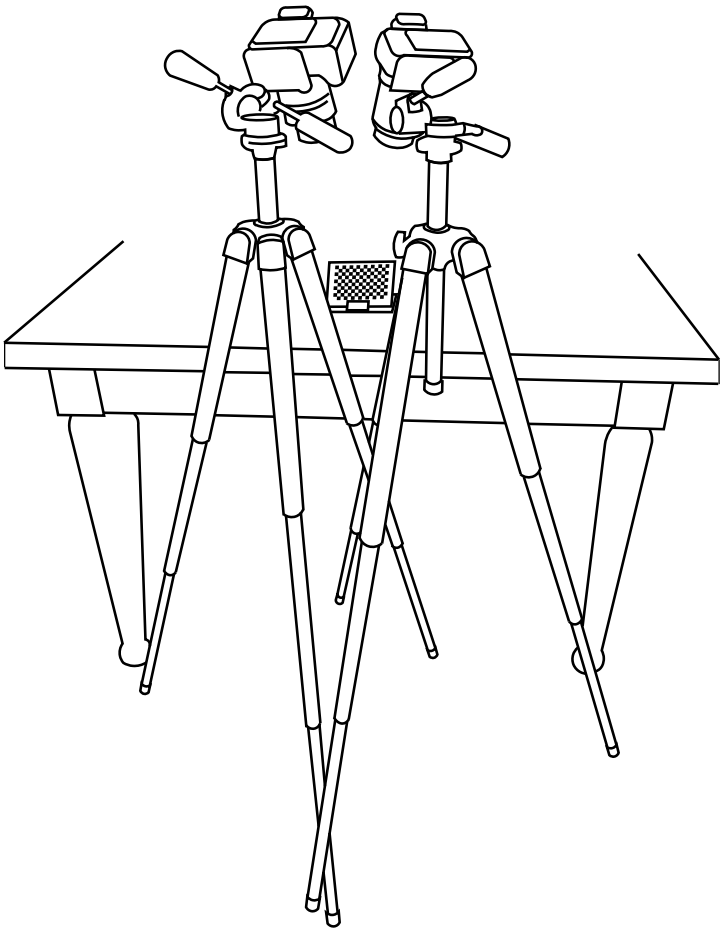
StereoMorph Tutorial

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



Aaron Olsen
September 9, 2014

Contents

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 (even DIY models), free open-source 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 time-consuming at one or more steps in the data collection process, making them better suited to the collection of high quality data from a relatively small number of specimens or objects.

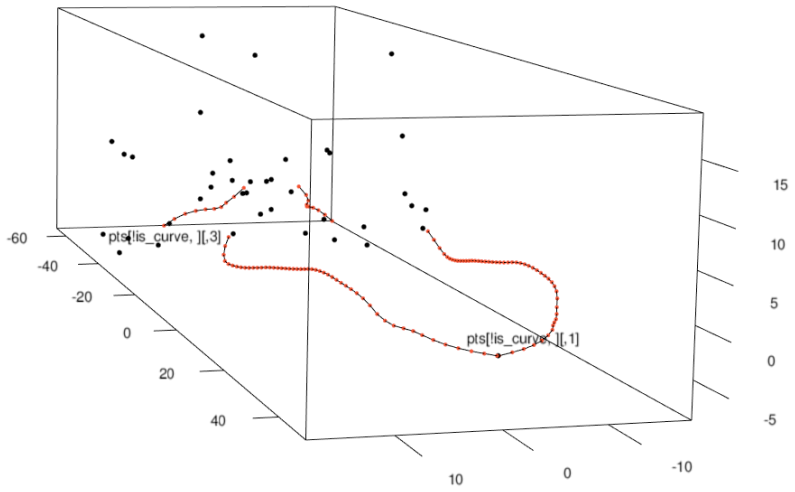
The StereoMorph method was designed specifically for cheap and rapid collection of landmarks and curves from a large number of specimens or objects. It's adapted from methods that have been used to study 3D animal motion for several decades (Wood & Marshall 1986; Socha, O'Dempsey & LaBarbera 2005; Hedrick 2008; Brainerd *et al.* 2010). All these studies rely on a method known as direct linear transformation (DLT) for the reconstruction of points from two or more camera views into 3D coordinates. Given two or more cameras positioned arbitrarily around some volume of space, DLT provides a mathematical basis for transforming a point's position in two or more camera views from 2D pixel coordinates into 3D real-word units (e.g. centimeters). StereoMorph uses DLT in the exact same way. Users photograph a specimen or object with two cameras, manually digitize points and curves they wish to collect in both views and these points and curves are reconstructed into 3D real-world coordinates.

Camera calibration, image digitization and point and curve reconstruction can all be performed using the R package StereoMorph. Additionally, StereoMorph is cheaper than most of the above methods: the method can be implemented for the cost of two digital cameras and two tripods (less than $1000 total). For users that already own a suitable camera, only the cost of an additional camera and tripod would be required. Since only two photos are required per specimen and all landmarks and curves are manually digitized directly on 2D images, the StereoMorph method is best suited to the collection of a few landmarks and curves from a relatively large number of specimens or objects. However, it is important to note that StereoMorph cannot produce 3D surface reconstructions nor be used to collect 3D surface data – only points and curves.

This tutorial will take users through all the steps required to go from points and curves digitized in a set of images

Landmarks and curves digitized on the same bird skull from two different cameras.

to a 3D reconstruction using the R package StereoMorph.



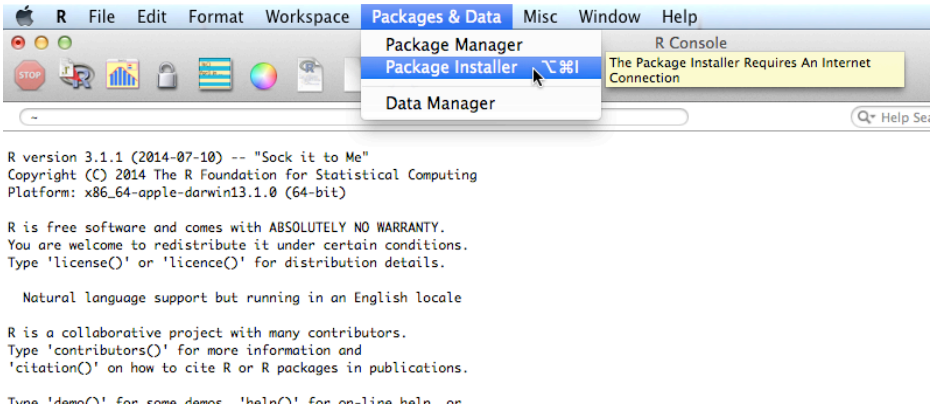
Landmarks (black) and curve points (red) reconstructed from 2D images using the StereoMorph R package. (Plotted using plot3d() from the 'rgl' R package)

Getting Started

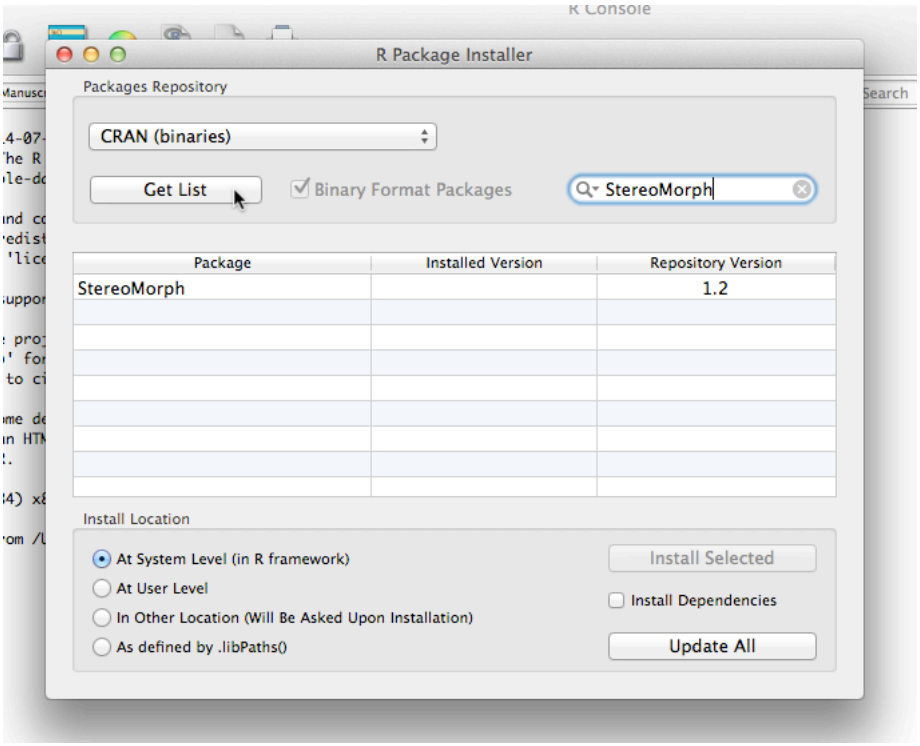
1. If you do not already have R installed on your computer, begin by installing R (see http://cran.r-project.org). R can be installed on Windows, Linux and Mac OS X.

2. Open R.

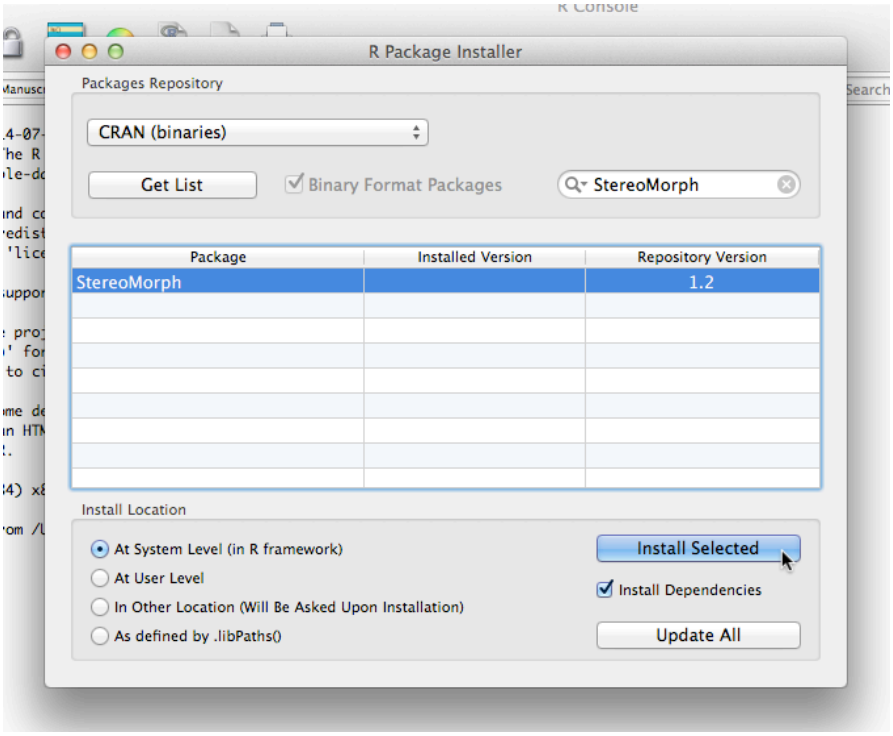
3. In R, go to *Packages & Data* > *Package Installer*.



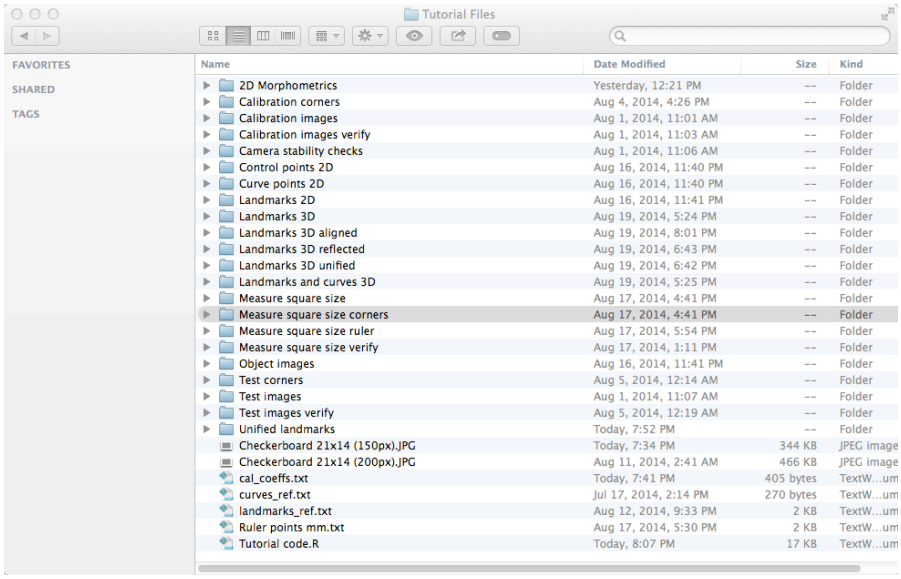
4. Find the StereoMorph package binary by typing "StereoMorph" into the *Package Search* box and clicking *Get List*.



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

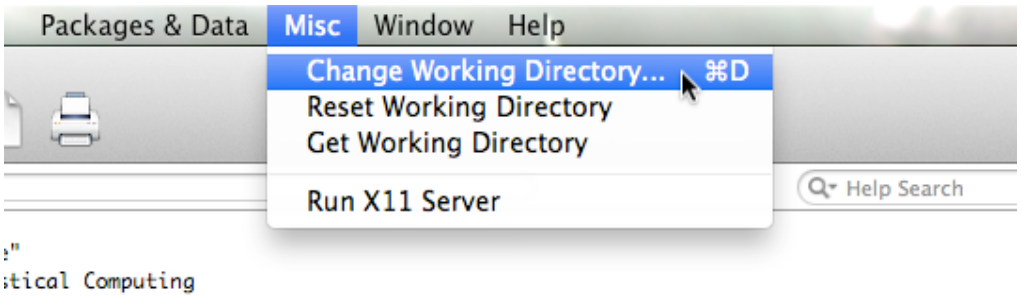


6. Download and unzip the StereoMorph Tutorial folder from http://github.com/aaronolsen/StereoMorph/blob/master/Tutorial%20Files.zip (~63 MB). Click on "view the full file" to download the zip folder. This folder contains all the files needed to perform the steps in this tutorial.

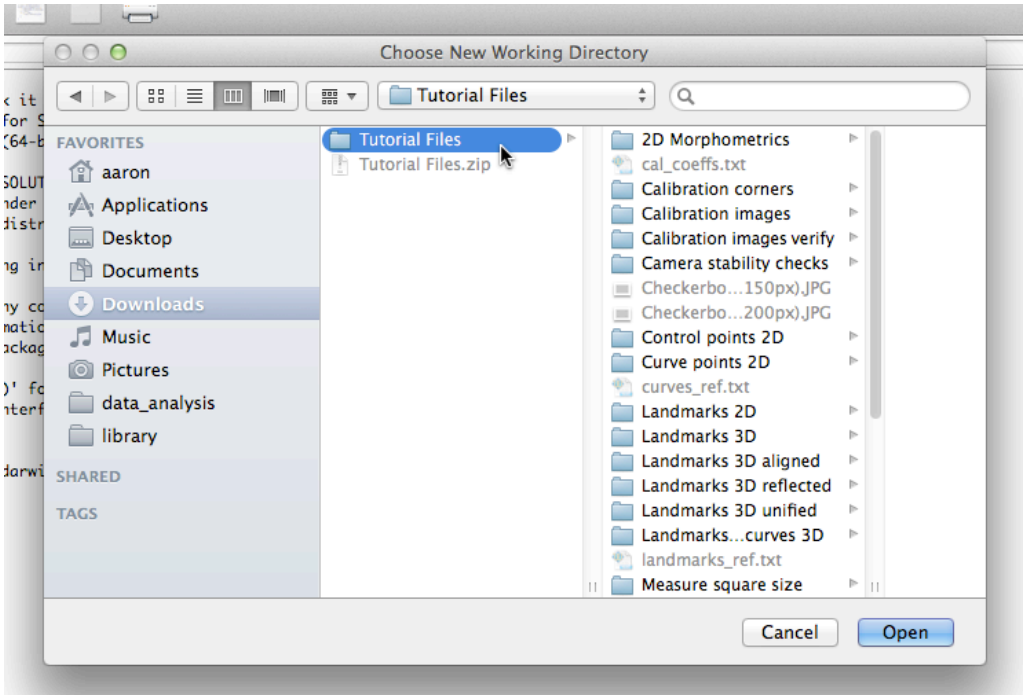


StereoMorph Tutorial files

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



8. Locate and select the unzipped StereoMorph Tutorial folder and click *Open*.



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

```
> library(StereoMorph)
```

For the rest of this tutorial, blue text preceded by a ">" in the style above will be used to indicate commands to be entered into the R console. All of the R commands executed in the tutorial can be found in the **Tutorial code.R** file in the Tutorial folder.

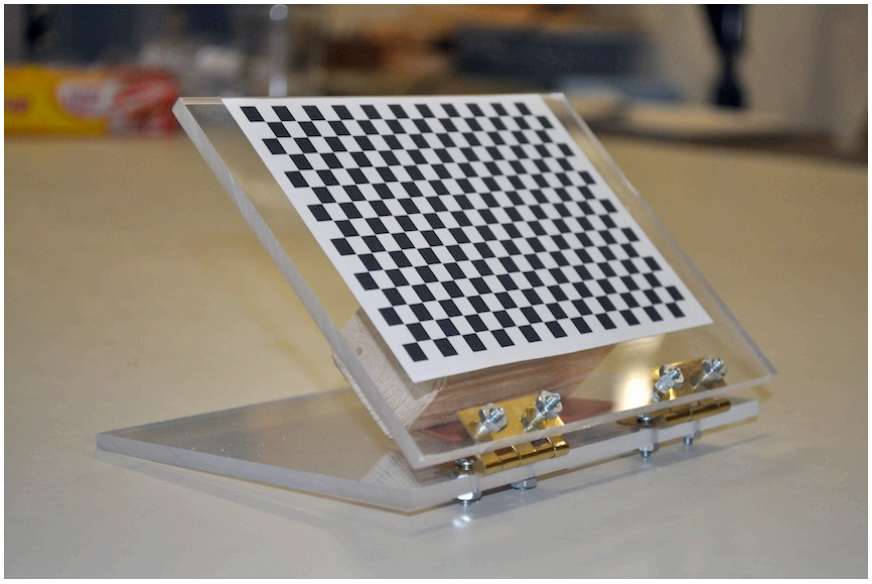
You are now ready to run all of the steps in the tutorial!

Creating a Checkerboard Pattern

This section demonstrates how to create a checkerboard pattern attached to a plate of plexiglass that can stand freely and be positioned at different angles. For users interested in a highly accurate calibration it's best to create two checkerboard patterns at slightly different scaling (discussed in more detail in the "Testing the calibration accuracy" section"). In this case, the relevant materials needed should be doubled.

Materials needed for this section:

- 2 pieces of 0.22" thick plexiglass cut to approximately the same size (slightly larger than the checkerboard pattern)
- 2 brass hinges (0.5"-1" wide)
- 8 Phillips head machine screws, with a diameter slightly smaller than the hinge holes, and centimeter-long thread
- 8 Nuts to match the machine screws
- Spray adhesive
- A power drill
- Drill bit slightly smaller than the hinge hole size



The final product of this section: a checkerboard pattern that can stand freely at different angles and in different positions for stereo camera calibration.

We'll take photographs of the checkerboard in different positions throughout the stereo camera space and use these to calibrate the cameras.

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

```
> library(StereoMorph)
```

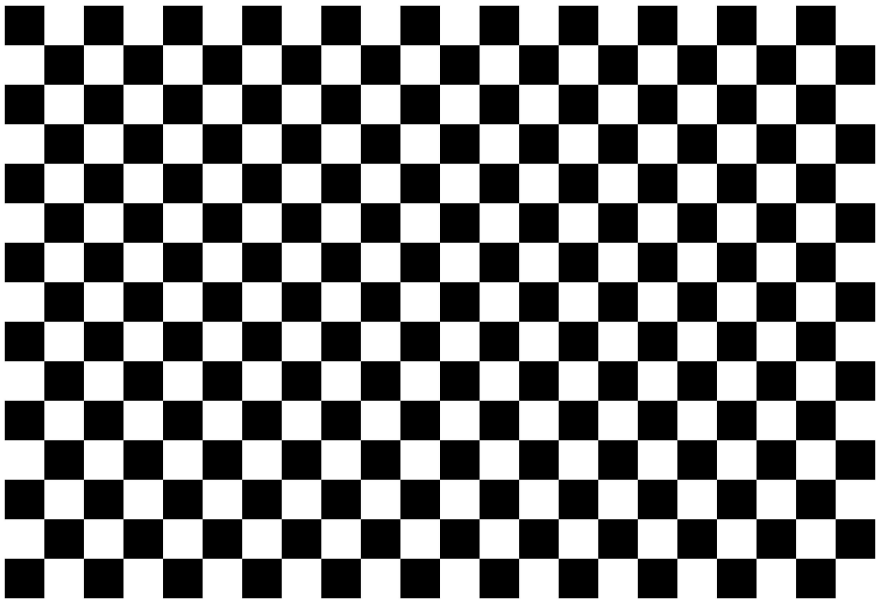
2. We'll create a checkerboard pattern by calling the function `drawCheckerboard()`. First, specify a filename for the checkerboard image. We'll create a checkerboard with 21 internal corners along one dimension and 14 along the other, with a square size of 200 pixels. It is best to create grid in which the number of squares along one dimension differs from the number along the other. This makes it easier to ensure that corners are detected in the same order for different orientations of the checkerboard.

```
> filename <- 'Checkerboard 21x14 (200px).JPG'
```

3. Call `drawCheckerboard()`.

```
> drawCheckerboard(nx=21, ny=14, square.size=200, filename)
```

The image will be saved to the current working directory - the StereoMorph Tutorial Folder (overwriting the current file with an identical version).



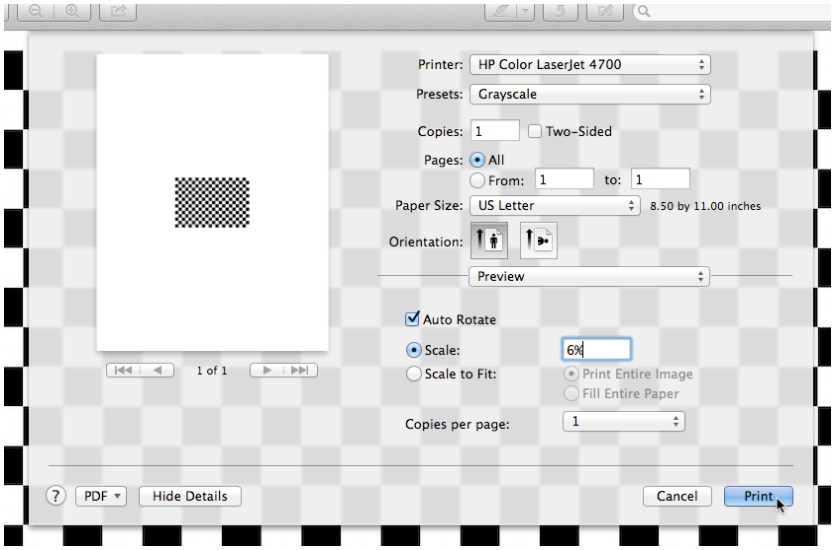
A 21x14 checkerboard pattern created by `drawCheckerboard()`.

Note that `nx` and `ny` are the numbers of internal corners along the horizontal and vertical axes, respectively, not the number of squares. We're using the internal corner count because when we auto-extract the corners, it's actually the internal corners that will be extracted. We want as many squares as possible but not too

many that we can't extract the corners when we photograph the grid. Because of potential imprecision in printing, more corners provide more sampling and may improve accuracy.

The `square.size` is the size of each square in pixels. We're creating a large image (4600 x 3200 pixels). When printing an image like this with straight edges, the printer will tend to blur the edges a bit when printed at full size. But if we scale the image down to less than 10%, the printed edges will be much sharper. The sharper edges might make a difference in how precisely the auto-extraction is able to find the corners.

4. Print the pattern onto a blank sheet of paper at 5-10% scaling. For this tutorial I printed two patterns: for the calibration I printed a checkerboard with 200 pixel squares scaled to 6% and to test the calibration accuracy I printed a checkerboard with 150 pixel squares scaled to 5%. Thicker paper (such as cardstock) works best because it is less likely to bubble over time once adhered to the plexiglass.



Printing a 21 x 14 checkerboard pattern, with 200 px squares at 6% scaling.

Any decent printer will do - for this tutorial I used an HP Deskjet F4200. The automated corner extraction uses model fitting to find each internal corner to sub-pixel resolution, enabling high accuracy even with a less precise printer.

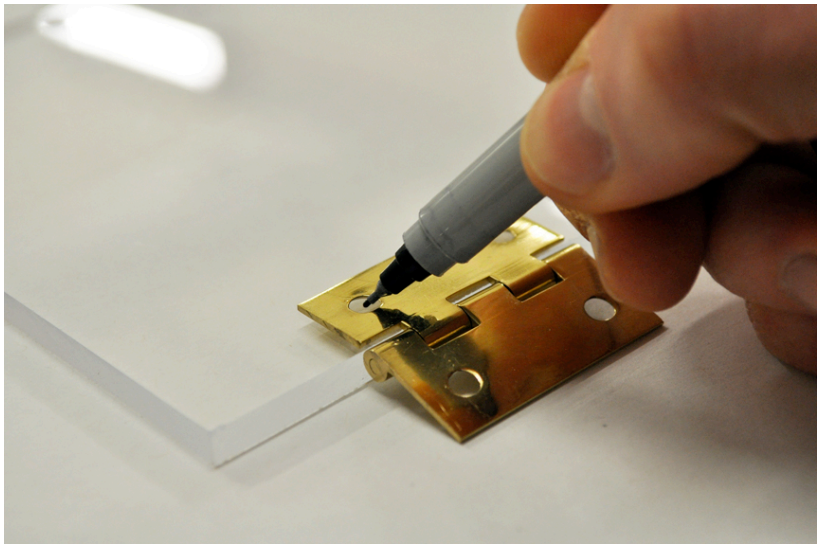
5. Write down the square size in pixels and the scaling at which the checkerboard was printed on the back of the checkerboard. This will be used in a later section to get the size of the checkerboard squares in real-world units.

6. Cut two pieces of 0.22" thick plexiglass to approximately the same size and slightly larger than the checkerboard pattern.

Two pieces of 0.22" plexiglass cut to approximately the same size.

You can cut the plexiglass by hand using the score and snap method or, if you have access to them, with a band saw or table saw. It's important to use 0.22" thick plexiglass; I've found that thinner plexiglass (such as 0.118" thick) warps easily. This warping isn't detectable to the naked eye but affects the calibration accuracy.

7. Drill holes in both plexiglass plates along the long edge where the hinges will attach. Simple brass hinges work fine. It's easiest to just lay the hinges on the plate and mark the center with a marker or pen. The hinges do not have to be perfectly square.

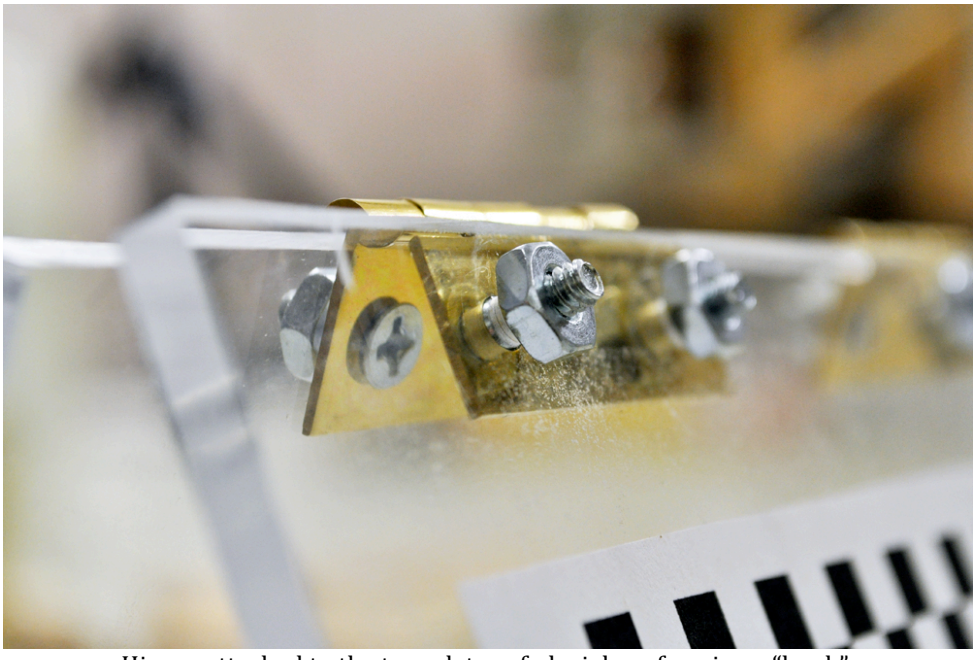


Marking drill holes for the hinges on the plexiglass.

8. Attach the hinges to each plate with small machine screws and nuts (the hinges will probably be sold with wood screws so you might have to buy the screws and nuts separately).

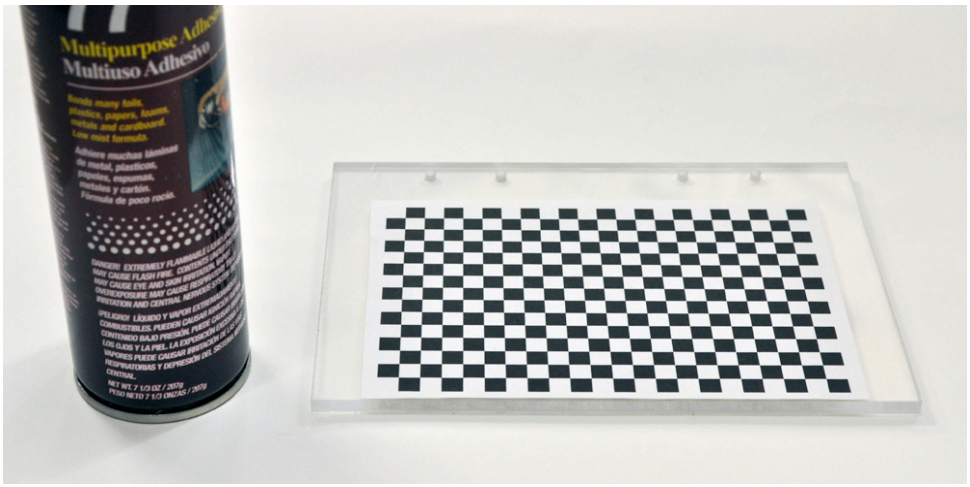
Phillips head machine screws and nuts used in this tutorial.

You'll need screws that are small enough in diameter to fit through the hole in the hinge and long enough to go through the hinge and plexiglass with some thread to attach the nut on the other side. About a centimeter of thread should be fine.



Hinges attached to the two plates of plexiglass, forming a "book".

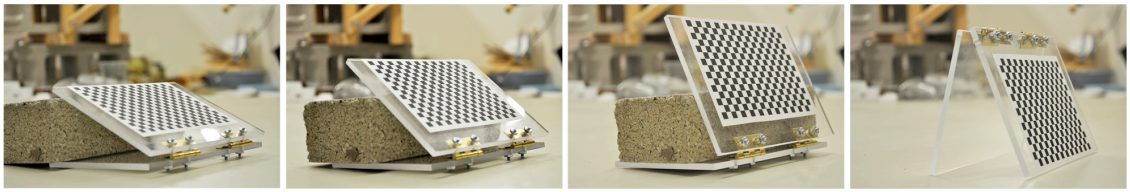
9. Cut out the pattern (leaving a bit of white around the edges) and attach the pattern to one of the plexiglass plates using spray adhesive.



Checkerboard pattern glued to the plexiglass plate.

The pattern does not have to be aligned perfectly - just as long as the entire pattern is on the plexiglass. Be sure that there are no folds or bubbles. You might want to practice once or twice before attaching it to the plexiglass since if there are bubbles or folds you might have to start over with a new piece of plexiglass.

You can use a heavy object or stand it up on its own to prop the grid at different angles. A rubber ball or bouncy ball can also work well if the surface has enough friction.

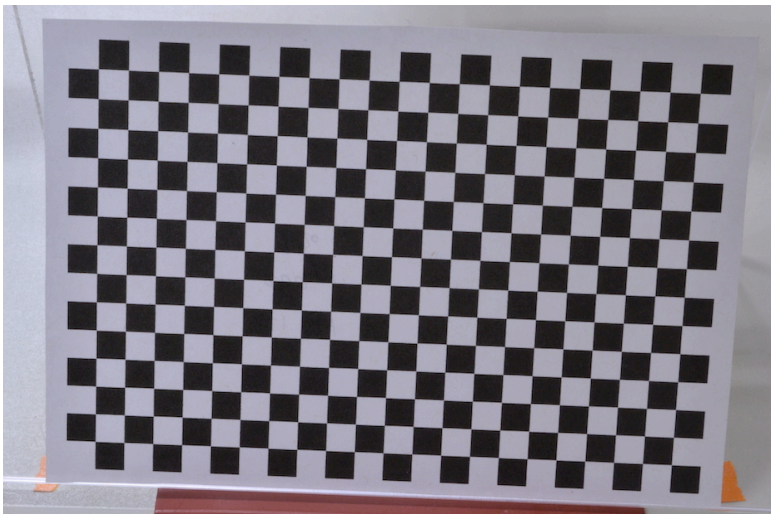


Checkerboard pattern set at different angles.

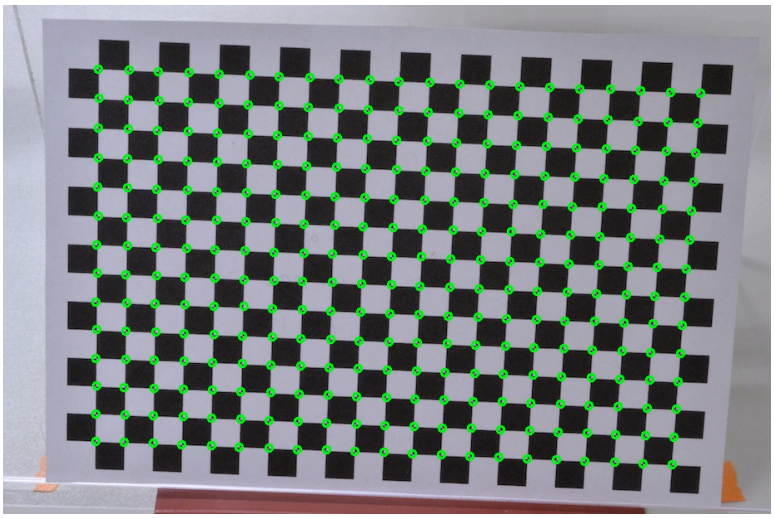
There are certainly many other ways of creating a stand-alone checkerboard pattern. Physically holding the grid in different positions is also possible. I use low lighting with long exposure times (since my specimens aren't moving I don't need bright lights) so if I physically held the checkerboards the images would be blurry. But with proper lighting or for calibrating a larger volume, physically holding a single plate of plexiglass or some other material is much easier.

Auto-detecting Checkerboard Corners

Given a photograph of a checkerboard pattern,



the `findCheckerboardCorners()` function in the StereoMorph package can automatically detect the internal corners of the checkerboard



and return these as a series of pixel coordinates to sub-pixel resolution (currently, this function only works with JPEG images).

```
          [,1]     [,2]
[1,]  575.1566 387.9233
[2,]  755.2105 395.1164
[3,]  935.8325 402.5703
[4,] 1115.8718 411.0474
...
```

Automated detection of the corners of a checkerboard pattern in a photograph will be used multiple times in this tutorial: to measure the size of checkerboard squares in real-world coordinates, to calibrate a set of cameras in stereo and to test the calibration accuracy. Automated corner detection can also be used to scale photographs for tasks such as 2D morphometrics, eliminating the need to manually digitize a series of points along a ruler.

1. Start by specifying the number of *internal* corners in the checkerboard. Note that the number of internal corners is not the number of squares but rather the number of corners where black squares adjoin one another. All the checkerboards used in this tutorial have 294 internal corners arranged in a 21 x 14 grid.

```
> nx <- 21
> ny <- 14
```

Which number you assign to `nx` versus `ny` is arbitrary, so long as you are consistent throughout.

2. Specify the location of the image to be read, the location where the corners should be saved and where to save a "verify image". The verify image is a copy of the input image with the corners drawn in so that you can check that the correct corners were found and determine in what order they were read (the `corner.file` and `verify.file` arguments are optional).

```
> image.file <- 'Calibration images/v1/DSC_0002.JPG'
> corner.file <- 'Calibration corners/v1/DSC_0002.txt'
> verify.file <- 'Calibration images verify/v1/DSC_0002.JPG'
```

3. Call `findCheckerboardCorners()`.

```
> corners <- findCheckerboardCorners(image.file=image.file, nx=nx,
  ny=ny, corner.file=corner.file, verify.file=verify.file)
```

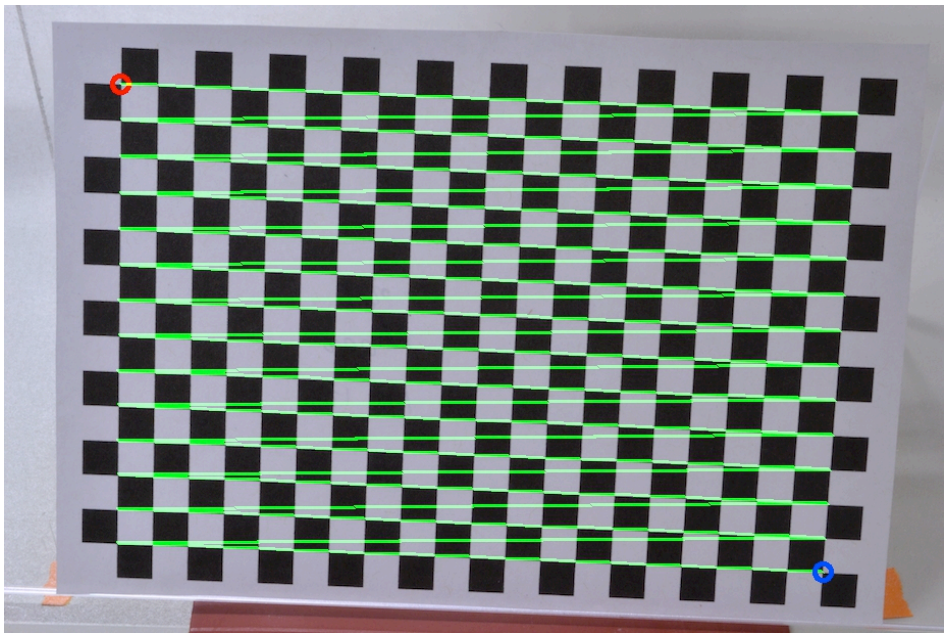
Since `print.progress` argument to `findCheckerboardCorners()` is `TRUE` by default, the progress of the function will be returned to the R console. Since corner detection can take several seconds per image, this allows users to track the progress of the function. Whether the expected number of internal corners were found will also be reported to the R console. You might have images for which `findCheckerboardCorners()` was unable to find the corners either because of lighting issues or because the checkerboard was at a very oblique orientation. If these are the calibration images, don't worry – the corners do not have to be found in every image pair for an accurate calibration (at least five images should provide a good calibration). The unsuccessful images will be ignored in subsequent steps. Note that the function assumes, by default, that the checkerboard squares in the image have a perimeter of at least 140 pixels (35 x 35 pixels).

The corners are returned and saved into the variable `corners` as a matrix with 294 rows and 2 columns. The corners are found to sub-pixel resolution by using information from a 23 x 23 square surrounding each corner to find a more exact corner point.

```
> corners
          [,1]     [,2]
[1,]  575.1566 387.9233
[2,]  755.2105 395.1164
[3,]  935.8325 402.5703
[4,] 1115.8718 411.0474
...
```

The order of the returned corners is important for calibrating cameras in stereo and for testing their accuracy. Between the two views and from photo to photo, the corners must be found in the same order. In this way, a corner in a particular row of the matrix will always correspond to the same corner on the checkerboard.

`findCheckerboardCorners()` will nearly always return the corners in the same order from photo to photo as long as the checkerboard is imaged in a similar orientation. The function looks for the top-left corner relative to the center of the checkerboard. The corners are then ordered first along the `nx` dimension and secondly along the `ny` dimension (this is why it is important that `nx` differs from `ny`). This can be verified by checking the verification image (example below) in which the first corner is indicated by a red circle, the last by a blue circle and all intermediate corners are connected by a green line (easy to remember as RGB).



The verification image indicating the found corners and the order in which they were found. The first corner is circled in red, the last in blue with intermediate points connected by green lines.

4. To find the checkerboard corners in multiple images, specify either a folder or a vector of files for `image.file`, `corner.file` and `verify.file`.

```
> image.file <- 'Calibration images/v1'
> corner.file <- 'Calibration corners/v1'
> verify.file <- 'Calibration images verify/v1'
```

The function will read all of the images in the folder(s) specified by `image.file` and the names of the images will be used to name the corner and verification images (with the proper file extensions). In this case, all files in the `image.file` folder(s) must be images.

5. Then call `findCheckerboardCorners()` just as before.

```
> corners <- findCheckerboardCorners(image.file=image.file, nx=nx,
  ny=ny, corner.file=corner.file, verify.file=verify.file)
```

Since 8 images are in the `image.file` folder, the function outputs an array of corners with the dimensions 294 x 2 x 8. The first matrix of corners can be obtained as follows.

```
> corners[, , 1]
          [,1]     [,2]
[1,]  575.1566 387.9233
[2,]  755.2105 395.1164
[3,]  935.8325 402.5703
[4,] 1115.8718 411.0474
...
```

In the next section we'll use `findCheckerboardCorners()` to measure the printed size of the squares in a checkerboard pattern.

Measuring Checkerboard Square Size

In order to use the checkerboard pattern we made in "Creating a Checkerboard Pattern" to calibrate cameras in stereo and make accurate measurements of objects in 3D, we need an accurate measure of the printed checkerboard square size in real-world units (e.g. millimeters). This can be accomplished in two ways: (1) calculate the printed size from the printing resolution (dots per inch, or DPI) and scaling and (2) photograph the checkerboard with a ruler and manually digitize points along the ruler to scale the checkerboard to real-world units. This section will demonstrate how to measure square size using both methods.

Measuring square size from DPI and scaling

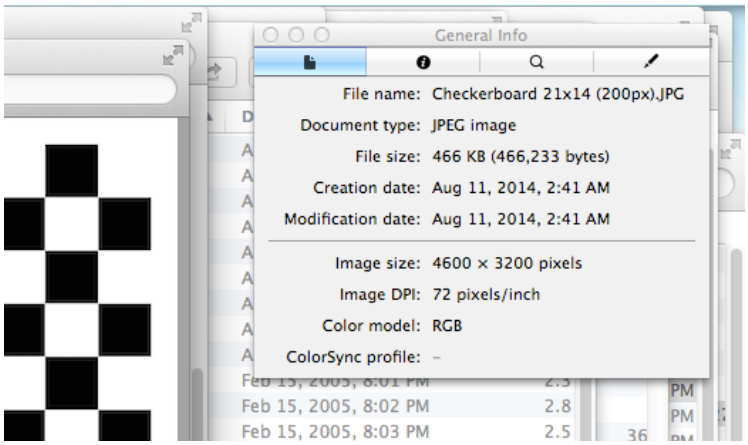
When a printer prints an image, it uses the size of the image in pixels, the DPI (dots per inch) setting for that particular image and the user-specified scaling to reproduce an image on paper. These three quantities can be used, with reasonable accuracy, to calculate the size of the printed image in real-world units.

$$\text{Image size in pixels} \times \frac{1}{\text{DPI}} \times \text{Scaling} \times \frac{25.4\text{ mm}}{1\text{ inch}} = \text{Image size in mm}$$

We can use the same formula to calculate the size of the squares in a checkerboard pattern by replacing image size with the square size in pixels.

1. Get the square size in pixels. This is the `square.size` parameter input to `drawCheckerboard()` when creating the checkerboard image (for the tutorial calibration pattern this was 200px).

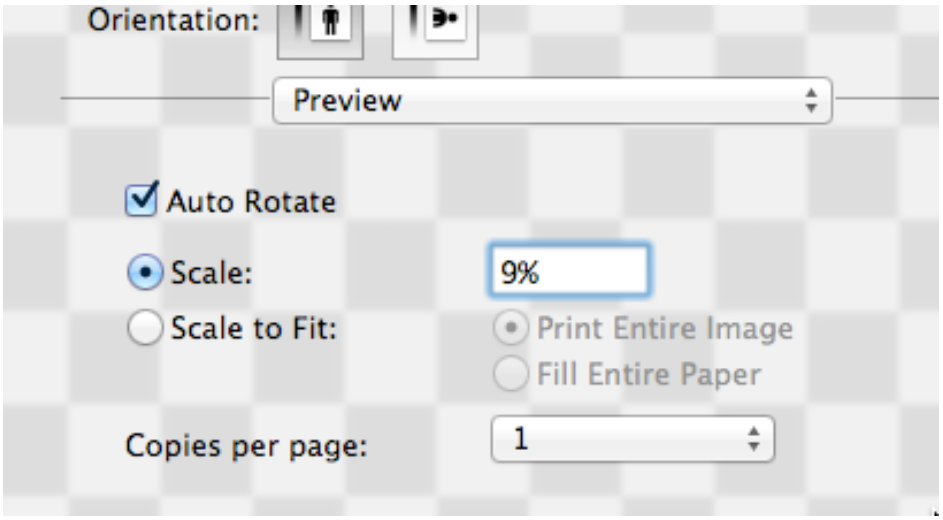
2. Find the image DPI. This can usually be found by looking at the properties of the image in the program you are using to print. For example, in Preview, the image DPI can be found in the Inspector (Tools > Show Inspector).



Looking up the image DPI in Preview.

Unless you have modified the DPI manually, the image should have a default DPI of 72 pixels/inch.

3. Get the scaling at which the checkerboard was printed. The scaling is set in the print dialog box for the program you are using to print the image. For the tutorial calibration pattern this was 9%.



Setting the print scaling in the Print dialog box.

4. Enter these values into the formula to calculate the square size in millimeters.

$$200\ \text{px} \quad \text{x} \quad \frac{1}{72\ \text{px/in}} \quad \text{x} \quad 0.09 \quad \text{x} \quad \frac{25.4\ \text{mm}}{1\ \text{in}} \quad = \quad 6.35\ \text{mm}$$

A 200 pixels-per-square checkerboard, scaled to 6% was used in this tutorial to test the calibration accuracy. We can also calculate the size of these squares in real-world units.

$$200\ \text{px} \quad \text{x} \quad \frac{1}{72\ \text{px/in}} \quad \text{x} \quad 0.06 \quad \text{x} \quad \frac{25.4\ \text{mm}}{1\ \text{in}} \quad = \quad 4.233\ \text{mm}$$

I've found that, for some printers, these formulas provide an accurate measure of the printed square size. I used two different printers for this tutorial: an HP Color LaserJet 4700dn to print the 9% checkerboard and an HP Deskjet F4200 to print the 6% checkerboard. Measuring the square size of the 6% checkerboard using a ruler (see the following section) returned nearly the exact same value as the formula above (4.2327 mm). However, ruler measurements for the 9% checkerboard yield a square size of approximately 6.365 mm, 0.015 mm greater than is predicted from the formula above. The actual square size is likely 6.365 mm since this provides the lowest error in combination with a test calibration square size of 4.233 mm. My best guess is that the accuracy of the above formulas will vary from printer to printer, emphasizing the importance of confirming the square size using a ruler as detailed in the following section.

Measuring square size using a ruler

For users who want to verify that their printer is producing checkerboards at the expected size, this section will demonstrate how to measure the checkerboard square size using a ruler.

Materials needed for this section:

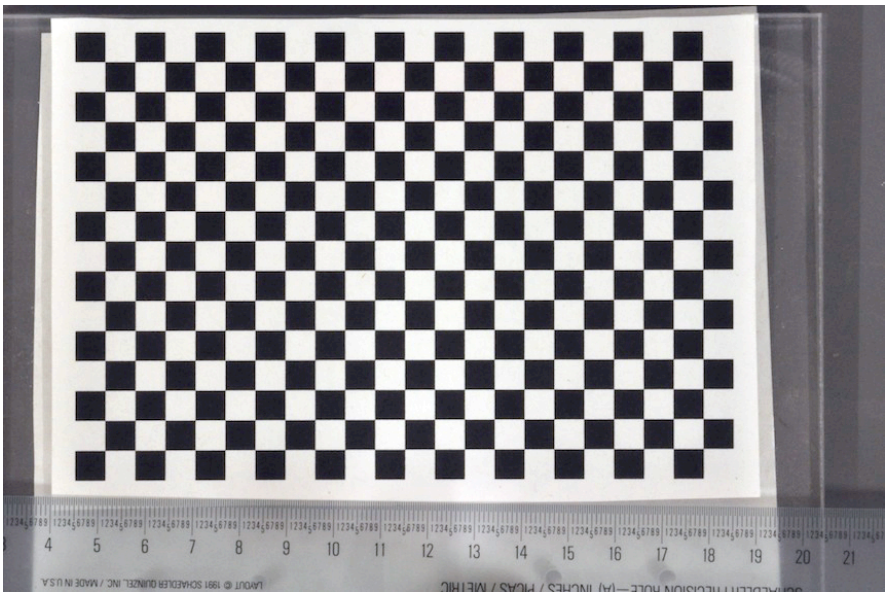
- Ruler (required precision will depend on the application)
- A printed checkerboard pattern

1. Locate a good ruler. For this tutorial and for my work, I use a 12" Single "A" - #46-IM precision rule from Schaedler (www.schaedlerprecision.com; approximately $30, including tax and shipping).

12" Single "A" - #46-IM precision rule from Schaedler.

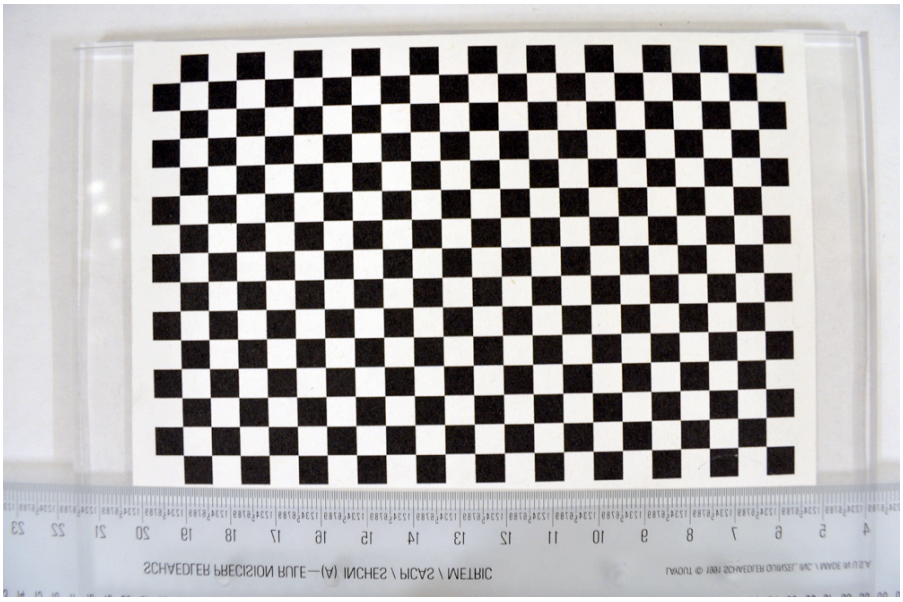
These rules have an accuracy tolerance of better than 0.00024". I haven't tried a calibration with a standard office ruler or the rulers that come with dissection kits. Either might work just as well, but if you're concerned about accuracy a precision rule might be a worthwhile investment.

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



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

It is essential that your camera lens does not have any significant lens distortion. For the above image, I used a Nikon D5000, which came with an 18-55 mm zoom lens, a standard lens often sold with Nikons. If you use the 18-55 mm lens at 18 mm (completely zoomed out), you get barrel distortion.



Checkerboard and ruler photographed at 18 mm focal length.
Notice the curvature of the ruler, which should be straight.
Nikon D5000 with AF-S DX Nikkor 18-55mm lens at 18mm, f/36.

At 55 mm, distortion is insignificant; in this tutorial all stereo camera images were taken with an AF-S DX Nikkor 18-55 mm lens at 55 mm.

Also be sure that the entire checkerboard pattern falls within the image. The checkerboard should be positioned approximately coplanar with the image plane or the end of the camera lens (i.e. the shaft of the camera 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.

3. Upload the photograph of the checkerboard and ruler to your computer. The StereoMorph Tutorial folder 'Measure square size' has photographs of the two checkerboards used in this tutorial with a ruler. For demonstration, we'll use the checkerboard scaled to 6%.

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

```
> library(StereoMorph)
```

5. Specify the file path(s) of the image(s) or a folder containing the image(s).

```
> image.file <- 'Measure square size/Checkerboard 6p.JPG'
```

6. Specify where to save the corners and verification image(s).

```
> corner.file <- 'Measure square size corners/Checkerboard 6p.txt'
> verify.file <- 'Measure square size verify/Checkerboard 6p.JPG'
```

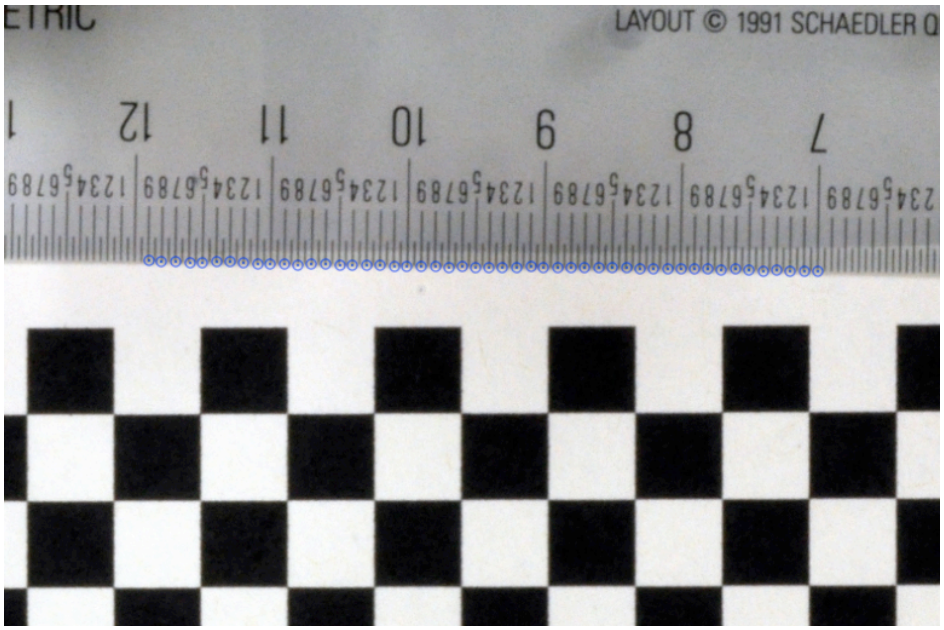
7. Specify the number of internal corners in the checkerboard.

```
> nx <- 21
> ny <- 14
```

8. Call findCheckerboardCorners() to find the internal corners in the image.

```
> corners <- findCheckerboardCorners(image.file=image.file, nx=nx,
  ny=ny, corner.file=corner.file, verify.file=verify.file)
```

To measure the square size in real-world units, we need to manually digitize several points at equal intervals along the ruler.



50 points digitized along a precision rule at 1 mm spacing (zoomed in).

This can be done easily with StereoMorph's digitizing application. We'll use the basic features of the digitizing application to do this – a more thorough introduction to the digitizing app will come in a later section ("Digitizing Photographs").

9. Specify where to save the manually digitized points ("landmarks") along the ruler. We'll start with an empty (and not currently existing) file.

```
> landmarks.file <- 'Measure square size ruler/Checkerboard 6p t.txt'
```

10. Specify a text file containing a list of the points to be collected. In this case, this is simply a list of names for an ordered point set. The file 'Ruler points mm.txt' in the StereoMorph Tutorial Folder contains a list from 'mm_pt001' to 'mm_pt200'. You can collect less than 200 points or you can add additional names to collect more.

```
> landmarks.ref <- 'Ruler points mm.txt'
```

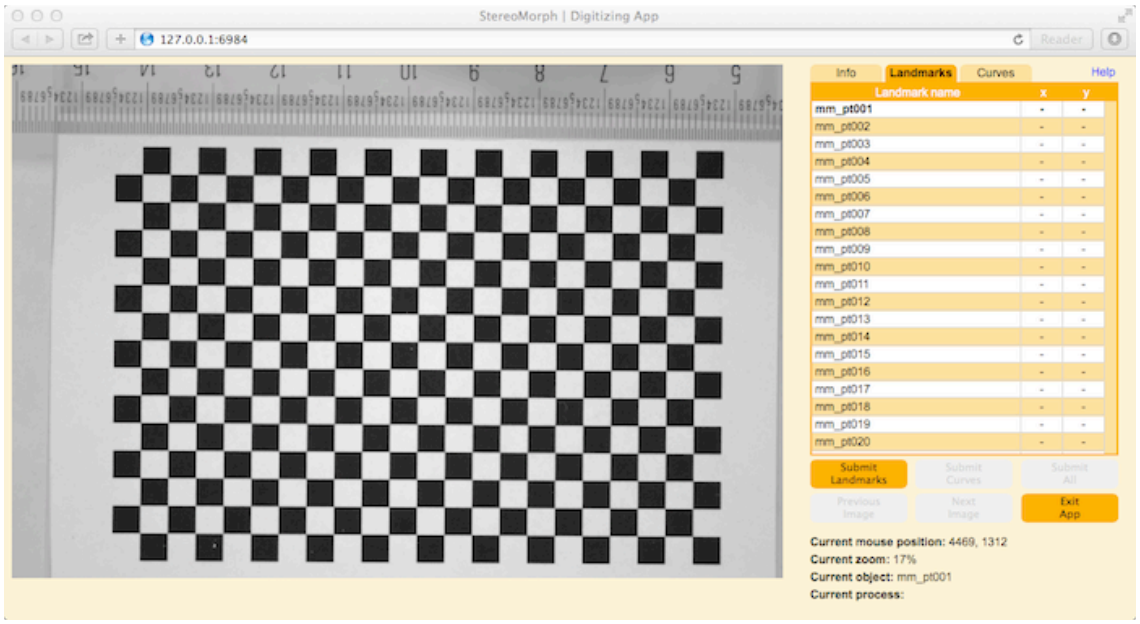
This input parameter can also be a vector.

```
> landmarks.ref <- paste0('mm_pt', formatC(1:200, NULL, 3, "d", "0"))
```

11. Call `digitizeImage()` to launch the StereoMorph digitizing app.

```
> digitizeImage(image.file=image.file, landmarks.file=landmarks.file,
  landmarks.ref=landmarks.ref)
```

The app should launch in your default web browser.



The StereoMorph digitizing app with the checkerboard and ruler.

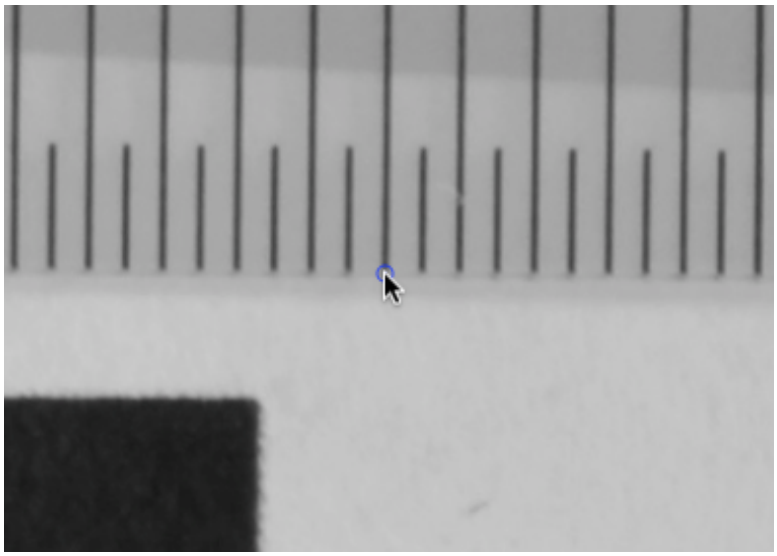
The left portion of the app is the image frame. This is where you can navigate around the image and add the ruler points using your mouse or trackpad. The right two-thirds is the control panel, where you can view and save the points you've digitized.

12. Click on the 'Landmarks' tab in the control panel, if this is not already selected. You'll see a list of the ruler point names with two columns ('x' and 'y') where the digitized points will be added. The first point should be selected (indicated by bold) by default.

13. Position your cursor somewhere over the image near the top-right corner and scroll either up or down. By scrolling either up or down (the direction will depend on your customized system settings) you can zoom in or out of the image (similar to Google Maps). To more quickly navigate around the image, the zoom tracks the position of your cursor and zooms in and out of particular region of the image based on the current position of your cursor. So, with the cursor near the top right, the size of the image increased while simultaneously positioning the top-right corner of the image closer to the center of the image frame.

14. Click anywhere on the image (single click) and drag. This causes the image to move with your cursor. By scrolling and clicking and dragging you can navigate around the image.

15. Point your cursor at the end of one of the ruler lines just outside the edge of the checkerboard, *opposite* the numbers (e.g. at the 5 cm line) and double-click or press the 'x' on the keyboard.



Double-click or press 'x' to digitize a point at place indicated by the cursor.

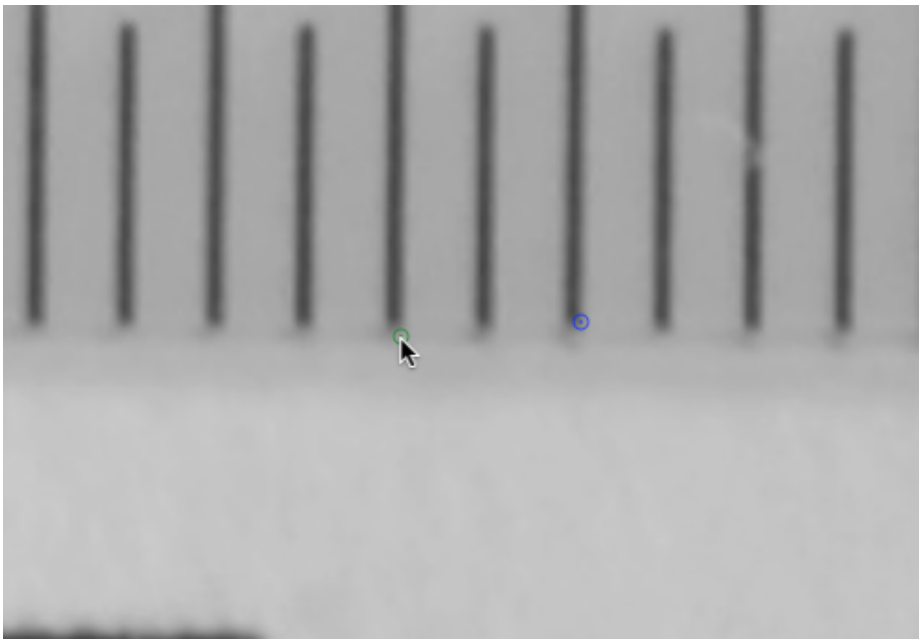
This will cause the selected point (here, 'mm_pt001') to be added at the current location of the cursor. By default the app automatically advances to the next point (here, 'mm_pt002'). This can be turned on and off by pressing shift+a but we'll keep it on for now because it makes collecting several sequential points faster. The first ruler point was probably not positioned exactly where we want it, so we'll re-position it.

17. Press 'p' on the keyboard to select the previous point ('mm_pt001') or select it by clicking anywhere in its row in the landmark tab. The point will change from blue to green.

16. Use the arrows on the keyboard to move the point up, down, left and right at single pixel increments. To be as precise as possible, it's important to position the points consistently along the ruler and the bottom left or right edge of the line is easy to use as a consistent landmark.

17. Once the point is re-positioned to the bottom edge of the ruler line, press 'n' to advance to the next point.

18. Position the cursor at the next *millimeter* tick mark (those with numbers above them) and double-click or press 'x' to add the second point.



Positioning a second ruler point.

This might also need some repositioning.

19. Place the cursor over the second ruler point and either double click or press 'x'. This is a third way in which to select a point.

20. Use the arrow keys to re-position the point more precisely.

If you were digitizing an image with your own checkerboard, you would repeat these steps across the image, collecting around 100 points. Several points will provide a large sample of measurements, essential for achieving high accuracy. Additionally, since it is only possible to manually digitize a point to pixel resolution, digitizing several points provides a means of measuring the distance to sub-pixel resolution.

Digitized points can be deleted by pressing the 'd' key. Any points in the landmark table with coordinates (–, –) will be ignored when saving.

21. Click the 'Submit Landmarks' button to save the current landmarks. If you open this file (e.g. 'Checkerboard 6p.txt' in the 'Measure square size ruler' folder), you'll see each digitized point as a separate row in a matrix.

```
mm_pt001  4317 451
mm_pt002  4278 450
```

If you need to go back and edit these points or add new points, specify the same file as the file to save landmarks to and these points will be loaded into the app.

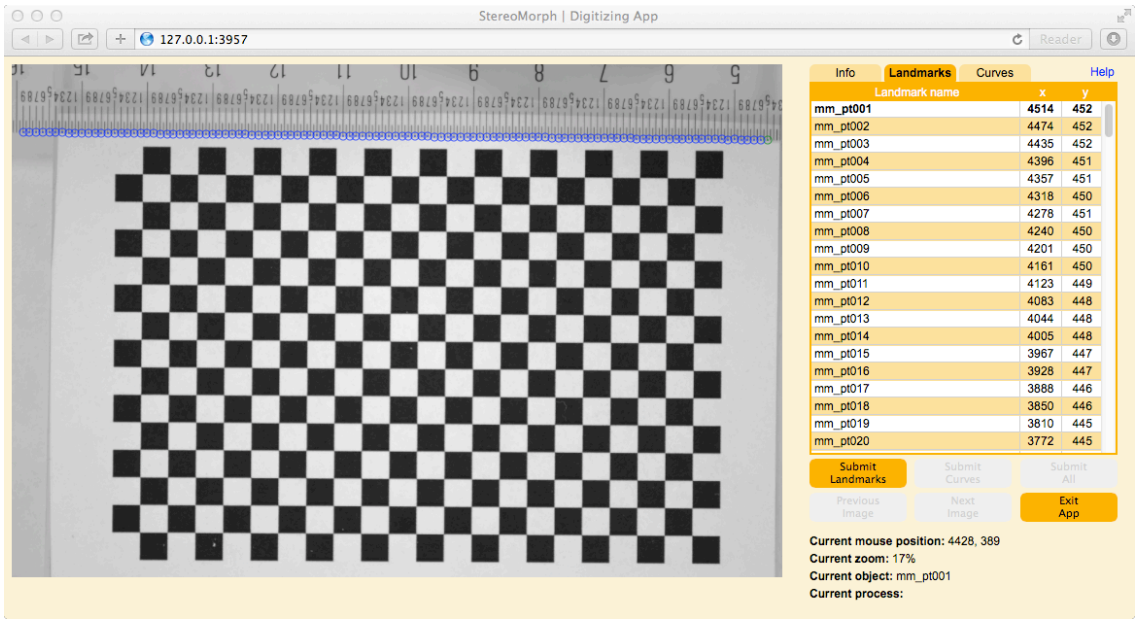
22. Click the 'Exit App' button to close the app and return to the R console.

23. We'll load in a file of already digitized ruler points for this image. Change the save-as file path for the ruler points to the 'Checkerboard 6p.txt' file in the 'Measure square size ruler' folder of the tutorial folder.

```
> landmarks.file <- 'Measure square size ruler/Checkerboard 6p.txt'
```

24. Then, re-launch the app.

```
> digitizeImage(image.file=image.file, landmarks.file=landmarks.file,
  landmarks.ref=landmarks.ref)
```



Points digitized manually along the ruler used to measure the checkerboard square size in real-world units.

The 115 digitized ruler points at 1mm spacing in 'Checkerboard 6p.txt' will be loaded into the image frame. These are the points we'll use to measure the square size.

25. Click 'Exit App' to close the app and return to the R console.

26. Before measuring the square size, specify the size of the interval we've digitized on the ruler in real-world coordinates.

```
> ruler.pt.size <- '1 mm'
```

27. Call `measureCheckerboardSize()`.

```
> measure <- measureCheckerboardSize(corner.file=corner.file, nx=nx,
  ruler.file=landmarks.file, ruler.pt.size=ruler.pt.size)
```

28. Lastly, call `summary()` on the output.

```
> summary(measure)
measureCheckerboardSize Summary
   Checks for planarity of checkerboard
      Mean length of opposing sides: 3301.76 px x 2138.61 px
      Differences in lengths of opposing sides: 1.42 px x -3.1 px
   Simple checkerboard model fit
      Square size in pixels: 165.0367 px
      Mean distance of points from model: 0.64 px +/- 0.5
   Ruler points model fit
      Distance between points on ruler: 38.99043 px
      Mean distance of points from model: 0.73 px +/- 0.58
   Real-world units
      Square size in real-world units: 4.232748 mm
      Real-world units per pixel: 0.02564732 mm
```

This reports the output of several different operations performed by `measureCheckerboardSize()`. The function first tests the planarity of the checkerboard relative to the image plane. If the checkerboard is tilted in any direction, opposing sides will differ substantially in their length. The differences in lengths of opposing sides shouldn't differ by more than a few pixels. `measureCheckerboardSize()` can be called without the ruler parameters to check the corners for planarity before digitizing any ruler points.

Rather than simply finding the distance between corner points, the function fits a transformed grid model to all of the points. This model accounts for the perspective effects of grid positioned in 3D space and imaged on a 2D plane. Fitting a model to the points ensures greater robustness in the side measurements, taking into account the position of all the internal corners rather than simply the four corners.

The function then fits a simple checkerboard model to the points, using the minimal number of parameters necessary to define a 2D square grid. The model goodness-of-fit is reported as the mean distance of the input points from the model-generated points. This should be less than a pixel. For the 6% percent checkerboard above the input points differ from the model-generated points by 0.64 pixels on average, indicating that the corners extracted from the photograph very closely resemble a perfectly planar grid. The checkerboard square size is then calculated from the model fit parameters. As opposed to simply averaging the distance between consecutive points (which leads to a biased estimate), the model

fitting provides a robust, best-fit solution to the square size. Here, the best-fit estimate of the square size is approximately 165 pixels.

Lastly, the function measures the ruler interval in pixels, again using a model fitting procedure to find a best-fit solution. Only in this instance, the model is a line with points at equal intervals. Here, the best-fit distance between ruler points is 38.99 pixels. The goodness-of-fit is reported as the mean distance of the input ruler points from the model, as with the internal corners.

The square size in real-world units can then be calculated as the square size in pixels multiplied by the input parameter `ruler.pt.size`, divided by the ruler interval in pixels.

$$\text{Square size in pixels} \times \frac{\texttt{ruler.pt.size}}{\text{Ruler interval in pixels}} = \text{Square size in } \texttt{ruler.pt.size} \text{ units}$$

The square size in the same units as `ruler.pt.size` is reported on the second to last line of the summary (and returned as `measure$square.size.rwu`) along with the size of each pixel in the photograph in real-world units. For this image, each pixel in the plane of the checkerboard is about 26 x 26 microns.

By digitizing many points along the ruler and hundreds of corner points at subpixel resolution, we're able to measure the square size with an accuracy exceeding even the pixel resolution of the image. There is a certain amount of error in measuring the square size using a ruler. I've found that measurements made with a ruler can differ among repeated tests by up to 10 microns. Note that for this printer, measurements using a ruler yield a square size, 4.232748 mm, nearly identical to that calculated from the printer resolution and scaling, 4.23333 mm; this may not always be the case. Repeating these steps for the 9% scaled checkerboard (printed using a different printer), I found the square size to be 6.365 mm, 0.015 mm greater than is predicted from calculations simply using DPI and scaling. In general, measuring the square size using a ruler provides the highest accuracy and these measurements can be confirmed by calibrating a set of cameras using a checkerboard of one size and testing the accuracy with a checkerboard of a different size.

Arranging the Cameras

This section covers key aspects on how to arrange cameras in a stereo setup.

Materials needed for this section:

- 2 cameras (preferably DSLR cameras with minimal distortion lenses)
- 2 camera remotes
- 2 sturdy tripods
- Masking or colored tape

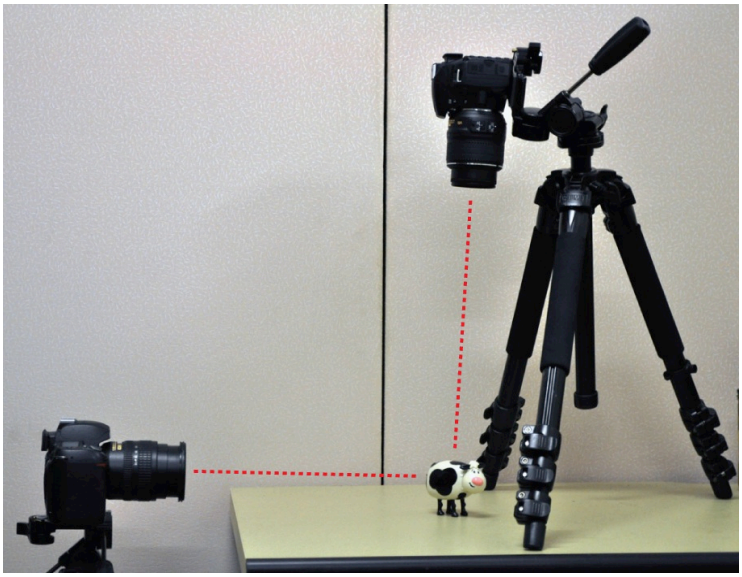
General considerations

Whether you're collecting landmarks or curves, the number of cameras you include, the lenses that you use and how you position the cameras depends on what you want to collect landmark and curve data from.

Here are some general principles:

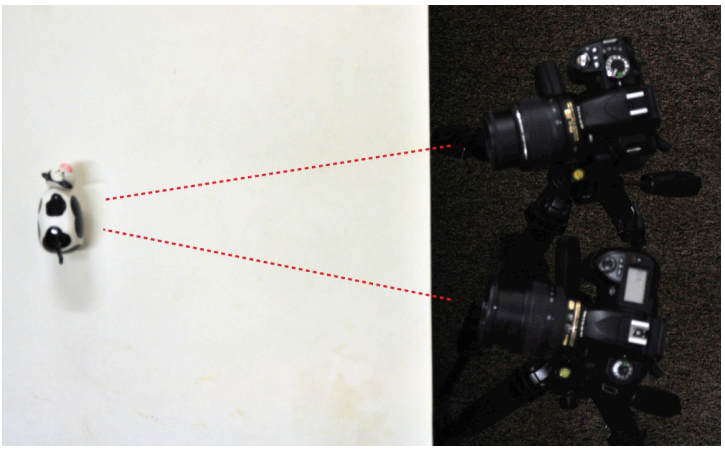
1. The views among the cameras must overlap. Since 3D reconstruction requires the pixel coordinates of a point in at least two cameras, the point must be visible in at least two camera views.

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 30 degrees



Two-camera stereo setup with the cameras at about 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. 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.

3. The volume of space visible in both cameras should be large enough to contain the object(s) and landmarks or curves you'll be digitizing. You might have to flip the object around a couple of times (to get the opposite side, for instance). If not all of the object is visible, you'll have to digitize sections of the object separately and then assemble the point sets based on overlapping points, which requires more time digitizing.

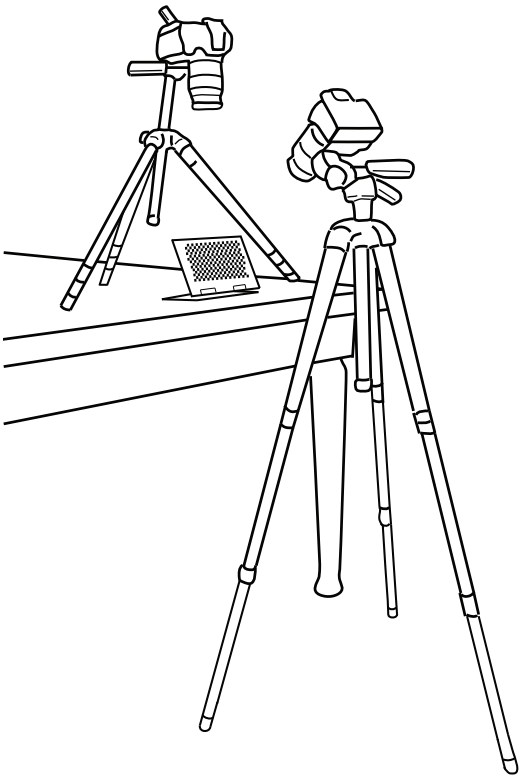
4. The cameras must not move for all of data collection and calibration. The cameras can be calibrated before or after data collection but throughout and between these steps the cameras must remain in the exact same position. Because the camera is often positioned half a meter or more away from the object, a sub-millimeter shift of the camera can translate into a large shift in the image frame, causing rather large inaccuracies.

5. The focal length (zoom) and focus of the lens must not change for all of data collection and calibration. The calibration is specific to a particular

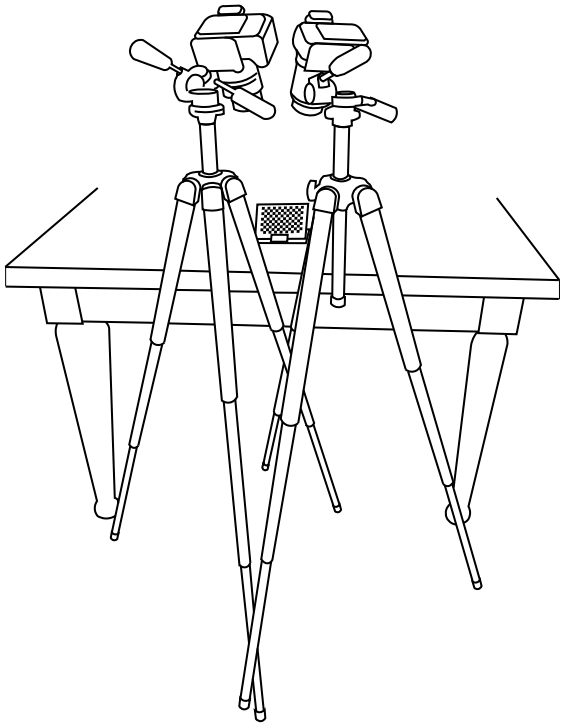
focal length and focus; thus, the cameras will have to be re-calibrated if either of these changes.

Given all of these caveats, the best course of action is to try out several different camera setups and test the accuracy fully before collecting any data (testing the calibration accuracy is detailed in the "Testing the Calibration Accuracy" section). Although your own setup might differ from the camera setup used in this tutorial, it will at least provide an example of one possible setup and how to accommodate the considerations above.

For this tutorial, the cameras were arranged as shown below on the left, using a tabletop to position the calibration grid and objects being photographed. The tabletop can also be used to position lights around the object, if desired.

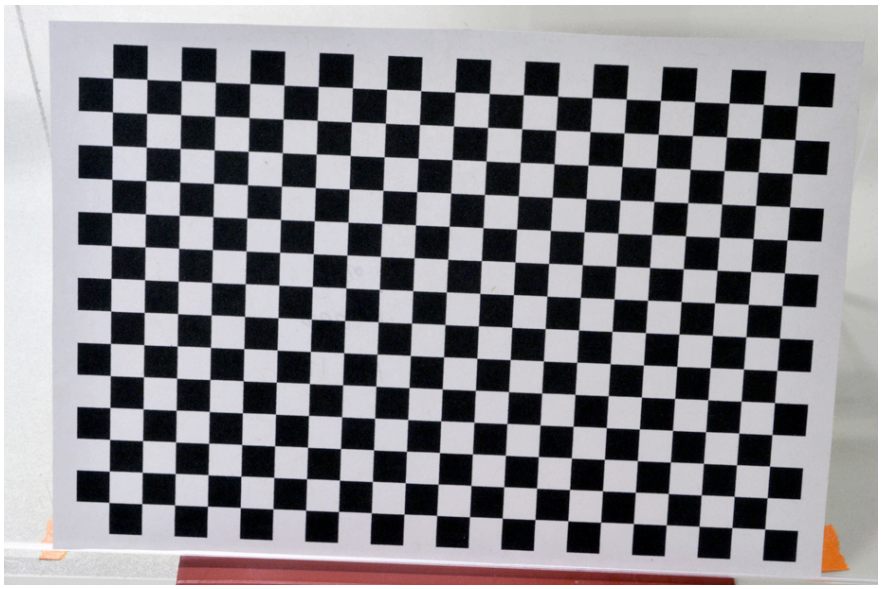


The two-camera stereo arrangement used in this tutorial.



A second possible two-camera arrangement providing more convergent views.

In this tutorial, all stereo camera images were taken with Nikon cameras, fitted with AF-S DX Nikkor 18-55 mm lenses at a focal length of 55 mm. At a focal length of 55 mm, distortion is nearly undetectable. It is essential that the lenses have minimal distortion. About 50 cm away from the tip of the lens, the checkerboard pattern almost nearly fills the image frame. This means that during the calibration step, we can fully sample the calibration volume with fewer calibration images.



A sample calibration image from one camera in a stereo camera setup.

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.

A small piece of rubber under a tripod leg can keep it from slipping on a smooth surface.

1. Before calibrating or starting to collect data, attach small pieces of tape to a surface in the calibration space.

Add tape to a fixed surface in the camera view to both box the calibration space and test whether the cameras move during data collection or calibration.

This serves both to remind you where the calibrated volume is when positioning objects and it also to test whether the cameras have shifted during data collection.

2. Take photos of the tape frame before beginning and after having taken all of the photos. If the images are identical then the cameras have not shifted significantly.

3. Make sure that all connections/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.

It's best to use a remote (wireless or cord depending on the Nikon model) 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.

4. If your lens has vibration reduction (VR) or automatic focus, be sure to turn both of these off. Vibration reduction uses a small gyroscope in the lens to compensate for camera motion and thus 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.

5. 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.

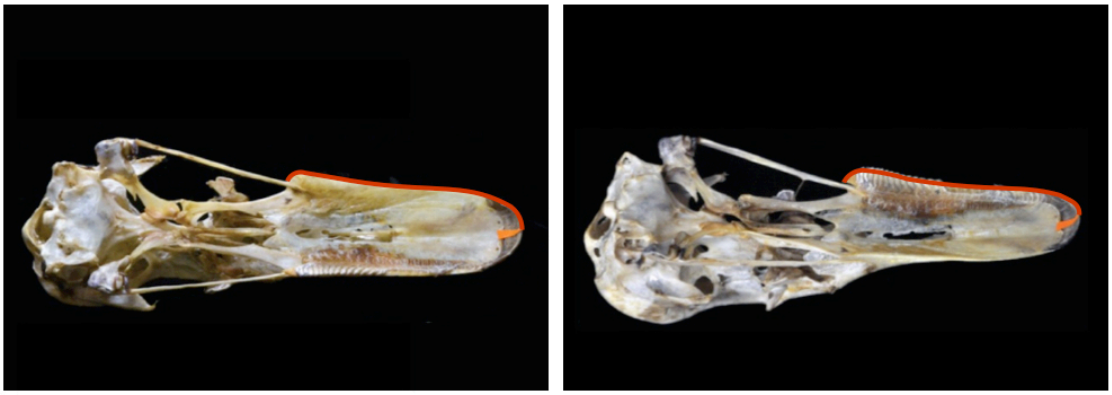
The smaller the aperture, the greater the depth of field (i.e. the more things are in focus both close and far away from the camera). This is essential in a stereo camera setup because in order to digitize points accurately throughout the calibration volume they must be in focus.

In the next step, we'll photograph the checkerboard we made in step 1 in different positions and orientations within the stereo camera setup and use these images to calibrate the cameras.

Arranging cameras for curve reconstruction

Collecting curve data using a stereo camera setup requires an extra consideration when arranging the cameras. Landmarks digitized in two different camera views are reconstructed in StereoMorph under the assumption that the same point in 3D space is digitized in both views. In principle, curves can be reconstructed in the same way, by breaking the curve down into a series of landmarks.

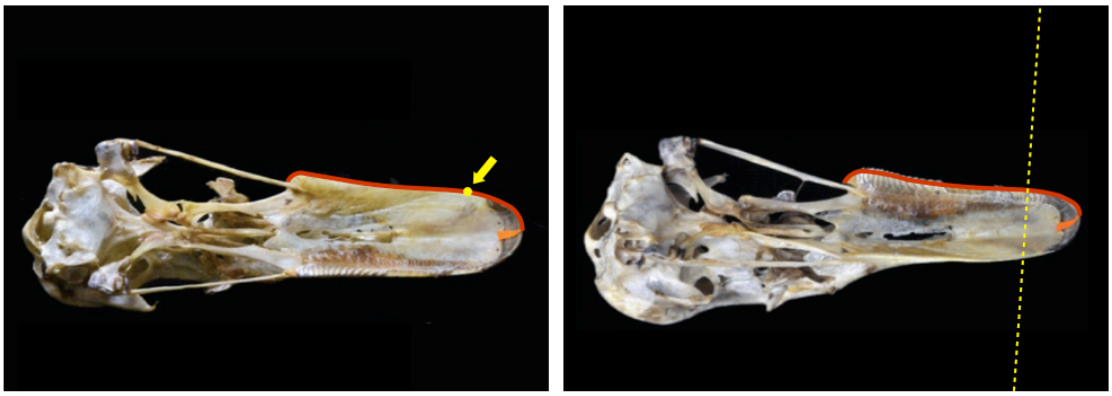
There is one complication to this, however. A point halfway along the curve in one camera view is not necessarily the same point as a point halfway along the curve in another camera view. This is due to the perspective effect of lenses. The depth of a 3D curve in one view dictates how it is projected into that image plane. Since the depth of a curve will differ depending on the perspective from which it is viewed, the same curve will be projected differently into different views.



The same curve digitized in two different camera views. Although not immediately obvious, a point halfway along the curve in one view is not necessarily the same point in 3D space as a point halfway along the curve in another.

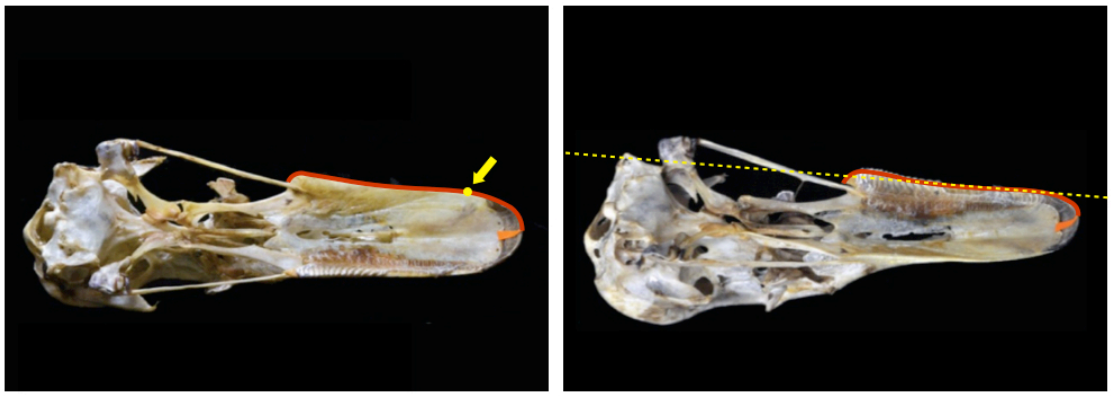
When the two cameras are calibrated, however, it's possible to use the calibration to identify corresponding points on the same curve in two camera views and reconstruct these corresponding points just as with landmarks. This is done using epipolar geometry.

The basis of epipolar geometry is that any point in one camera view must fall along a line in another camera view. This line is the epipolar line. The intersection of the epipolar line and the curve in the second view can be used to find the corresponding point on the second curve.



Demonstration of epipolar geometry. The point, in camera view 1 (indicated by a yellow arrow), must fall along a line in camera view 2. This line is the point's epipolar line and can be used to identify corresponding points along two curves.

But what if the epipolar line is parallel to the curve in the second view?



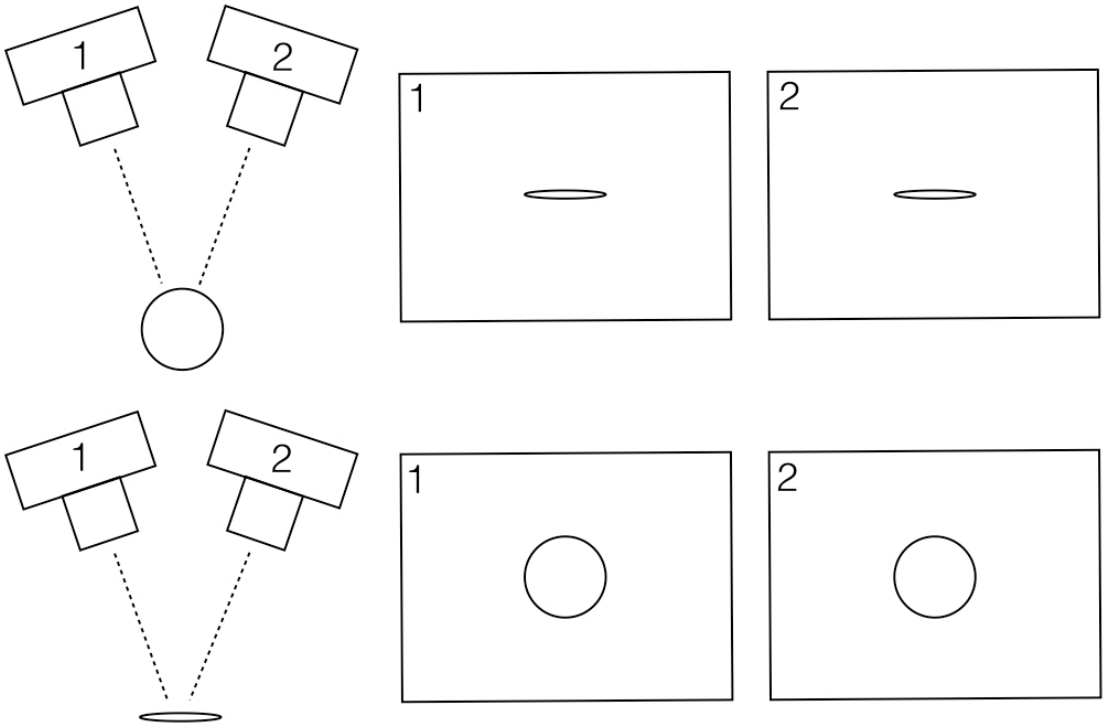
The epipolar line (yellow dashed line in right image) of the point in the first camera view (yellow point in left image) is parallel to the curve in the second view along a considerable portion of its extent. Absent other information, this makes it impossible to identify the corresponding point.

Without any additional information, this makes finding the corresponding curve point in the second view impossible. At some point, StereoMorph might include the ability to compare features along the curve in the image itself to find the corresponding point. It's best to arrange the specimen or cameras so as to minimize the chance of large sections of the curve being parallel to an epipolar line. This can be done by making a mental note of the following steps:

1. For the curve of interest, identify a 2D plane in which most of the curve lies. Of course the curve will have some three-dimensional aspect to it (or else you would not be using 3D reconstruction). But find a plane that encompasses most of the curve.

2. Position the cameras so that a line extending out each lens to the curve is orthogonal to this plane.

As an example, let's say we're photographing a curve that resembles a circle (shown in the diagram below).

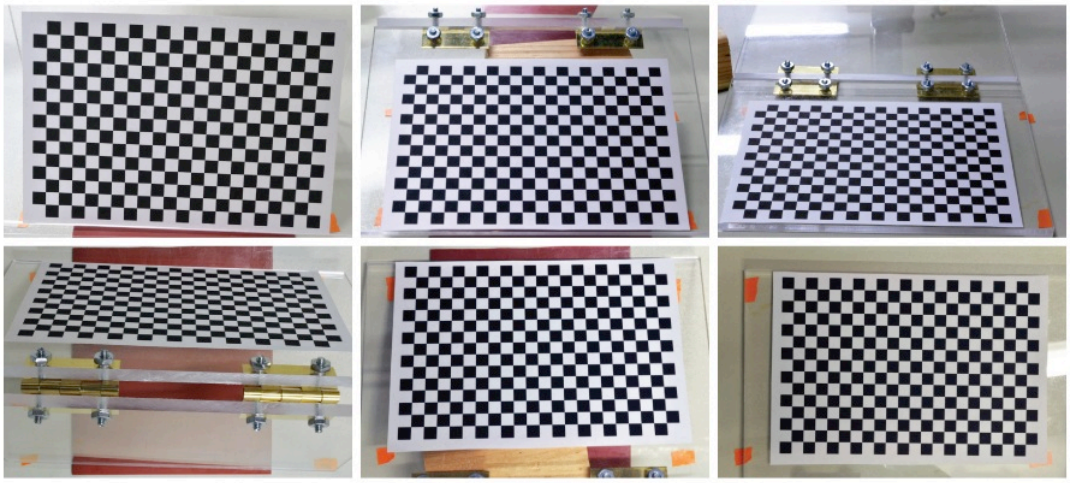


If we position the cameras so that they are viewing the curve, shown as a circle here, from the side (top left), the curve will appear in the two camera images as a nearly straight line (top middle and right). There is no way we could adequately measure the true shape of the curve because we're hardly seeing any of it in the images. Additionally, the curve points in the second view will be nearly parallel to the epipolar line of the points in the first view. In fact, these two properties are geometrically related such that if the camera views chosen do not adequately capture the major three-dimensional aspects of the curve, epipolar geometry will also be unable to accurately find the corresponding curve points.

In contrast, if the cameras are positioned so they are viewing the curve from the top (bottom left), the major aspects of the curve's shape are represented in the images and the curve in the second image will seldom be parallel to the epipolar lines of points along the curve in the first image. As a consequence, StereoMorph currently works best for reconstructing curves that do not have an excessively 3D shape (i.e. 3D spirals) because no matter how the cameras are positioned, these shapes will always have large portions that cannot be adequately captured in both camera views.

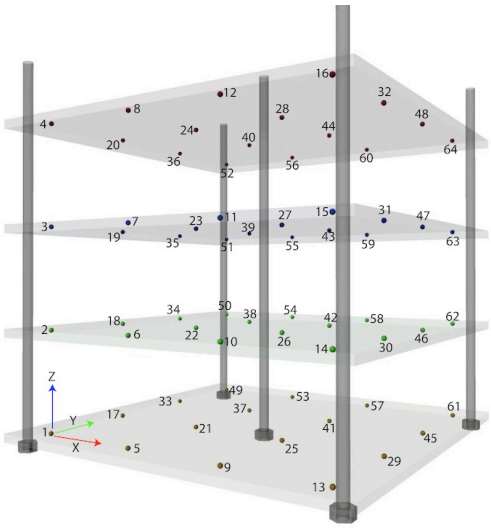
Calibrating Stereo Cameras

This section demonstrates how to photograph the checkerboard created in "Creating a Checkerboard Pattern" in different positions and orientations and use these photos to calibrate two cameras arranged in stereo.



Example of checkerboard photographed in different positions for stereo camera calibration.

Camera calibration using the direct linear transformation (DLT) method requires two sets of points: (1) a set of 3D coordinates and (2) their corresponding 2D pixel coordinates in each camera view. These two point sets are used to calculate calibration coefficients. These coefficients can then be used to reconstruct any point in 3D given its 2D pixel coordinates in at least two camera views. DLT calibration has traditionally been done using a "calibration object", typically a 3D box-shaped structure filled with markers at known 3D positions.



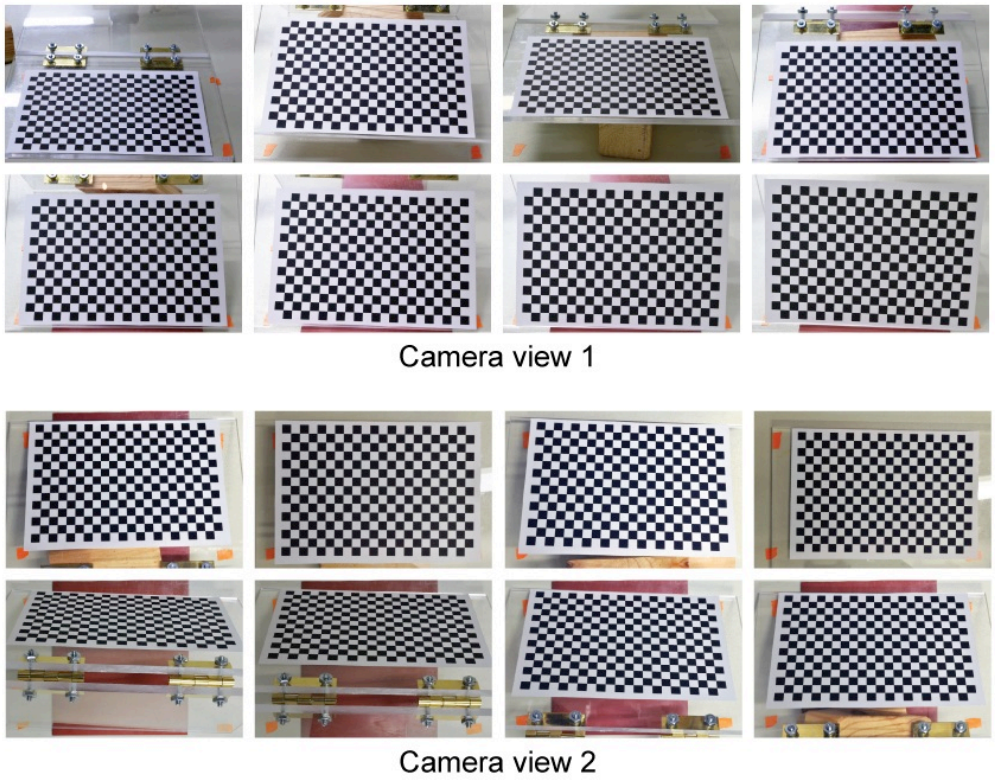
Example calibration object. From XROMM Wiki at wiki.brown.edu.

First, designing and building a 3D calibration object is not trivial. The object must have several points of known 3D position relative to one another and a sufficient number of these points, filling the 3D volume, should be visible from a single camera view. Secondly, to achieve high accuracy, these objects must be made using high precision machining. Thirdly, the calibration points must be digitized manually every time the cameras are calibrated.

Rather than use a 3D calibration object photographed in a single position, StereoMorph performs a DLT calibration using a 2D checkerboard pattern photographed in several positions and orientations. Photographs from two camera views of a planar checkerboard in a single orientation are insufficient to calibrate cameras in stereo because all of the checkerboard points lie in a single plane. Photographing a checkerboard in several different positions throughout the volume solves this problem by providing a sample of points throughout the volume. However, this poses a new problem. Each plane of points is in a different coordinate system; the 3D position and orientation of one plane relative to another is unknown.

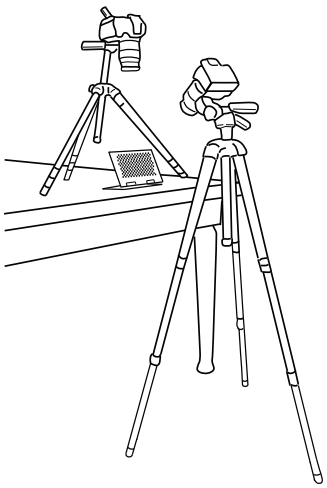
StereoMorph solves this by using the `nlminb()` function (in the R package 'stats') to estimate the six transformation parameters (three translation, three rotation) required to transform the first checkerboard into each subsequent checkerboard in 3D space. The transformation parameters that minimize the calibration error are chosen as the optimal parameters. These transformation parameters are then used to generate the 3D coordinates needed to compute the DLT calibration coefficients.

1. Once you have arranged two cameras in a stereo setup (see the previous section), photograph the checkerboard pattern in several different positions and orientations within the volume to be calibrated (the volume in which the camera views overlap).



Eight photos of the checkerboard per camera view used in this tutorial for camera calibration.

In the image above the first camera view corresponds to the camera positioned on the floor (in the sketch to the right) and the second camera view corresponds to the camera sitting on the top of the table. Note in particular that the second camera is viewing the checkerboard pattern upside-down relative to the first camera.



The camera arrangement used in this tutorial.

2. Import the photographs from different views into two different folders. For demonstration, we'll use the calibration photographs in the folder 'Calibration images' (in the StereoMorph Tutorial folder). Photographs from camera one are in the folder 'v1' (view 1) and photographs from camera two are in 'v2'. For the rest of this tutorial, views 1 and 2 will be used to refer to photographs taken from camera 1 and 2, respectively.

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

```
> library(StereoMorph)
```

We'll use the `findCheckerboardCorners()` function, introduced in the 'Auto-detecting Checkerboard Corners' section, to find the internal corners in each calibration image.

4. First, specify the number of *internal* corners in the checkerboard.

```
> nx <- 21
> ny <- 14
```

5. Then, specify the file locations of the calibration images, where to save the checkerboard corners and, if desired, where to save the verification images showing the automatically detected corners.

```
> image_file <- paste0('Calibration images/v', c(1, 2))
> corner_file <- paste0('Calibration corners/v', c(1, 2))
> verify_file <- paste0('Calibration images verify/v', c(1, 2))
```

Since the images are in two different folders ('v1' and 'v2'), the `paste0()` function is used to create a vector (example below) for the two different folders within 'Calibration images'.

```
> paste0('Calibration images/v', c(1, 2))
[1] "Calibration images/v1" "Calibration images/v2"
```

The `findCheckerboardCorners()` will automatically assign save-as filenames.

6. Call `findCheckerboardCorners()`. Note that this function can take several seconds per photo to run.

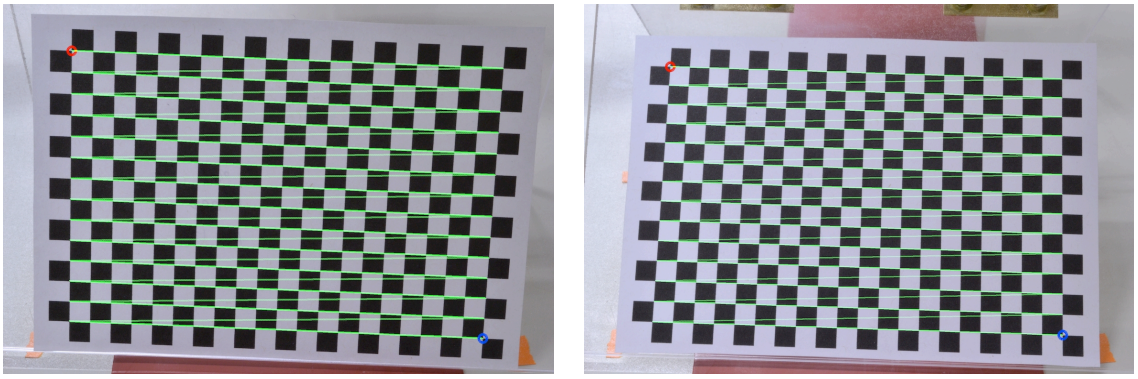
```
> corners <- findCheckerboardCorners(image.file=image_file, nx=nx,
  ny=ny, corner.file=corner_file, verify.file=verify_file)
```

By default, the `findCheckerboardCorners()` function will print its progress to the R console as each image is loaded and whether or not the specified number of internal corners were found (note that if the number of internal corners found does not match the number specified, no corners are saved).

```
Loading image 1 (DSC_0002.JPG)...
      294 corners found successfully.
Loading image 2 (DSC_0003.JPG)...
      294 corners found successfully.
...
```

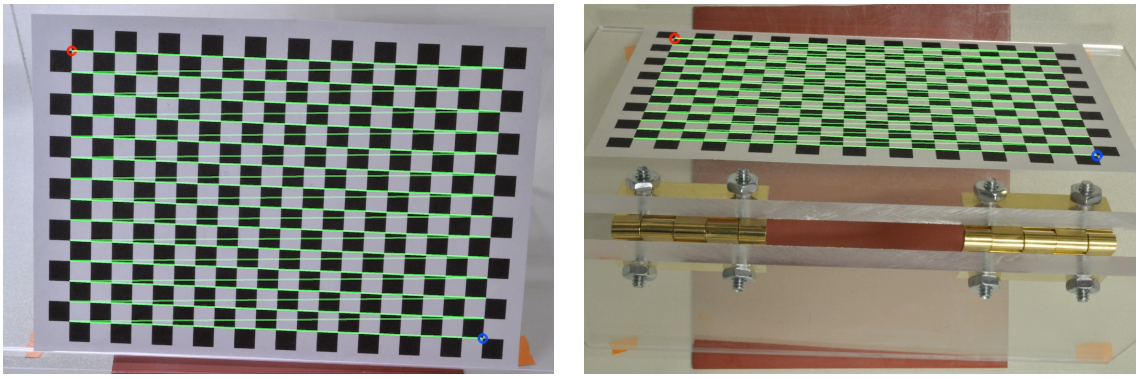
The function will find the corners successfully for all of the images in the tutorial set. When using your own images don't worry if the function fails for a couple of images; these will be ignored in subsequent steps. Four to five pairs of calibration images are usually sufficient for an accurate calibration.

7. Once `findCheckerboardCorners()` has finished running, check all of the verification images to **be sure that the order is consistent within each camera view and between the two views**.



The checkerboard in two different positions from the same camera view (view 1)

Within the same view, the corners will be returned in the same order as long as the orientation of the checkerboard does not change radically within the same set. However, in the case of this calibration the order is not the same between the two views.



An image pair from two different camera views

Although it looks like the corners in view 1 and 2 have been returned in the same order, they are actually in the reverse order. Camera 1 is viewing the checkerboard head-on while camera 2 is viewing the checkerboard from the top. The point circled in red in view 2 and closest to the red mat on the table top is the same as the point circled in blue in view 1. We can easily fix this when we import in the corners using the `readCheckerboardsToArray()` function.

8. Import the corners just saved to individual files by constructing a two-column matrix of file paths, where the columns correspond to each view. The file paths must be specified as a matrix because the `readCheckerboardsToArray()` function will use the matrix structure to separate the corners by view.

```
> corners_by_file <- cbind(
  paste0(corner_file[1], '/', paste0('DSC_000', 2:9, '.txt')),
  paste0(corner_file[2], '/', paste0('DSC_000', 2:9, '.txt')))
```

9. Call `readCheckerboardsToArray()` to read all of the corner matrices into an array; include all files – empty files will read in as `NA`s. The array will have four dimensions: the first two dimensions correspond to the rows and columns of each corner matrix (294 x 2), the third corresponds to the number of positions of each checkerboard (8) and the fourth corresponds to the number of camera views (2).

```
> corners <- readCheckerboardsToArray(file=corners_by_file, nx=nx,
  ny=ny, col.reverse=c(F, T), row.reverse=c(F, T))
```

In passing the vector `c(F, T)` to the `col.reverse` parameter ('`F`' for `FALSE` and '`T`' for `TRUE`), the function will maintain the current order of the corners in the first view and reverse the column order of the corners in second view. The same is done for the row order via the `row.reverse` parameter. This reverses the order of all the corners for each image from the second camera view. **This is essential** – the same row in each corner matrix must correspond to the same point across all of the images (among and between views).

10. Set `grid_size_cal` to the square size of the calibration checkerboard. The tutorial calibration checkerboard, measured using a precision ruler (see "Measure Checkerboard Square Size"), is approximately 6.365 mm.

```
> grid_size_cal <- 6.3646
```

11. Call `dltCalibrateCameras()`. This can take from 30 seconds to five minutes depending on the number of images and the speed of your computer. We'll just use the first six image pairs of the checkerboard. Image pairs for which corners were found in neither or only one image will be ignored.

```
> dlt_cal_cam <- dltCalibrateCameras(coor.2d=corners[, , 1:6, ],
  nx=nx, grid.size=grid_size_cal, print.progress=TRUE)
```

The function begins by reducing the grid point density.

```
Reduce grid point density (12 total)
      1) Aspect 1, View 1; Mean fit error: 0.4515 px; Max: 1.7772 px
      2) Aspect 1, View 2; Mean fit error: 0.6454 px; Max: 2.8456 px
...
```

This fits a camera perspective model to the checkerboard corner points and reduces this corner set to nine points (3x3). These nine points contain the same information as the original number of points (here, 293), but allows the optimization function to proceed much more quickly. The mean fit error should be less than a pixel.

The function then searches for the optimal parameters to transform each checkerboard, yielding a set of 3D coordinates that results in the lowest calibration error.

```
Full Transform RMSE Minimization
Number of parameters: 30
Number of points: 54
```

The first checkerboard (or grid) is fixed in 3D space at (0, 0, 0). `dltCalibrateCameras()` then searches for a set of six parameters per grid (three for translation and three for rotation) to transform each grid relative to the first. Here, we have six image pairs corresponding to a grid in six different orientations. Because the first grid is fixed, the six transformation parameters are only needed for five grids. This means a total of 30 (6 times 5) parameters must be estimated. Since the point density was reduced to nine per checkerboard, there are 54 total points (9 times 6).

The minimization begins with three grids. Once the minimization converges to an appropriate solution, the minimization adds another grid, using the previously optimized parameters as starting parameters. This proceeds sequentially until all of the input checkerboards have been included (the sequential addition of grids helps in proper convergence).

12. Use the `summary()` function to print a summary of the calibration run.

```
> summary(dlt_cal_cam)

dltCalibrateCameras Summary
      Minimize RMS Error by transformation
            Total number of parameters estimated: 30
            Final Minimum Mean RMS Error: 0.885
            Total Run-time: 35.97 sec

      Mean Reconstruct RMSE: 0.414
      Calibration Coefficient RMS Error:
            1: 0.666
            2: 0.66
      3D Coordinate Matrix dimensions: 1764 x 3
```

The reconstruction and calibration error are decent proxies for the accuracy of the calibration; an RMS error less than one is good. If the RMS error is greater than one, there might be a problem somewhere in the calibration. Testing the accuracy (covered in the next section) can show this more definitively.

The function returns the calibration coefficients in `cal.coeff`:

```
> dlt_cal_cam$cal.coeff
               [,1]          [,2]
 [1,]  2.772121e+01 -2.557153e+01
 [2,] -4.918245e-02 -5.183902e+00
 [3,]  6.226293e+00  1.498422e+00
 [4,]  2.329910e+03  2.187563e+03
 [5,]  1.128266e+00 -9.899814e-01
 [6,] -2.795001e+01  7.701885e+00
 [7,] -1.246494e+00  2.450539e+01
 [8,]  1.608486e+03  6.728743e+02
 [9,] -3.854642e-06 -4.570336e-05
[10,] -5.200086e-04 -2.396548e-03
[11,]  2.593912e-03  1.183541e-03
```

Note that there are 11 rows and two columns. The 11 rows correspond to the 11 DLT coefficients and the two columns correspond to the two camera views. This 11x2 matrix can be used to reconstruct any point, given its 2D pixel coordinates in both camera views. That is, the calibration coefficients are the only values we need from this section to perform the rest of the steps in this tutorial.

13. Save the calibration coefficients to a text file.

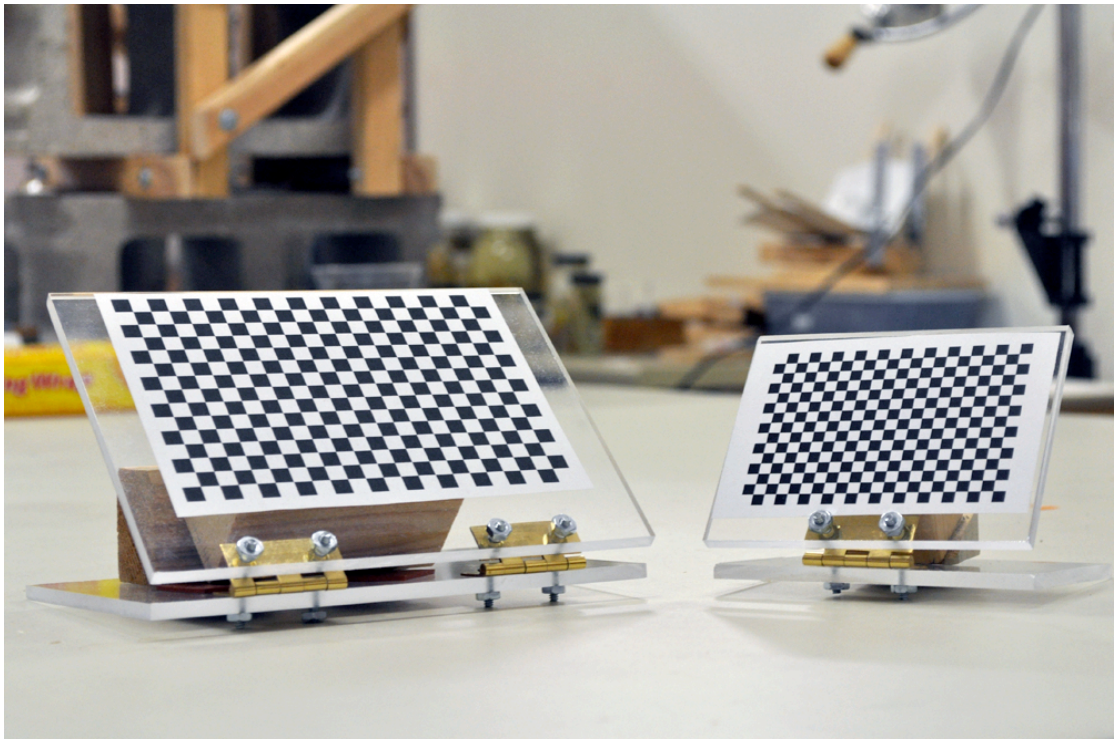
```
> write.table(x=dlt_cal_cam$cal.coeff, file="cal_coeffs.txt",
  row.names=F, col.names=F, sep="\t")
```

The next section will detail how to determine the accuracy of the calibration.

Testing the Calibration Accuracy

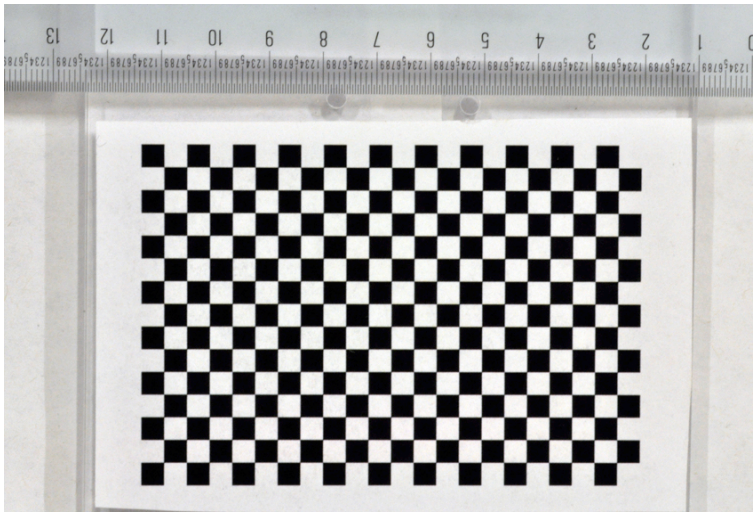
The previous section demonstrated how to use a checkerboard pattern of known square size and the `dltCalibrateCameras()` function to calibrate two cameras in stereo. While `dltCalibrateCameras()` returns calibration and reconstruction RMS errors, these are measures of how well the DLT camera model fits the calibration points and not the reconstruction accuracy per se. Moreover, the RMS errors are scale-independent. If we have not measured the calibration checkerboard square size correctly, the RMS errors will be unaffected but our reconstructions will be improperly scaled. This section will demonstrate how to use a checkerboard (ideally having a square size different from that used in the calibration step) to test the accuracy of a calibrated stereo camera setup.

1. Repeat the steps in "Creating a Checkerboard Pattern" to create another checkerboard of a different square size than that used in the calibration. This will allow you to test whether you have the proper scaling for the calibration checkerboard (if they had the same square size, we would not be able to test this). For this tutorial, I used a checkerboard printed at 9% scaling for the calibration and a checkerboard printed at 6% scaling to test the calibration accuracy.



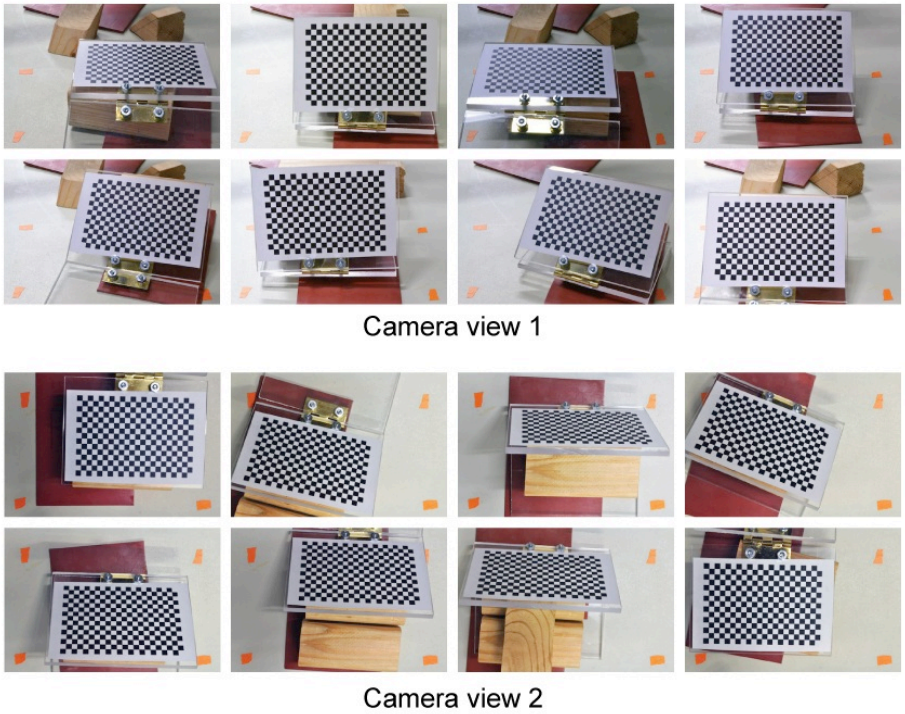
Two checkerboards, printed at 9% and 6% scaling. In this tutorial, the cameras are calibrated with the left pattern and the calibration is tested with the right.

2. Repeat the steps in "Measuring Checkerboard Square Size" to measure the square size of the test checkerboard pattern. The 6% scaled test checkerboard in this tutorial has a square size of 4.233 mm when measured using a precision rule.



Measuring the square size of the test checkerboard pattern (21 x 14, printed at 6% scaling).

3. Take photographs of the new checkerboard pattern as in "Calibrating Stereo Cameras", ensuring a sampling of points throughout the calibrated volume (anywhere the object you're digitizing could conceivably be).



Eight photos of the test checkerboard (a different size than the calibration checkerboard) per camera view used in the test calibration step.

4. Import the photographs from different views into two different folders. For demonstration, we'll use the test calibration photographs in the folder 'Test Calibration images' (in the StereoMorph Tutorial folder). Photographs from camera one are in the folder 'v1' and photographs from camera two are in the folder 'v2'.

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

```
> library(StereoMorph)
```

Just as in the calibration step, we'll use the `findCheckerboardCorners()` function to find the internal corners in each calibration image.

6. Specify the number of internal corners in the checkerboard.

```
> nx <- 21
> ny <- 14
```

7. Specify the file locations of the test calibration images, where to save the checkerboard corners and, if desired, where to save the verification images.

```
> test_image_file <- paste0('Test images/v', c(1, 2))
> test_corner_file <- paste0('Test corners/v', c(1, 2))
> test_verify_file <- paste0('Test images verify/v', c(1, 2))
```

8. Call `findCheckerboardCorners()`.

```
> corners <- findCheckerboardCorners(image.file=image_file, nx=nx,
  ny=ny, corner.file=corner_file, verify.file=verify_file)
```

The function will find the corners successfully for all but three of the images in the tutorial set. Again, don't worry if the function fails for a couple of images; these will be ignored in subsequent steps.

9. Once `findCheckerboardCorners()` has finished running, check all of the verification images to **be sure that the order is consistent within each camera view and between the two views**. For the tutorial images the corners in the first view are in the reverse order relative to the order in the second view (as was also seen with the calibration images). This will be corrected when the corners are read in from the corner files.

10. Import the corners just saved to individual files by constructing a two-column matrix of file paths, where the columns correspond to each view.

```
> corners_by_file <- cbind(
  paste0(test_corner_file[1], '/', paste0('DSC_00', 11:20, '.txt')),
  paste0(test_corner_file[2], '/', paste0('DSC_00', 11:20, '.txt')))
```

11. Call `readCheckerboardsToArray()` to read all of the corner matrices into an array; include all files – empty files will read in as `NA`s.

```
> test_corners <- readCheckerboardsToArray(file=corners_by_file,
  nx=nx, ny=ny, col.reverse=c(F, T), row.reverse=c(F, T))
```

As in the calibration step, the vector `c(F, T)` is passed to `col.reverse` and `row.reverse` to reverse the order of the corners from the second view relative to the first.

Since we are importing corners from 10 images, the third dimension of the `test_corners` array is 10.

```
> dim(test_corners)
[1] 294   2  10   2
```

12. Set `grid_size_test` to the square size of the test calibration checkerboard. The tutorial test calibration checkerboard, measured using a precision ruler (see "Measuring Checkerboard Square Size"), is approximately 4.323 mm.

```
> grid_size_cal <- 4.2327
```

13. Load in the calibration coefficients.

```
> cal.coeff <- as.matrix(read.table(file="cal_coeffs.txt"))
```

14. Call `dltTestCalibration()`. Image pairs for which corners were found in neither or only one image will be ignored.

```
> dlt_test <- dltTestCalibration(cal.coeff=cal.coeff,
  coor.2d=test_corners, nx=nx, grid.size=grid_size_test)
```

15. Use the `summary()` function to print a summary of the accuracy test.

```
> summary(dlt_test)

dltTestCalibration Summary
      Number of grids: 7
      Number of points: 1029
      Aligned ideal to reconstructed (AITR) point position errors:
            AITR RMS Errors (X, Y, Z): 0.02343, 0.01799, 0.02063
            Mean AITR Distance Error: 0.03297
            AITR Distance RMS Error: 0.03605
      Inter-point distance (IPD) errors:
            IPD RMS Error: 0.03090397
            IPD Mean Absolute Error: 0.02392322
            Mean IPD error: -0.002694208
      Adjacent-pair distance errors:
            Mean adjacent-pair distance error: 0.0004335479
            Mean adjacent-pair absolute distance error: 0.01592422
            SD of adjacent-pair distance error: 0.01840659
      Epipolar errors:
            Epipolar RMS Error: 1.594549 px
            Epipolar Mean Error: 1.555131 px
            SD of Epipolar Error: 0.352441 px
```

Now to unpack this summary. One of the challenges to assessing the accuracy of a DLT calibration is that any reconstructed points will be in the coordinate system of the points used to calibrate the cameras. This means that even if we had an object with points of known 3D position, reconstructing the object using the DLT coefficients would yield 3D points arbitrarily translated and rotated to somewhere in 3D space. We'd have to perform some alignment step in order to compare the reference 3D points to the reconstructed points. The alignment will cause underestimation of larger errors and overestimation of smaller errors. The best solution is to use a variety of different accuracy metrics that, when taken together provide a complete assessment of accuracy.

`dltTestCalibration()` provides four assessments of calibration accuracy:

1) **Aligned ideal to reconstructed (AITR) error.** For every checkerboard that is reconstructed, the function takes an ideal checkerboard of the same dimensions (uniform square sizes and planar) and aligns the ideal corner points to the reconstructed corner points using least squares alignment. Then, the distance is measured between each ideal point and its corresponding reconstructed points. If the points were perfectly reconstructed, the ideal and reconstructed points would overlap perfectly. The "AITR RMS (root mean square) errors" are first measured along each axis (x, y and z in the coordinate system of the calibration points). This is one way to quantify how accuracy differs along different dimensions. The

"mean AITR distance error" is the mean 3D distance between ideal and reconstructed points. This will usually be larger than any of the single axis errors since it incorporates error along all axes. This is also returned as RMS error. One disadvantage of this measure is that the ideal grid will be pulled toward areas of high error to minimize the total alignment error. This can cause underestimation of larger errors and overestimation of smaller errors.

2) **Inter-point distance (IPD) error.** This 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 actual distance in an ideal grid (again, uniform square sizes and planar). This measure avoids the problems with the alignment step in AITR error but doesn't readily provide any information of error along a particular dimension (although this could perhaps be assessed by taking into account the positions of the reconstructed points). The distance errors are returned as "IPD RMS Error" and "IPD Mean Absolute Error". The reconstructed distances can be either shorter or longer than the actual distance. The "Mean IPD error" takes the mean of these errors. If there is no bias toward over- or underestimation of distance this should be nearly zero. The results will differ slightly at each run because the point pairs are chosen randomly.

3) **Adjacent-pair distance errors.** This is identical to IPD error except that randomly chosen points are adjacent on the grid. This means the ideal distances are uniform and the minimum possible distance for IPD error assessment. This is a common stereo camera error assessment used in the literature (e.g. Tashman & Anderst 2003; Brainerd *et al.* 2010). Since the points 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. These errors will usually be slightly less than the general IPD errors since error is likely to be greater for points at a greater distance from one another.

4) **Epipolar errors.** In a stereo camera setup, 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 input to `dltTestCalibration()` includes the same point in two or more camera views, we can use epipolar error to assess calibration accuracy. Epipolar error must be low to identify corresponding points along curves in different views. The mean and standard deviation of epipolar error is returned in pixels and should be less than a couple of pixels.

So how accurate is the calibration? Millimeters were used as units in measuring the checkerboard square size so all values not in pixels are in millimeters. The

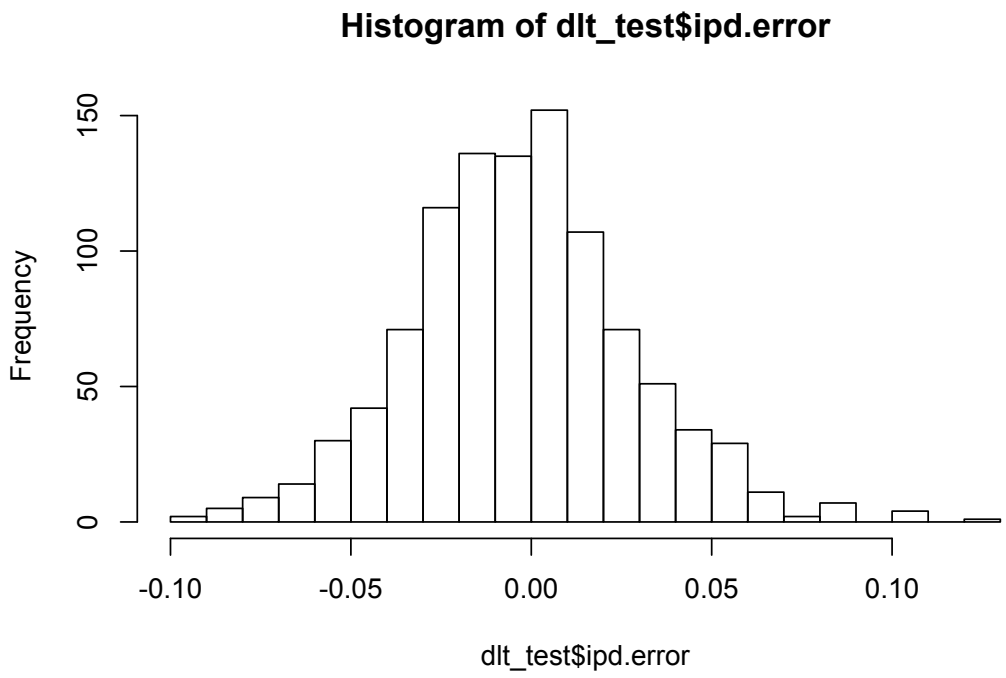
positional errors along all three dimensions are around 20 microns on average and the different distance errors range from 16-36 microns. The "mean IPD error" shows a slight bias towards underestimating inter-point distances but only by 3 microns on average. And for points closer together, this bias is less than a micron (0.0003 mm). Lastly, the epipolar error is less than two pixels on average, with a low standard deviation.

It's important to compare these errors to the total calibrated volume. Since the calibration checkerboard is 21 x 14 and the square size is approximately 6.36 mm, each dimension of the calibrated volume is at least 89 mm (14*6.36). 20 microns represents 0.02% positional error (0.020 mm/89 mm), an extremely low error.

`dltTestCalibration()` returns all of the values used to calculate the stats in the summary output. So `plot()` and `hist()` can be used to look at the full error values. For instance, we can look at a histogram of all the IPD errors.

16. Create a histogram of all the inter-point distance errors, using `hist()`.

```
> hist(dlt_test$ipd.error, breaks=20)
```

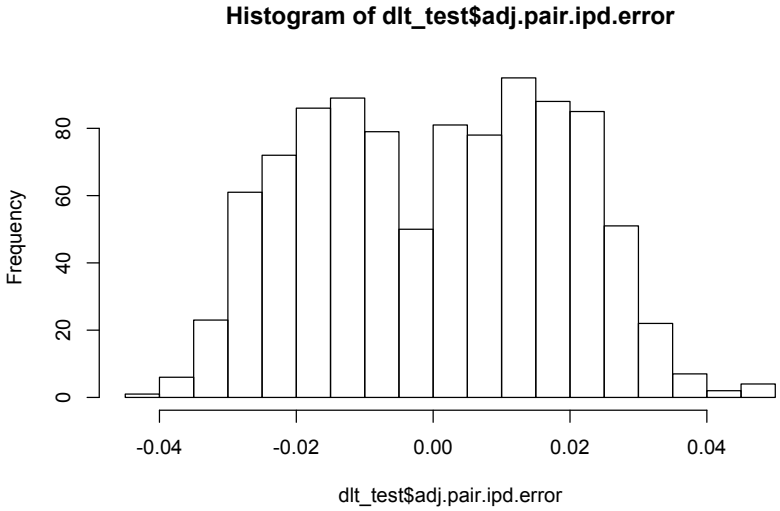


A histogram of inter-point distance errors (in mm); *N* = 1029.

The histogram shows that nearly all of the distances measured between random pairs of points across each checkerboard are within 0.100 mm of their actual distance.

17. Create a histogram of all the inter-point distance errors, considering only adjacent internal corners (in this case, points within about 4 mm of each other).

```
> hist(dlt_test$adj.pair.ipd.error, breaks=20)
```

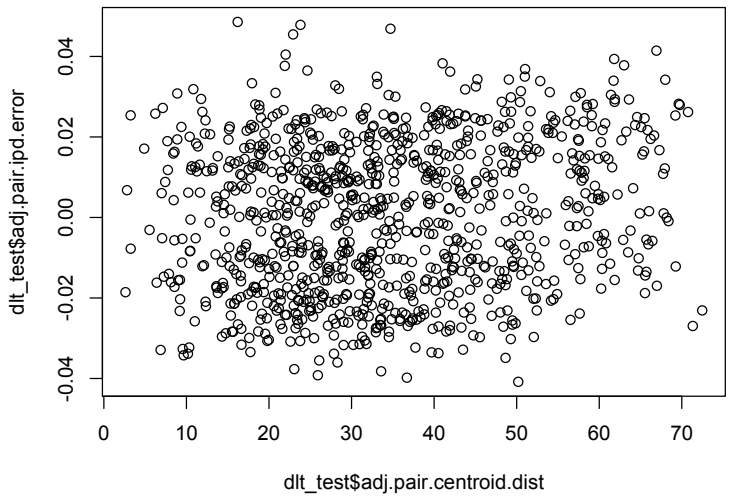


A histogram of adjacent inter-point distance errors (in mm); $N = 980$.

When considering only random pairs of adjacent points on each checkerboard, nearly all of the inter-point distances are within 0.040 mm of their actual distance.

18. To test how reconstruction error varies as a function of the distance from the center of the calibrated volume (i.e. do reconstructed points in the periphery of the calibrated space have a higher error than points in the middle?), plot the adjacent inter-point distance as a function of the distance of each adjacent pair from the centroid of all adjacent pairs (an approximate center of the calibrated volume).

```
> plot(dlt_test$adj.pair.centroid.dist, dlt_test$adj.pair.ipd.error)
```

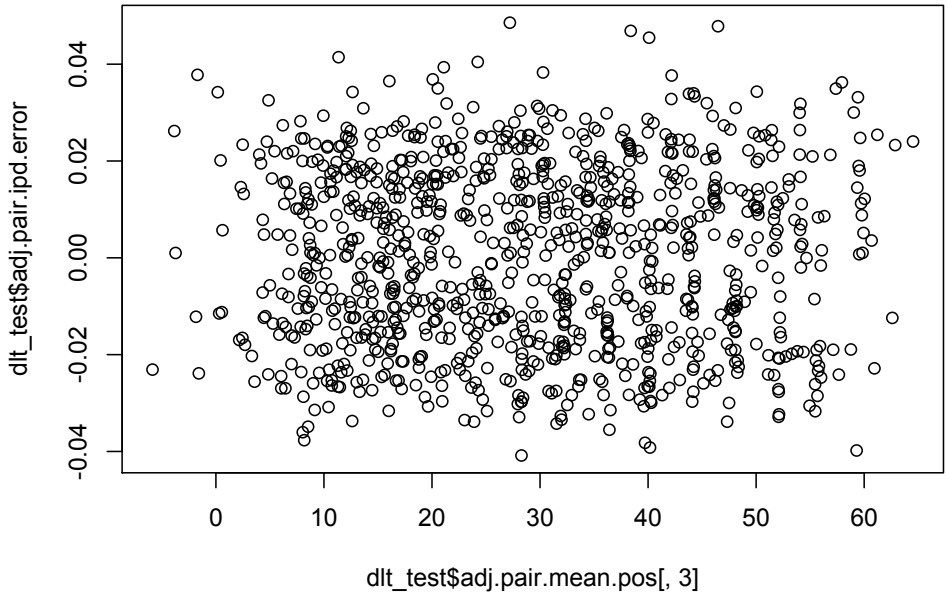


IPD error for adjacent points versus the distance of the adjacent pairs from the centroid.

There is no obvious trend in the plot, indicating that reconstruction error is uniform throughout the calibrated volume.

19. To test how reconstruction error varies as a function of the position along a particular axis, plot the adjacent inter-point distance as a function of the mean position of each adjacent pair along an axis.

```
> plot(dlt_test$adj.pair.mean.pos[, 3], dlt_test$adj.pair.ipd.error)
```



IPD error for adjacent points versus the mean position of adjacent pairs along the z-axis.

There is also no obvious trend in this plot, providing a second confirmation that reconstruction error is uniform throughout the calibrated volume. The axes here are defined in the same coordinate system as the 3D calibration coordinates estimated in the calibration step. Thus, the orientation of these axes relative to the cameras is arbitrary and will depend on the orientations of the checkerboard patterns used in the calibration.

Now that the cameras are accurately calibrated, the next section will provide instructions on photographing an object for the collection of shape data.

Photographing an Object

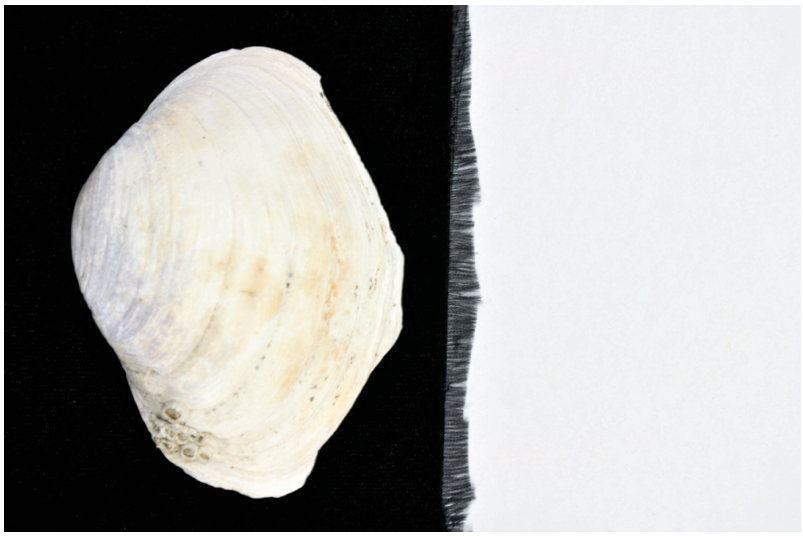
This section provides tips on how to photograph an object for shape data collection.

Recommended materials for this section:

- A yard or less of black velvet

This is the step where the DLT method has a major advantage over other 3D morphometric methods. Once the cameras are calibrated, the number of objects that can be photographed is only limited by the time it takes to position and photograph each object.

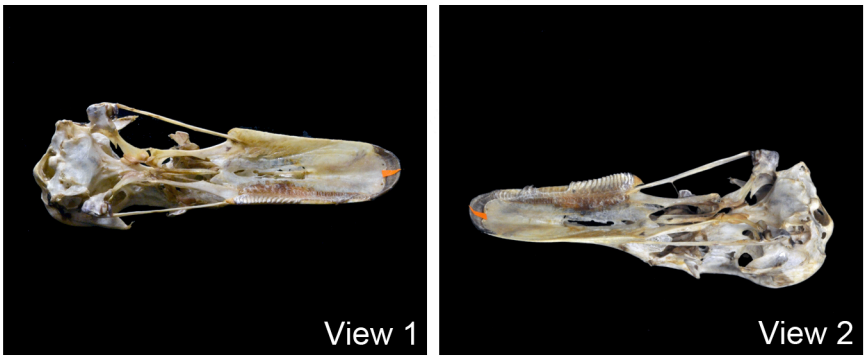
It is best to have a uniform background that provides good contrast to your specimen. First, this can decrease the photo size by as much as half (encoding a large black space takes up less space than a multi-colored, noisy background). If you're taking several hundred photographs this is advantageous for data storage. 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 stuff available at fabric stores works great and only costs about $10 a yard.



A shell on black velvet. Black velvet works great as a solid, black background.

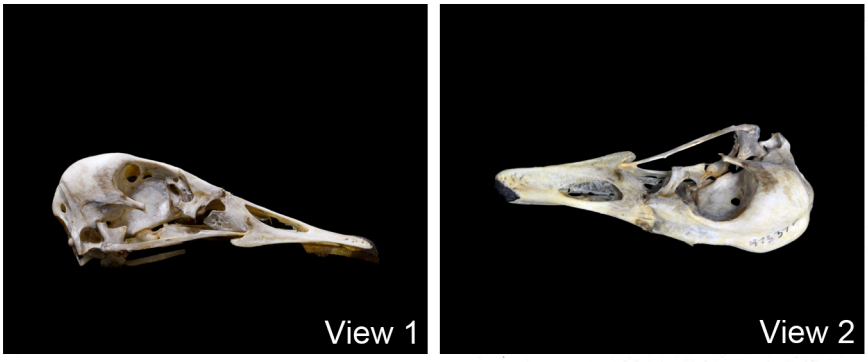
If you need to collect landmarks from several different places on an object that are not visible in a single camera view, reposition the object a few times, taking a photograph from both camera views each time. For the StereoMorph functions, these different orientations of the object are referred to as "aspects".

The tutorial data set contains landmarks and curves from these three different aspects of a Canada Goose skull. The first aspect provides views of the ventral aspect (underside) of the skull.



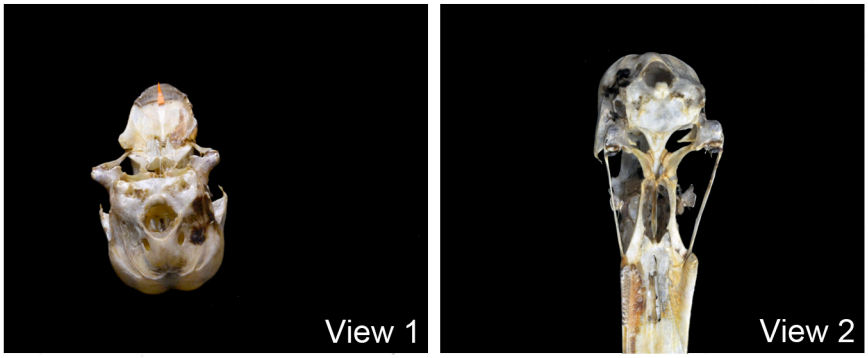
First aspect for digitizing landmarks on the ventral side of the skull. The left and right images are the first and second camera views. The specimen is in the same position in both images, just viewed from different perspectives.

From the second aspect, landmarks on the lateral or side aspect of the skull can be digitized.



Second aspect for digitizing landmarks on the ventral aspect of the skull.

And the third aspect offers views of the back of the skull.



Third aspect for digitizing landmarks on the lateral aspect of the skull.

Depending on the data you want to collect you might be able to get away with a single image of each specimen, in which case landmarks and curves only have to be

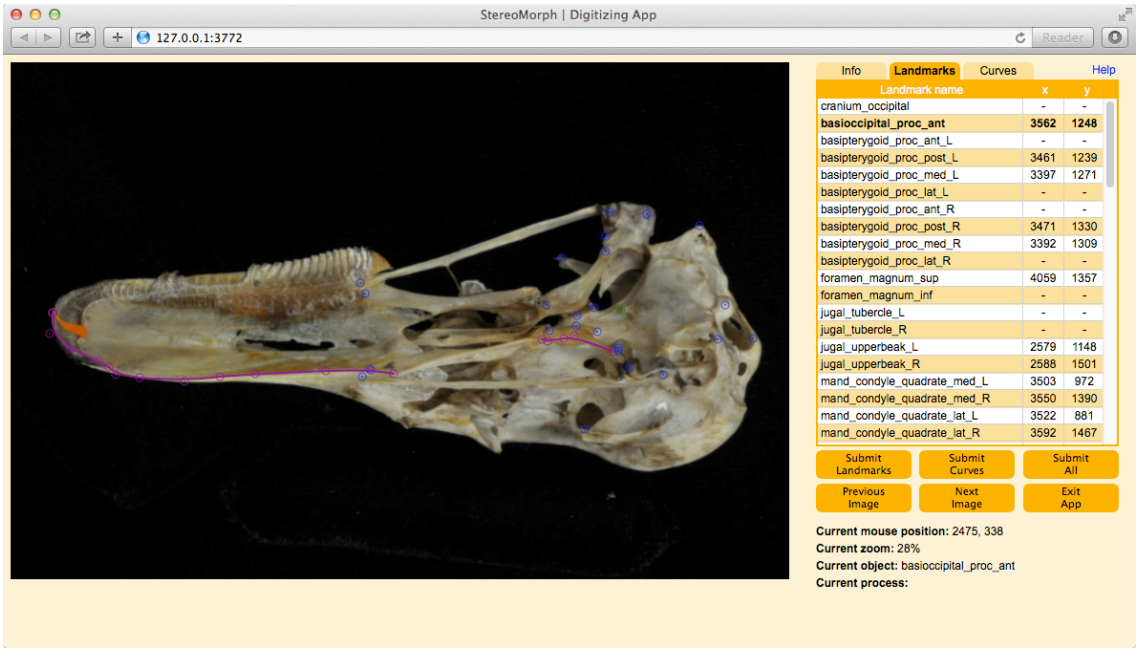
digitized in two images. If you have multiple aspects, you will need some overlap in landmarks among the images (at least three, preferably five to six) in order to combine all of the points into a single 3D point set (detailed in the section "Unifying, Reflecting and Aligning"). However, you don't have to digitize the same landmarks in every aspect.

Lastly, sometimes it's necessary to use a reference marker if it's difficult to find the exact same point between two views. I do this when digitizing beaks with a broad, flat tip. If using museum specimens, one should of course use tape that does not leave a residue.

No-residue tape can be used for points that are difficult to identify exactly in two different views.

Digitizing Photographs

The StereoMorph package provides a new, easy-to-use digitizing application for collecting landmark and Bézier curves from photographs. Even if you don't use StereoMorph for collecting 3D landmarks and curves, you might find the digitizing app useful for collecting 2D data from photographs. The app runs in a user's default web browser. Safari, Chrome and Opera all provide full compatibility with the app (Firefox does not allow some features of the app and the app is not tested for Internet Explorer). Although the app runs in a browser you do not have to be connected to the internet to use it. The app runs on a local server, with the R package 'shiny' handling communication between the browser and R console. In this step, I'll provide an overview of how to get started using the app.



StereoMorph Digitizing Application.

1. Load the StereoMorph library into the current R session, if not already loaded and ensure that the StereoMorph Tutorial folder is your current working directory.

```
> library(StereoMorph)
```

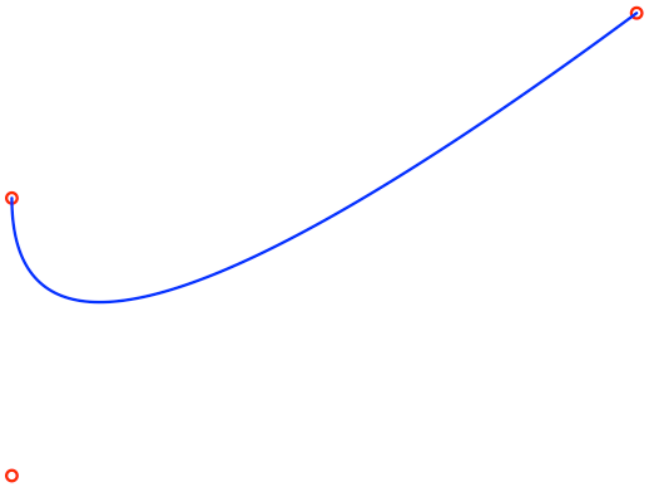
2. Call the function `digitizeImage()` with input parameters specifying the image(s) to be digitized, where the landmarks and curves should be saved and the names of the landmarks and curves to be collected. Using the files in the StereoMorph Tutorial folder, the function call would look like this:

```
> digitizeImage(
  image.file = 'Object Images',
  landmarks.file = 'Landmarks 2D',
  control.points.file = 'Control points 2D',
  curve.points.file = 'Curve points 2D',
  landmarks.ref = 'landmarks_ref.txt',
  curves.ref = 'curves_ref.txt')
```

The first argument, `image.file`, is the only required argument. It specifies the file path of the image or images to be digitized. Here it's a folder that contains all the object images in the tutorial folder. You can also input a vector of file paths if you only want to digitize a particular set of images. If you input several photographs into `digitizeImage()` you can switch from one photo to the next within the app.

The argument `landmark.file` is where the landmarks will be saved to (or loaded from if some landmarks have already been digitized). If you input a folder, the filenames of the landmark files will be saved with the same as the image names, only with the extension ".txt". If you want different names for the landmark filenames, you can input the file names as a vector.

There are two curve file inputs. The `control.points.file` specifies where to save the control points. These are the points (the red points in the image below) that you can drag around to define and adjust the curve shape. The `curve.points.file` specifies where to save the points that actually make up the curve. These are the several hundred points at single pixel spacing (the blue points in the image below) that describe the actual curve and can be used in subsequent analyses.



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

The landmarks to be digitized are input via `landmarks.ref`. In the example above, we've input a file path, `landmarks_ref.txt`, with the landmarks listed as a single column, each separated by a new line.

```
cranium_occipital
basioccipital_proc_ant
basipterygoid_proc_ant_L
...
```

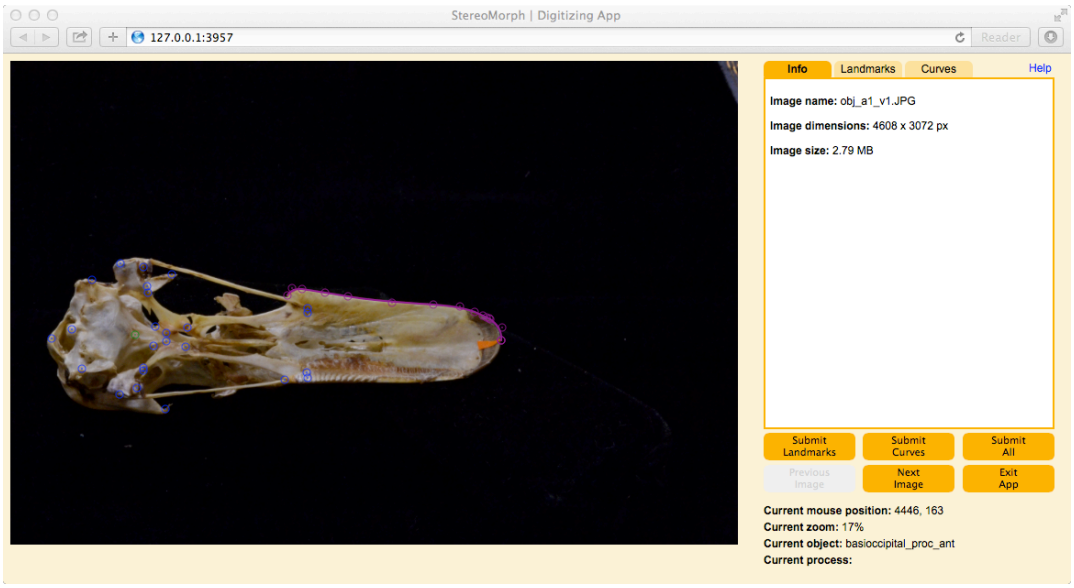
`landmarks.ref` can also be a vector of landmark names, rather than a file path.

The curves to be digitized are also input as a file path 'curves_ref.txt'. In 'curves_ref.txt', the curves are in the form of a three-column matrix. The StereoMorph Digitizing App assumes that all curve start and end points are also landmarks. So the curves to be digitized are defined using a matrix where the first column is the name of the curve, the second column is the start point/landmark and the last column is the end point/landmark.

```
tomium_R      upperbeak_tip          upperbeak_tomium_prox_R
orbit_L       preorbital_proc_L      postorbital_proc_L
...
```

In the above curve reference matrix the first curve, *tomium_R*, starts at the landmark *upperbeak_tip* and ends at the landmark *upperbeak_tomium_prox_R*. `curve_ref` can also be a matrix of reference curves, rather than a path to the file containing the matrix.

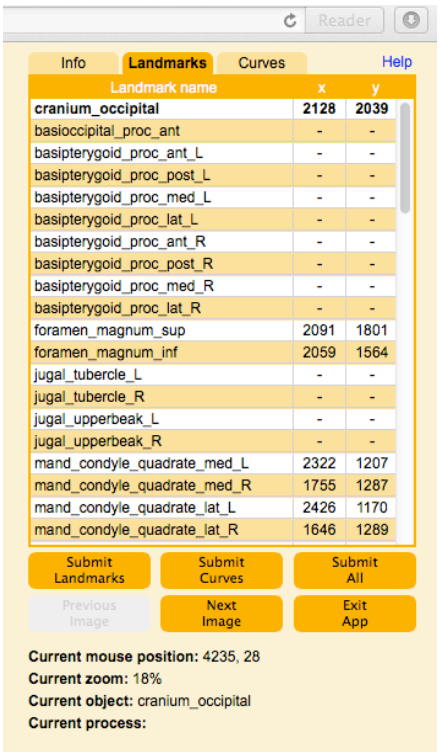
Once you call `digitizeImage()`, the digitizing app will open in your default web browser, displaying the first image in the Object images folder.



StereoMorph Digitizing Application after loading first image.

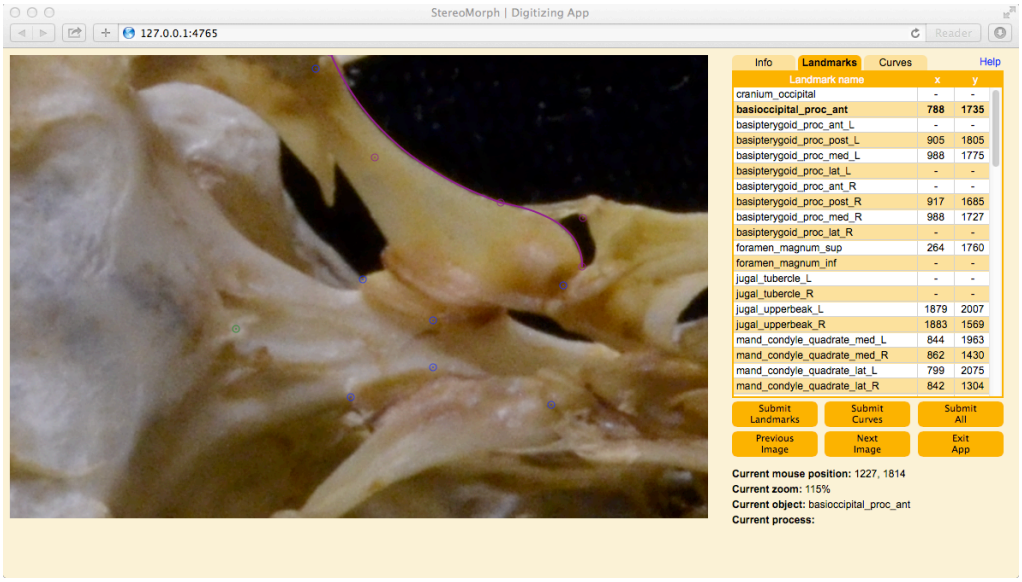
The left two-thirds of the window are the image frame. This is where you can navigate around the image and add landmarks and curves using your mouse or trackpad. The right two-thirds are the control panel, for viewing and saving landmark/curve lists and navigating between different images.

The 'Info' tab contains some basic information about the image, the 'Landmarks' tab contains the list of landmarks and the 'Curves' tab contains the list of curves.



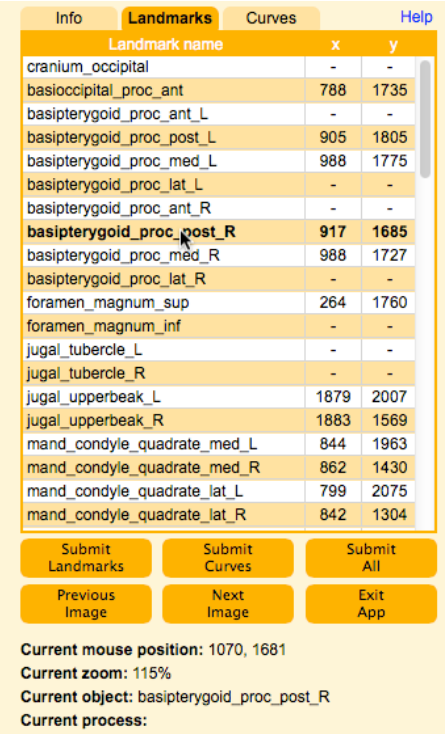
StereoMorph Digitizing App control panel.

3. You can navigate around the image in the image frame using the same basic mouse actions as in Google Maps. Position your cursor somewhere over the image and scroll just as you would scroll up and down a web page (using either the scroll wheel of a mouse or the scroll feature on a trackpad). This will zoom in and out of the image. To more quickly navigate around the image, the zoom tracks the position of your cursor and zooms in and out of particular region of the image based on the current position of your cursor. For instance if you wanted to zoom in to the bottom-left corner of the image, you would position your cursor in the bottom-left of the image and scroll in the appropriate direction. This will increase the size of the image while simultaneously positioning the bottom-left corner of the image into the center of the image frame.



Zoom into the object for precise landmark positioning.

4. Click and drag somewhere on the image. This will cause the image to move with your cursor. Just be sure that your cursor is not over a selected landmark or control point - this will cause the marker to move rather than the image.



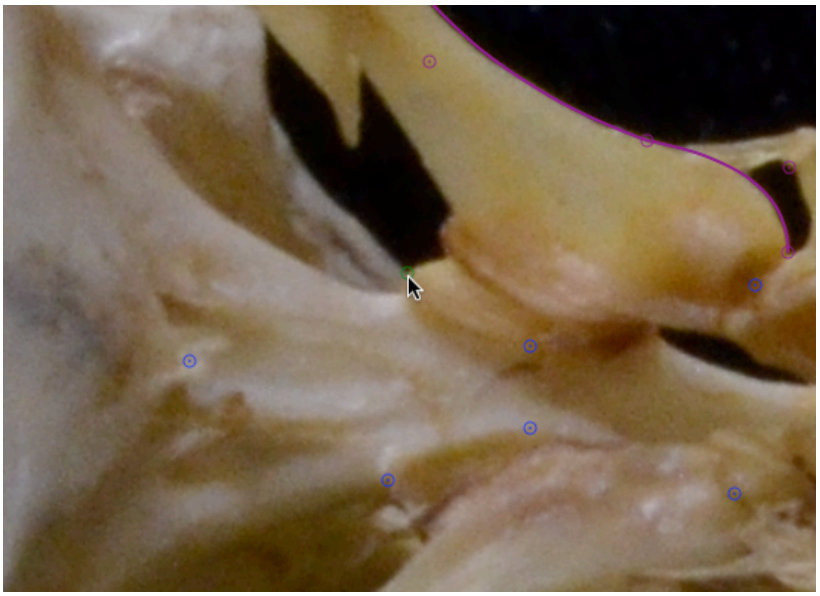
Selecting a landmark with the cursor. Once selected, the corresponding row will become bold.

5. Click on the 'Landmarks' tab in the control panel. This lists all of the landmarks input via `landmarks.ref` and their corresponding pixel coordinates.

6. To add, move or delete a landmark, you must first select that landmark. Click on the corresponding row in the landmarks table or, if the marker is already digitized, double-click on the landmark itself with the mouse (pressing the letter 'x' while the cursor is over the landmark in the image frame is a keyboard shortcut).

7. Once the landmark is selected, set its location by double-clicking anywhere over the image (or positioning the cursor and pressing 'x'). You can then move the landmark by clicking-and-dragging it or using the arrows on the keyboard.

8. To delete a landmark, just type 'd' once the landmark is selected. Landmarks with '-' values in the landmarks table will be ignored when saving.

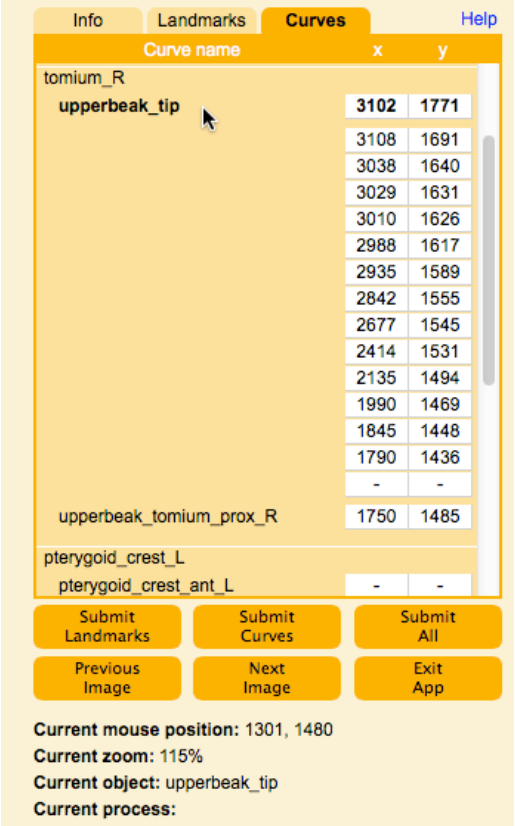


Move a landmark by clicking-and-dragging with the mouse cursor.

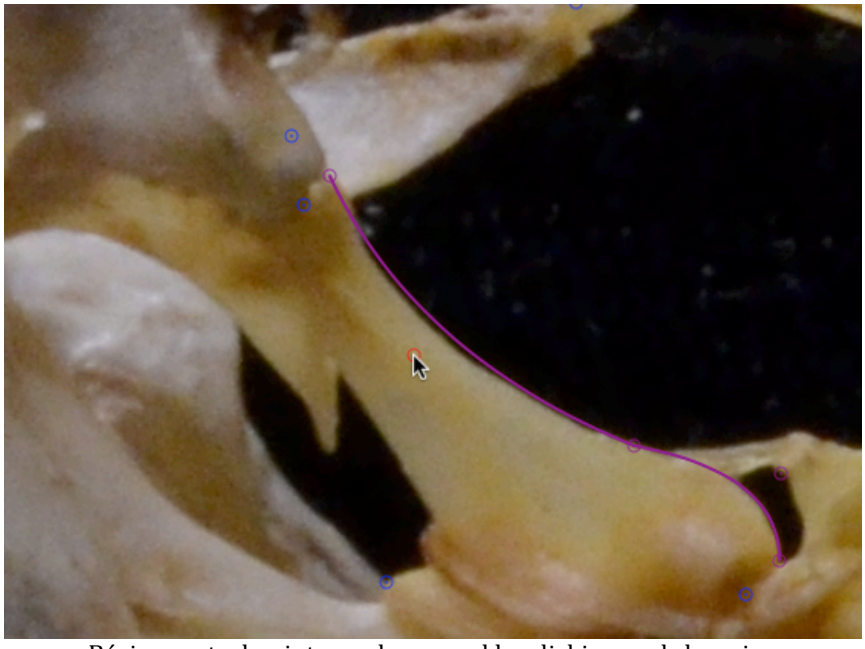
9. Click on the 'Curves' tab in the control panel to access the digitized Bézier curves.

This lists all of the curves input via `curves.ref` and their corresponding pixel coordinates, including the starting and ending control points (both are also treated as landmarks) and all the control points in between that define the Bézier curve. The app treats curves with more than three control points as Bézier splines (a string of Bézier curves, joined at the ends).

The control points can be added and moved in same way as the landmarks: by selecting the control point, double-clicking somewhere in the image frame and then clicking-and-dragging or using the arrow keys to reposition. Control points can also be deleted by pressing 'd' while the point is selected.

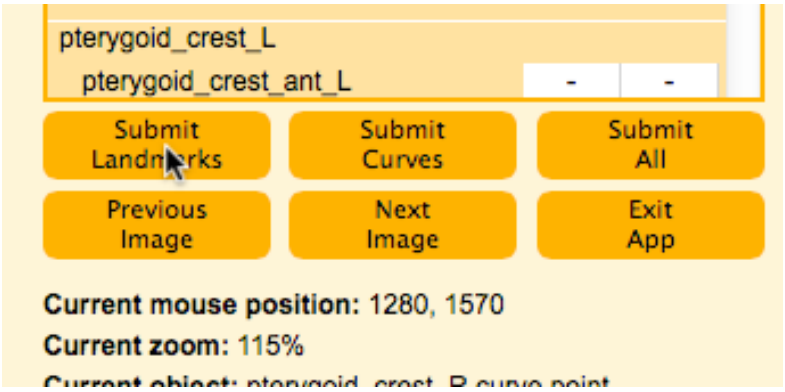


The Curves tab lists the start, end and middle control points for every Bézier curve.



Bézier control points can be moved by clicking-and-dragging with the cursor or using the arrow keys.

10. Once you have collected landmarks or curves, press one of the three submit buttons in the control panel to save them. 'Submit Landmarks' only saves the landmarks and 'Submit Curves' only saves the curves (both control points and curve points). 'Submit All' saves both the landmarks and curves. This saves them to the location you specified in the input to `digitizeImage()`.



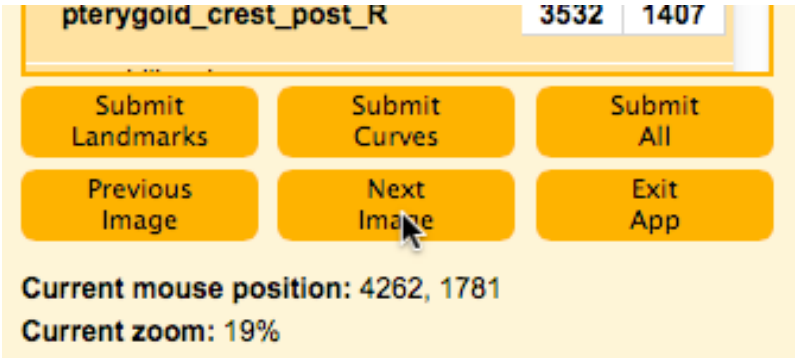
Use the 'Submit' Buttons in the bottom of the control panel to save the landmarks and curves.

If a `curve.points.file` was included, R will also save a file with all of the curve points. They will look something like this,

```
pterygoid_crest_R0001 1140 1672
pterygoid_crest_R0002 1140 1671
pterygoid_crest_R0003 1140 1670
pterygoid_crest_R0004 1140 1669
...
```

at single pixel spacing and with numbers added after the curve name to indicate the point order. These are the points we'll use in the next step to reconstruct the curves into 3D.

11. Click 'Next Image'. This will move to the next image if more than one image or a folder of images is specified in `image.file`.



Move to the previous or next image using the buttons at the bottom of the control panel.

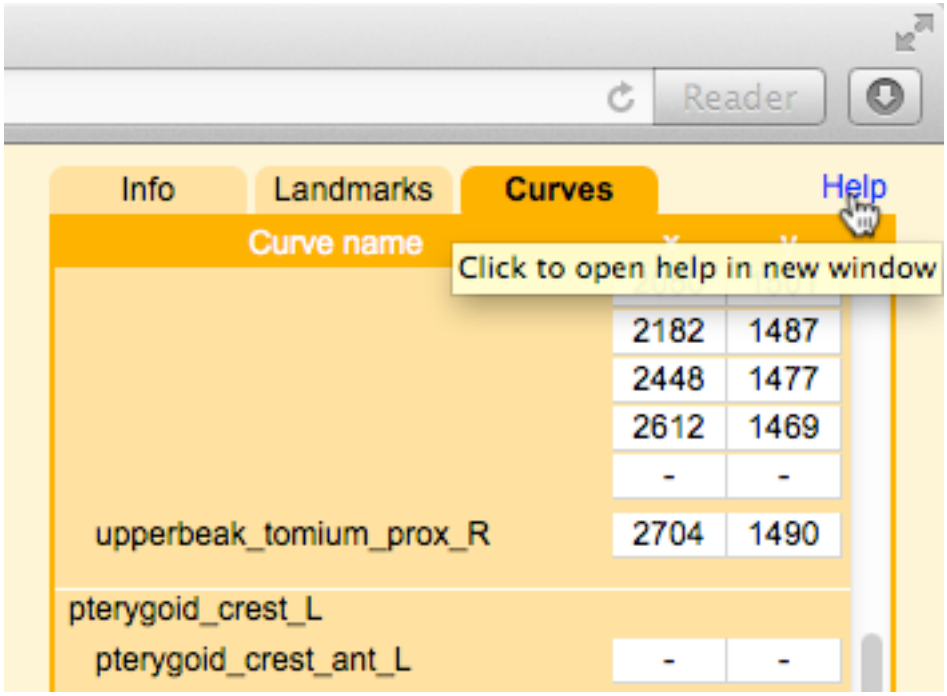
Although, this tutorial uses the full features of the app, you can also use the app to digitize only landmarks or only curves. To only digitize landmarks, call `digitizeImage()` with only a landmark file and landmark reference list.

```
> digitizeImage(
  image.file = 'Object Images',
  landmarks.file = 'Landmarks 2D',
  landmarks.ref = 'landmarks_ref.txt')
```

You can also call `digitizeImage()` with just image.file if you'd simply like to check the pixel coordinates of a few features in a photograph.

```
> digitizeImage(image.file = 'Object Images')
```

For more detailed instructions on how to use the StereoMorph Digitizing App, check out the `digitizeImage()` function in the StereoMorph package manual or click on the 'Help' link at the top right corner of the application window. The help file will open in a new browser window and has information on all the features available in the digitizing app, including keyboard shortcuts.



Opening the digitizing app help file.

The next step will cover how to reconstruct the digitized landmarks and curve points in 3D.

Reconstructing 2D Points and Curves into 3D

This section will demonstrate how to take the points and curves digitized in the previous section and reconstruct them into 3D. With direct linear transformation (DLT), 3D reconstruction is simple. All that is needed are the pixel coordinates of a point in at least two camera views and the DLT calibration coefficients. Steps will be demonstrated using files in the StereoMorph Tutorial folder. Point reconstruction will be covered first followed by curve reconstruction.

Point reconstruction

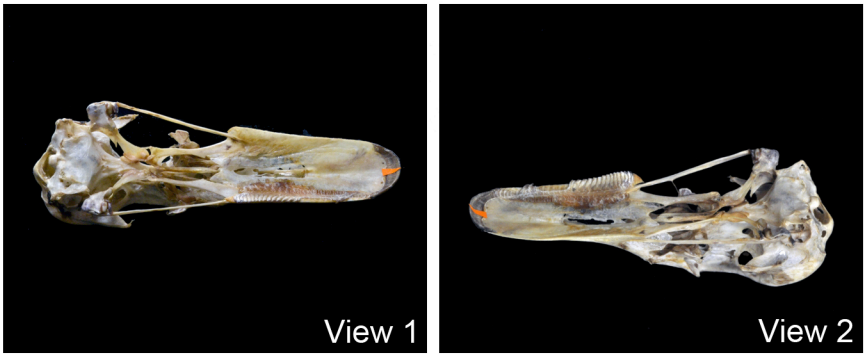
1. Load the StereoMorph package, if not already loaded, and the 'rgl' R package for viewing 3D points. Also ensure that the StereoMorph Tutorial folder is your current working directory.

```
> library(StereoMorph)
> library(rgl)
```

2. Read in the calibration coefficients. This should be a matrix of 11 rows (11 coefficients per view) and two columns (corresponding to the two camera views).

```
> cal.coeff <- as.matrix(read.table("cal_coeffs.txt"))
```

We'll begin by reconstructing points from the first aspect. Recall from the photographing step that this aspect provides views of the ventral (underside) of the skull.



Two camera views of the object in the first position, showing the ventral aspect of the skull.

3. Specify the location of the 2D landmark files (in pixel coordinates).

```
> landmarks_2d <- paste0("Landmarks 2D/obj_a1_v", 1:2, ".txt")
```

4. Read these landmarks into a matrix using `readLandmarksToMatrix()`.

```
> lm_matrix <- readLandmarksToMatrix(landmarks_2d, row.names=1)
```

The StereoMorph function `dltReconstruct()` accepts landmarks in matrix, array or list format (read using `readLandmarksToMatrix()`, `readLandmarksToArray()` and `readLandmarksToList()`, respectively). Here's what the first ten rows of the landmark matrix look like.

```
> lm_matrix
                                  x1   y1   x2   y2
basioccipital_proc_ant           800 1741 3562 1248
basipterygoid_proc_med_L         988 1775 3397 1271
basipterygoid_proc_med_R         988 1727 3392 1309
basipterygoid_proc_post_L        905 1805 3461 1239
basipterygoid_proc_post_R        917 1685 3471 1330
foramen_magnum_sup               264 1760 4059 1357
jugal_upperbeak_L               1879 2007 2579 1148
jugal_upperbeak_R               1883 1569 2588 1501
mand_condyle_quadrate_lat_L      799 2075 3522  881
mand_condyle_quadrate_lat_R      842 1304 3592 1467
...
```

The first two columns are the x- and y-coordinates in the first view (in pixels) and the third and fourth columns are the x- and y-coordinates in the second view. When multiple files are input into `readLandmarksToMatrix()`, the columns are added to the end of the matrix. If any of the files have landmarks not present in another file, these are included in the matrix but are `NA` for the views in which they are missing.

5. Call `dltReconstruct()`.

```
> dlt_recon <- dltReconstruct(cal.coeff, lm_matrix)
```

`dltReconstruct()` outputs a matrix of the 3D landmark coordinates (`dlt_recon$coor.3d`) in real-world units (here, mm) and an RMS error vector representing the reconstruction error (`dlt_recon$rmse`) in pixels. The RMS error is calculated by projecting the 3D coordinates back into each 2D image and comparing these projected pixel coordinates to the actual pixel coordinates in each view. It can be used to evaluate how closely points digitized in different images actually represent the same point. Errors of few pixels are expected given the difficulty in identifying the exact same point from two different views.

6. Use the summary() function to check the RMS error by landmark.

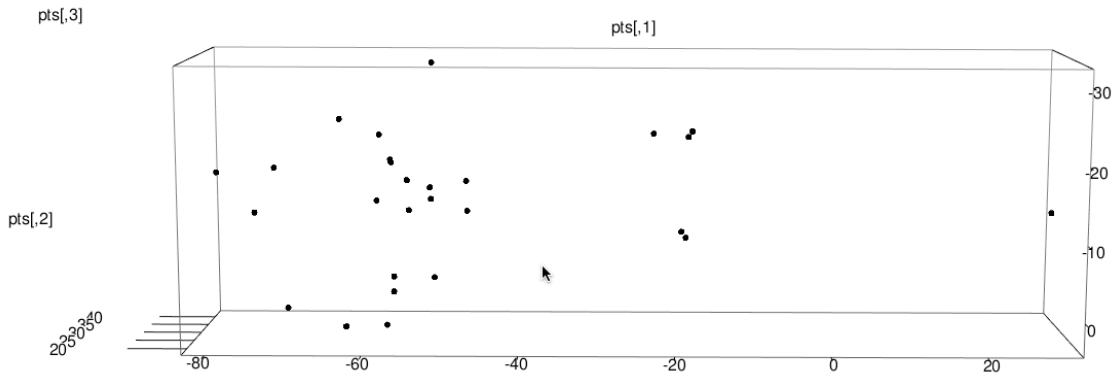
```
> summary(dlt_recon)

dltReconstruct Summary
      RMSE by Landmark:
            basioccipital_proc_ant: 6.714
            basipterygoid_proc_med_L: 3.180
            basipterygoid_proc_med_R: 5.642
...
```

The 3D landmarks can be plotted using the plot3d() function in the rgl package. We simply have to set the aspect so that the points are plotted with the correct relative lengths along each axis.

7. Plot the 3D landmarks using plot3d().

```
> pts <- na.omit(dlt_recon$coor.3d)
> r <- apply(pts, 2, 'max') - apply(pts, 2, 'min')
> plot3d(pts, aspect=c(r/r[3]), size=5)
```



3D landmarks from the first aspect visualized using plot3d() in the 'rgl' package.

8. Save the 3D landmarks to a text file.

```
> write.table(dlt_recon$coor.3d , file = "Landmarks 3D/obj_a1.txt",
  quote=F, sep="\t", col.names=F, row.names=T)
```

9. Repeat these steps to reconstruct landmarks from any remaining aspects.

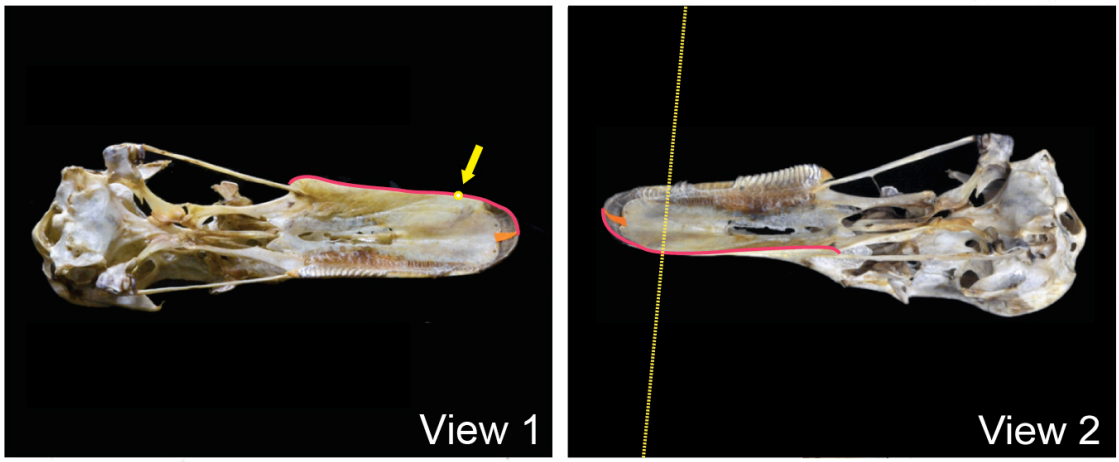
```
> for(i in 2:3){
    landmarks_2d <- paste0("Landmarks 2D/obj_a", i, "_v", 1:2, ".txt")
    lm_matrix <- readLandmarksToMatrix(landmarks_2d, row.names=1)
    dlt_recon <- dltReconstruct(cal.coeff, lm_matrix)
    write.table(dlt_recon$coor.3d ,
        file = paste0("Landmarks and curves 3D/obj_a", i, ".txt"),
        quote=F, sep="\t", col.names=F, row.names=T)
  }
```

The next section will demonstrate how to take reconstructed landmarks from several aspects, unify them into a single point set, reflect landmarks missing on one side and then align the whole set to the midline plane.

Curve Reconstruction

Curves can be reconstructed in the same way as points by breaking the curve down into a series of points. There is one complication to this, however. A point halfway along the curve in one camera view is not necessarily the same point as a point halfway along the curve in another camera view. This is due to the perspective effect of lenses. The depth of a 3D curve in one view dictates how it is projected into that image plane. Since the depth of a curve will differ depending on the perspective from which it is viewed, the same curve will be projected differently into different views.

It's not entirely hopeless; fortunately our calibration provides a way around this. In a stereo camera setup, a point in one camera view must fall along a line in another camera view.



Demonstration of epipolar geometry. The point, in camera view 1 (indicated by a yellow arrow), must fall along a line in camera view 2. This line is the point's epipolar line and can be used to identify corresponding points along two curves.

This line is called the epipolar line. The distance from this line to the point's position in the second camera view is the basis for epipolar error, which was mentioned when we tested the calibration accuracy. Corresponding points can be found between two curves by taking each point along a curve in one view and finding where its epipolar line intersects the curve in a second view (Yekutieli *et al.* 2007). This is done by the `dltMatchCurvePoints()` function in StereoMorph. The only downside is that when the epipolar line is parallel to a curve segment it becomes impossible to identify the exact corresponding point. `dltMatchCurvePoints()` has a threshold for this and can set these points as NA.

For point reconstruction, we read in the landmarks from the first aspect as a matrix. This time we'll read in both the landmarks and curve points from the first aspect.

1. Load the StereoMorph package, if not already loaded, and the 'rgl' R package for viewing 3D points. Also ensure that the StereoMorph Tutorial folder is your current working directory.

```
> library(StereoMorph)
> library(rgl)
```

2. Read in the calibration coefficients. This should be a matrix of 11 rows (11 coefficients per view) and two columns (corresponding to the two camera views).

```
> cal.coeff <- as.matrix(read.table("cal_coeffs.txt"))
```

3. Specify the file paths to the 2D curve points (both camera views) created by the StereoMorph Digitizing App.

```
> curves_2d <- paste0("Curve points 2D/obj_a1_v", 1:2, ".txt")
```

4. Since the number of 2D curve points for a particular curve likely differ between the two views (they are not corresponding points yet), read these curve points into a list rather than a matrix.

```
> cp_list <- readLandmarksToList(curves_2d, row.names=1)
```

The list has the following format,

```
$pterygoid_crest_R
$pterygoid_crest_R[[1]]
                          V2   V3
pterygoid_crest_R0001 1140 1672
pterygoid_crest_R0002 1140 1671
pterygoid_crest_R0003 1140 1670
...

$pterygoid_crest_R[[2]]
                          V2   V3
pterygoid_crest_R0001 3263 1358
pterygoid_crest_R0002 3264 1358
pterygoid_crest_R0003 3264 1359
...
```

where `[[1]]` and `[[2]]` correspond to the two camera views and the curve points (in pixel coordinates) are in the form of a matrix.

5. Input the curve point list, `cp_list`, into `dltMatchCurvePoints()` along with the already loaded calibration coefficients. This will find corresponding points between the *pterygoid_crest_R* and the *tomium_R* curves.

```
> dlt_mcp <- dltMatchCurvePoints(cp_list, cal.coeff,
  min.tangency.angle=0.3)
```

The `min.tangency.angle` is a threshold (in radians) for determining when points should be skipped because they lie in portions of the curve that are nearly parallel to the epipolar line. This ideal value for this parameter might vary from setup to setup.

6. Use the `summary()` function to print a summary of the epipolar errors in the curve point matching.

```
> summary(dlt_mcp)

dltMatchCurvePoints Summary
      Curve name: pterygoid_crest_R
            Reference point count: 327
            Start/end epipolar distances: 1.68 px, 0.75 px
            Portion of non-reference matched: 86.2 %
            Mean epipolar-curve point distance: 0.8 px +/- 1.69
            Max epipolar-curve point distance: 9.13 px
      Curve name: tomium_R
            Reference point count: 1446
            Start/end epipolar distances: 0.23 px, 3.2 px
            Portion of non-reference matched: 92.9 %
            Mean epipolar-curve point distance: 0.28 px +/- 0.27
            Max epipolar-curve point distance: 3.36 px
```

For both curves, over 85% of the points were able to be matched; that is, 85% of the points did not lie along the epipolar line. Points on the epipolar line in the non-reference view, assumed to be the point corresponding to points in the reference view, were on average less than a pixel from the digitized curve points (the 'Mean epipolar-curve point distance'). These two values give an indication that the point matching was reasonably accurate.

The key output of this function is `dlt_mcp$match.lm.list`, which is a list identical to the input except that all curve points have been replaced with matching points between the two views (the number of rows in each curve is now equal). If landmarks are input, these are passed through unmodified.

7. Convert the matching curve point list into a matrix.

```
> cp_matrix <- landmarkListToMatrix(dlt_mcp$match.lm.list)
```

8. Read in the 2D landmark coordinates from the same aspect, as done in 'Reconstructing Points'.

```
> landmarks_2d <- paste0("Landmarks 2D/obj_a1_v", 1:2, ".txt")
> lm_matrix <- readLandmarksToMatrix(landmarks_2d, row.names=1)
```

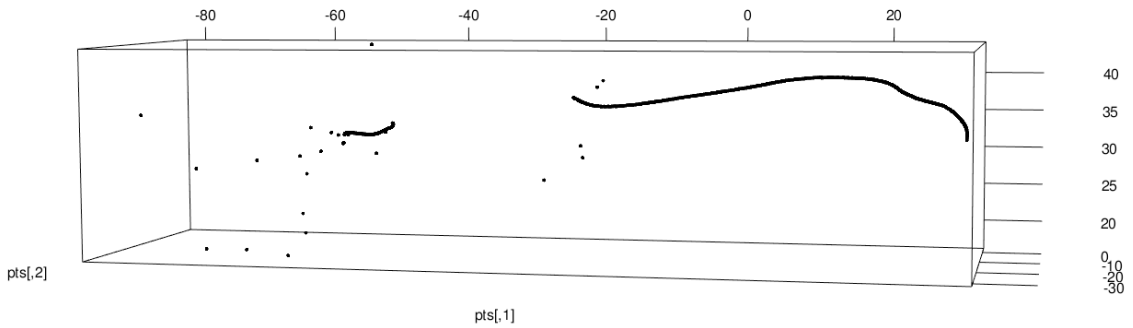
9. Call `dltReconstruct()`. For the 2D pixel coordinates, input both the matched 2D curve points and the 2D landmarks as a single matrix.

```
> dlt_recon <- dltReconstruct(cal.coeff, rbind(cp_matrix, lm_matrix))
```

This returns a matrix of the 3D landmark and curve points and their corresponding RMS errors.

10. Plot the 3D coordinates using 'rgl', setting the aspect to get the proper proportions.

```
> pts <- na.omit(dlt_recon$coor.3d)
> r <- apply(pts, 2, 'max') - apply(pts, 2, 'min')
> plot3d(pts, aspect=c(r/r[3]), size=1)
```



3D landmarks and curve points from the second aspect visualized using `plot3d()` in the 'rgl' package.

This returns many hundreds or thousands of points per curve (for example, the tomium curve has 1446 points). In most cases, that many points are not needed to adequately describe a curve. It would be useful to down sample the curve points to a number that sufficiently describes the complexity of the curve. This is also important for comparing curve points directly from multiple specimens; in this case the same number of curve points are needed for all specimens.

Polynomial curve-fitting or Bézier curve-fitting could be used to fit a function to the points and then select evenly spaced points along the function. However, since

the curves were digitized at single pixel spacing, the curve is already well-sampled (over 1000 points for just a few centimeters). In this case, the 3D coordinates can be used directly to generate evenly spaced points. This can be preferable over function fitting for curves not easily described by a mathematical function.

The StereoMorph function `pointsAtEvenSpacing()` takes a matrix of points, calculates the cumulative distance from start to end and then uses the cumulative distance and intermediate points to generate evenly spaced points between the start and end point. Linear interpolation is used between neighboring points, so the returned points will either coincide with the input points or fall on straight lines between consecutive points.

11. Convert the landmark matrix into a list so that each curve can be accessed separately.

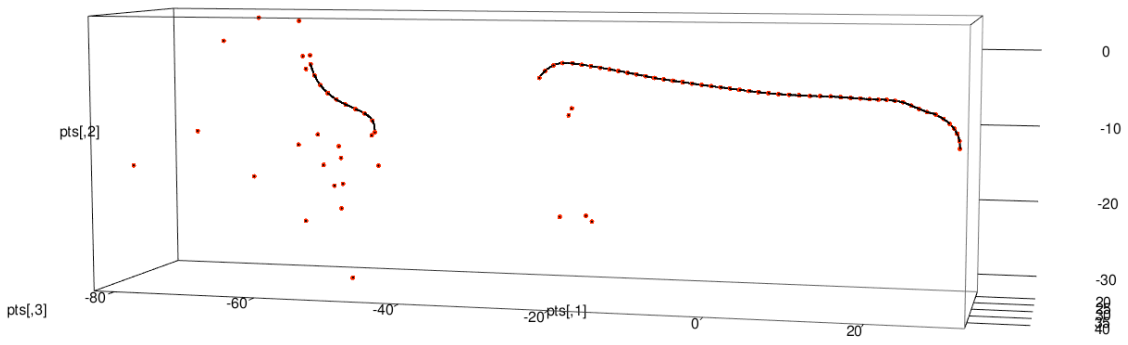
```
> lm.list <- landmarkMatrixToList(dlt_recon$coor.3d)
```

12. Call `pointsAtEvenSpacing()` for each curve and overwrite the existing entry in `lm.list`.

```
> lm.list$pterygoid_crest_R <- pointsAtEvenSpacing(
  x=lm.list$pterygoid_crest_R, n=10)
> lm.list$tomium_R <- pointsAtEvenSpacing(x=lm.list$tomium_R, n=50)
```

13. Plot the approximately evenly spaced points on top of the previous curve points.

```
> pts <- na.omit(dlt_recon$coor.3d)
> r <- apply(pts, 2, 'max') - apply(pts, 2, 'min')
> plot3d(pts, aspect=c(r/r[3]), size=1)
> lm.matrix <- landmarkListToMatrix(lm.list)
> plot3d(na.omit(lm.matrix), size=4, col='red', add=TRUE)
```



Approximately evenly spaced points along the two curves (in red) superimposed on the original full set of curve points, visualized using `plot3d()` in the 'rgl' package.

14. Convert the landmark list back into a matrix.

```
> lm.matrix <- landmarkListToMatrix(lm.list)
```

15. Write the landmarks and curve points to a text file.

```
> write.table(lm.matrix, file="Landmarks and curves 3D/obj_a1.txt",
  quote=F, sep="\t", col.names=F, row.names=T)
```

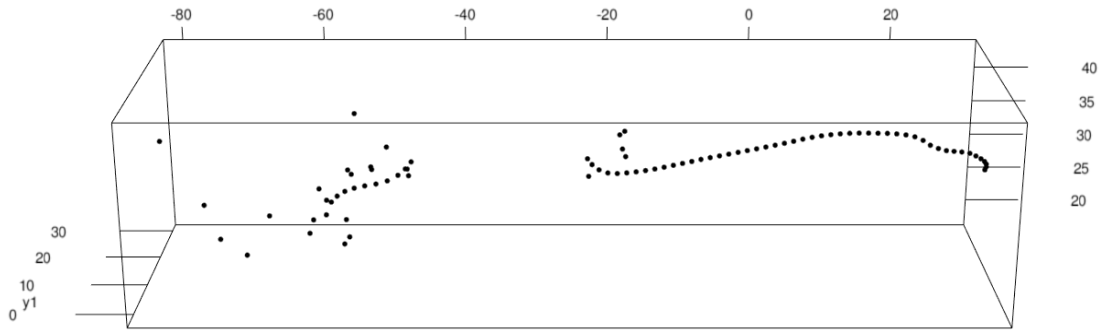
The next section will demonstrate how to take reconstructed landmarks and curve points from several aspects, unify them into a single point set, reflect landmarks missing on one side and then align the whole set to the midline plane.

Unifying, Reflecting and Aligning

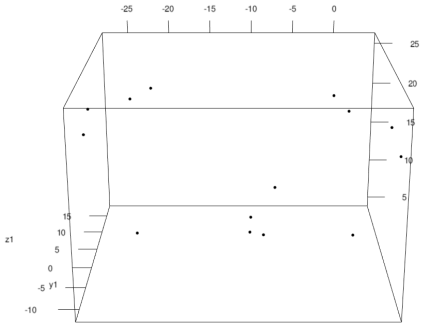
This section will demonstrate how to unify 3D landmarks and curve points from several aspects into a single point set, reflect landmarks missing on one side and align the whole set to the midline plane.

Unifying landmarks

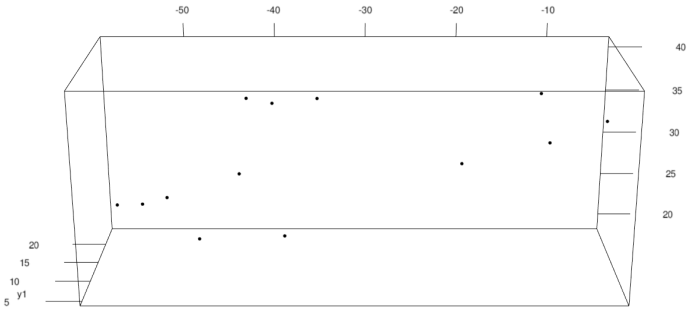
If the same object has been photographed in more than one position (aspects), the landmarks and curve points collected from each aspect will be in different coordinate systems.



Aspect 1



Aspect 2



Aspect 3

Landmarks and curve points from three different aspects of an object.

In order to combine these three aspects, they must be unified based on shared points. This can be done in StereoMorph using the `unifyLandmarks()` function.

1. Load the StereoMorph package, if not already loaded, and the 'rgl' R package for viewing 3D points. Also ensure that the StereoMorph Tutorial folder is your current working directory.

```
> library(StereoMorph)
> library(rgl)
```

2. Specify the file paths of each aspect of 3D landmarks and curves. For demonstration, the landmarks and curve points reconstructed in the previous section will be used.

```
> landmarks_3d <- paste0("Landmarks and curves 3D/obj_a", 1:3, ".txt")
```

3. Read each of these into an array using the `readLandmarksToArray()` function.

```
> lm.array <- readLandmarksToArray(landmarks_3d, row.names=1)
```

4. Call `unifyLandmarks()` to unify all the aspects into a single point set.

```
> unify_lm <- unifyLandmarks(lm.array, min.common=5)
```

The `unifyLandmarks()` function begins by choosing two point sets and aligning them with each other based on three or more shared points. Then, any additional point sets are unified with this combined point set, one-by-one, saving each unified point set at each step as the new combined point set.

`unifyLandmarks()` finds an ideal sequence based on the error of unification (how well the common points line up with each other). By setting `min.common` to 5, `unifyLandmarks()` will only align two aspects if they share at least five points. This does not mean that all point sets have to share the same five points; once two point sets are unified, any points shared between those two point sets and the next point set will be used in the alignment. Thus, the number of shared points used for each unification will usually depend on the order in which the points are unified. While it's possible to unify points based on only three points this is likely to cause poor alignments, especially if the shared points happen to be nearly collinear.

5. Use the `summary()` function to see the unification sequence and errors.

```
> summary(unify_lm)

unifyLandmarks Summary
Unification sequence: 1, 2, 3
Unification RMSE:
      0.7412698
      0.8801677
```

The first part shows the sequence in which the point sets were unified followed by the root-mean-square error (here, in millimeters) for each unification step.

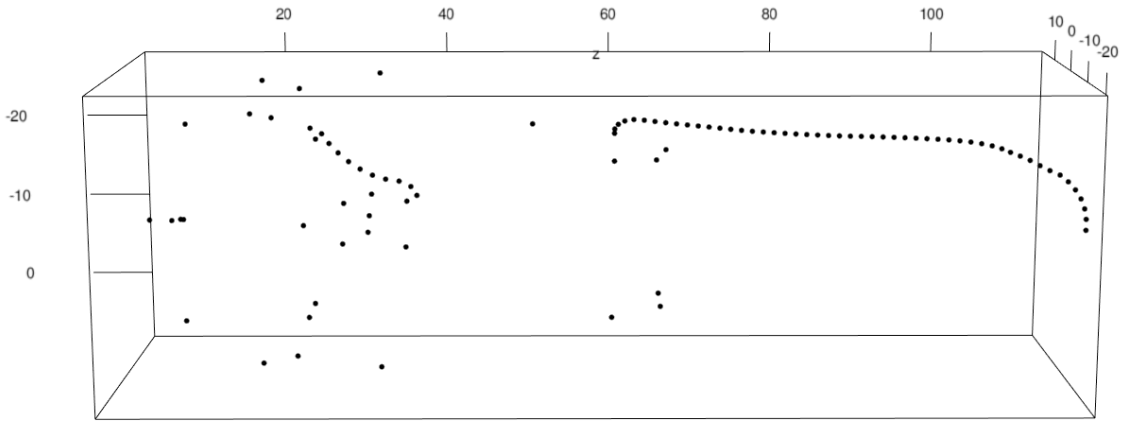
```
Unification landmark errors by sequence:
      [[1]]
            basipterygoid_proc_post_R        0.160657
            jugal_upperbeak_R                1.046571
            opisthotic_process_R             0.8580522
            ...

      [[2]]
            foramen_magnum_sup            0.7183007
            mand_condyle_quadrate_lat_L   0.6221617
            mand_condyle_quadrate_lat_R   0.2217136
            ...
```

The second part shows the unification errors by landmark for each sequence (indicated by the number in double brackets). Both indicate that the unification errors are fairly low. The major source of error here is the difficulty of finding the exact same points from different views of an object.

6. Plot the points using `plot3d()` from the 'rgl' package.

```
> pts <- na.omit(unify_lm$lm.matrix)
> r <- apply(pts, 2, 'max') - apply(pts, 2, 'min')
> plot3d(pts, aspect=c(r/r[3]), size=3)
```



All landmarks and curve points unified into a single point set.

7. Save the unified landmarks to a text file.

```
> write.table(unify_lm$lm.matrix, file="Landmarks 3D unified/obj.txt",
  quote=F, sep="\t", col.names=F, row.names=T)
```

Reflecting missing landmarks

When collecting landmarks from objects with left/right (bilateral) symmetry, it's often not possible to collect every landmark on both sides of the object. In this case, the plane of bilateral symmetry can be used to reflect landmarks only present on one side across the midline, creating a set of landmarks complete on both sides (Klingenberg, Barluenga & Meyer 2002). This can be done using the `reflectMissingLandmarks()` function in StereoMorph.

1. Load the StereoMorph package, if not already loaded, and the 'rgl' R package for viewing 3D points. Also ensure that the StereoMorph Tutorial folder is your current working directory.

```
> library(StereoMorph)
> library(rgl)
```

2. Specify the file paths of a 3D point set of landmarks and/or curve points.

```
> lm_unified <- paste0("Landmarks 3D unified/obj.txt")
```

3. Import the landmarks as a matrix.

```
> lm.matrix <- readLandmarksToMatrix(lm_unified, row.names=1)
```

4. Call `reflectMissingLandmarks()`.

```
> reflect <- reflectMissingLandmarks(lm.matrix, average=TRUE)
```

Or, if you have just completed the unification step, you can use `unify_lm$lm.matrix`.

```
> reflect <- reflectMissingLandmarks(unify_lm$lm.matrix, average=TRUE)
```

Setting average to TRUE will reflect the missing landmarks across the midline and then average the left and right sides (so that they are mirror images).

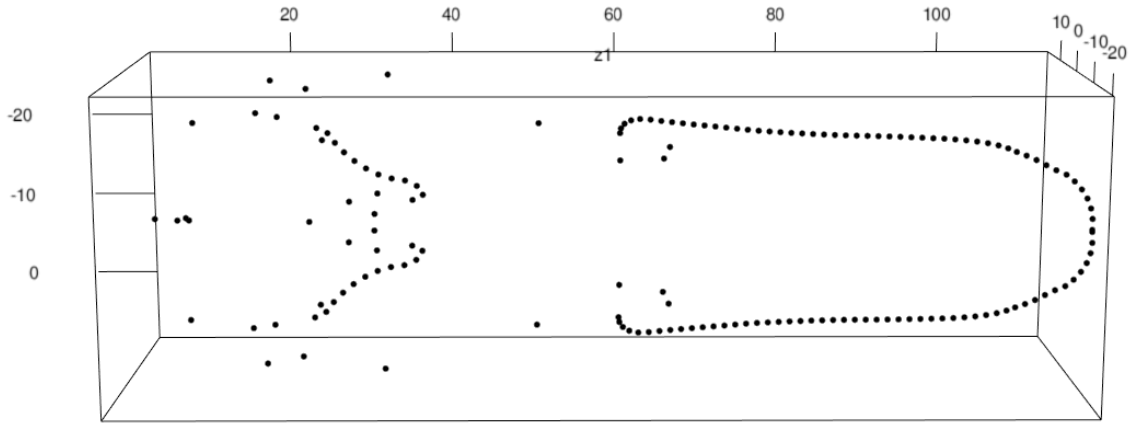
5. Print a summary of the bilateral errors.

```
> summary(reflect)
```

The bilateral errors are calculated by measuring the distance between a point present on both sides already and its contralateral point after reflection (and before averaging).

6. Plot the points using `plot3d()` from the 'rgl' package.

```
> pts <- na.omit(reflect$lm.matrix)
> r <- apply(pts, 2, 'max') - apply(pts, 2, 'min')
> plot3d(pts, aspect=c(r/r[3]), size=3)
```



Unified landmarks after reflecting landmarks missing on one side across the midline.

7. Save the reflected landmarks to a text file.

```
> write.table(reflect$lm.matrix,
  file="Landmarks 3D reflected/obj.txt", quote=F, sep="\t",
  col.names=F, row.names=T)
```

Aligning landmarks to the midline

The 3D landmark and curve points resulting from the previous steps of reconstruction and unification will be arbitrarily positioned and oriented in 3D space. For visualization purposes it's often desirable to align a set of landmarks to the midline plane so that multiple objects can be viewed in a consistent orientation.

This can be done with the StereoMorph function `alignLandmarksToMidline()`. This function translates and rotates the points so that the midline points are aligned with the xy-plane. If bilateral landmarks were averaged in the reflection step, the midline points will lie exactly in the midline plane (i.e. they will have z-values of zero). Aligning landmarks to the midline plane does not change the scaling of the landmarks or the position of any landmarks relative to one another.

1. Load the StereoMorph package, if not already loaded, and the 'rgl' R package for viewing 3D points. Also ensure that the StereoMorph Tutorial folder is your current working directory.

```
> library(StereoMorph)
> library(rgl)
```

2. Specify the file paths of a 3D point set of landmarks and/or curve points.

```
> lm_reflected <- paste0("Landmarks 3D reflected/obj.txt")
```

3. Import the landmarks as a matrix.

```
> lm.matrix <- readLandmarksToMatrix(lm_reflected, row.names=1)
```

4. Call `reflectMissingLandmarks()`.

```
> align <- alignLandmarksToMidline(lm.matrix)
```

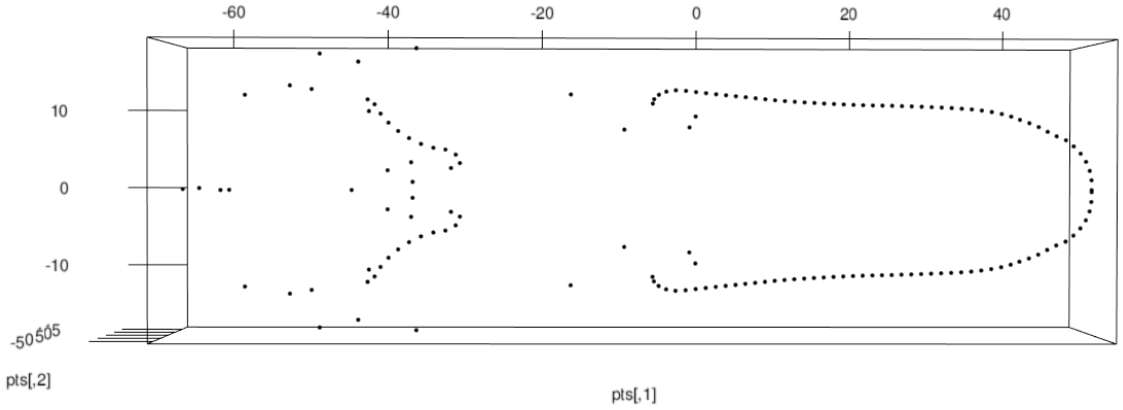
Or, if you have just completed the unification step, you can use `unify_lm$lm.matrix`.

```
> align <- alignLandmarksToMidline(reflect$lm.matrix)
```

Calling `summary()` will print the midline alignment errors (the distances between the midline points and the estimated midline plane). However, since the bilateral landmarks were averaged in the reflection step, all of the midline points were already aligned to the midline plane. Thus, all of the midline alignment errors are zero.

5. Plot the points using `plot3d()` from the 'rgl' package.

```
> pts <- na.omit(align$lm.matrix)
> r <- apply(pts, 2, 'max') - apply(pts, 2, 'min')
> plot3d(pts, aspect=c(r/r[3]), size=3)
```



Unified landmarks after aligning the landmarks to the midline.

Unified landmarks after aligning the landmarks to the midline.

The landmarks are now aligned along the midline and landmarks on opposite sides have the same coordinates except an equal and opposite value along the z-axis.

```
> align$lm.matrix['quadrate_jugal_L', ]
[1] -45.431501  -6.010435  16.372618
> align$lm.matrix['quadrate_jugal_R', ]
[1] -45.431501  -6.010435 -16.372618
```

6. Save the aligned landmarks to a text file.

```
> write.table(align$lm.matrix,
  file="Landmarks 3D aligned/obj.txt", quote=F, sep="\t",
  col.names=F, row.names=T)
```

Using StereoMorph to Collect 2D Shape Data

Although StereoMorph was created for the collection of 3D shape data, it provides several new tools that can also be used for the collection of 2D shape data. The StereoMorph Digitizing Application provides an easy-to-use and cross-platform compatible interface for the collection of landmarks and curves from photographs. Additionally, the automated checkerboard detection can be used to automatically scale landmarks and curves to real-world coordinates (rather than manually digitizing a ruler, for instance). This section demonstrates how to use StereoMorph to collect 2D shape data from photographs.

1. Load the StereoMorph package, if not already loaded.

```
> library(StereoMorph)
```

2. There is a folder named '2D Morphometrics' in the StereoMorph Tutorial folder. It contains the demo files that will be used in this section. Make this folder your current working directory. If your current working directory is StereoMorph Tutorial folder, you can make '2D Morphometrics' the current directory with the following command.

```
> setwd('2D Morphometrics')
```

3. Create and print a checkerboard pattern on to cardstock paper as described in "Creating a Checkerboard Pattern". It's not necessary to glue the checkerboard to a piece of plexiglass since it will simply lie flat in the photograph. For this demo, I made a 5 x 3 checkerboard with squares 567 pixels wide.

```
> drawCheckerboard(nx=5, ny=3, square.size=567, filename='Checkerboard
  5x3 (567px).JPG')
```

When printed at 5% scaling, the squares will be approximately 10 mm wide.

$$567 \text{ px} \quad \text{x} \quad \frac{1}{72 \text{ px/in}} \quad \text{x} \quad 0.05 \quad \text{x} \quad \frac{25.4 \text{ mm}}{1 \text{ in}} \quad = \quad 10.0013 \text{ mm}$$

4. Measure the checkerboard square size in real-world by following the steps in the section "Measuring Checkerboard Square Size".

5. Take a photograph of the object you want to digitize with the checkerboard in the image.

An image of an object to be digitized and a checkerboard pattern to automatically scale the 2D shape data.

Just as when measuring the checkerboard square size, it is important that the camera lens does not introduce significant distortion and that the checkerboard pattern, and the plane from which you are collecting data, is approximately coplanar with the image plane. If the checkerboard or object is at an angle relative to the image plane, some of the points being digitized will be closer to the camera than others, resulting in a difference in size on the image plane where there is not actually a difference in size (this is the perspective effect).

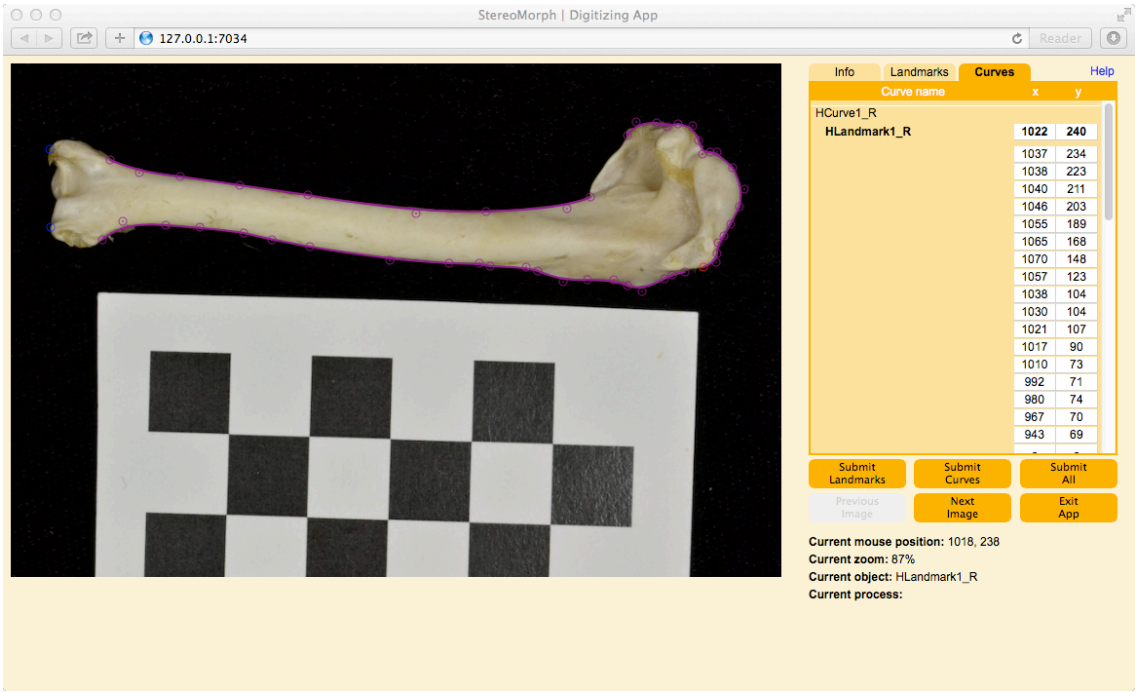
For the same reason, the checkerboard should be in the same plane as the plane from which points are being digitized. If the object has some depth to it, raise the checkerboard so that it is approximately coplanar with the points you'll be digitizing.

6. Upload the photographs of each object with a checkerboard pattern. For demonstration, I'll use the image 'RHumerus.JPG' in the 'Object Images' folder of the '2D Morphometrics'.

7. To digitize landmarks and curves, open the uploaded image(s) with the StereoMorph Digitizing Application and follow the steps in the section “Digitizing Photographs”.

```
> digitizeImage(image.file = 'Object Images',
  landmarks.file = 'Landmarks px',
  control.points.file = 'Control points',
  curve.points.file = 'Curves px',
  landmarks.ref = 'landmarks_ref.txt',
  curves.ref = 'curves_ref.txt')
```

Several landmarks and curves have been digitized and saved to the folders ‘Landmarks pixels’ and ‘Curves pixels’.



An image of an object from which 2D data will be collected, opened in the StereoMorph Digitizing Application. Several landmarks (in blue) and curves (in purple) have been digitized. See “Digitizing Photographs” for details.

8. To scale the landmarks and curves to real-world units (e.g. millimeters), begin by specifying the number of *internal* corners in the checkerboard pattern. Note that this is not the number of black squares but rather the number of points where the corners of the black squares contact each other. The choice of which is nx and which is ny is arbitrary.

```
> nx <- 5
> ny <- 3
```

9. Call `findCheckerboardCorners()` to find the internal corners of the checkerboard.

```
> find_corners <- findCheckerboardCorners(image.file='Object images',
  nx=nx, ny=ny, corner.file='Corners', verify.file='Corners verify')
```

Since there are three images in the 'Object images' folder, the function will find corners in all three images and save the corners to text files with the same name as the images in the folder 'Corners'. The demo files run particularly fast because they are small images. For details on using this function, see the "Auto-detecting Checkerboard Corners" section.

10. Read in the corner pixel coordinates from the first file as a matrix.

```
> corners <- as.matrix(read.table('Corners/RHumerus.txt'))
```

11. Find the checkerboard square size in pixels.

```
> measure <- measureCheckerboardSize(corners, nx=nx)
```

The square size is found in the exact same way as described in the section "Measuring Checkerboard Square Size" only in this case we are not including ruler points. The estimated square size in pixels is returned as `measure$square.size.px`.

```
> square_size_px  <- measure$square.size.px
```

12. Specify the square size in real-world units. For the demo images, the square size of the checkerboard is 10.001 mm.

```
> square_size_mm <- 10.001
```

To convert any points in pixels to millimeters, simply multiply the pixel coordinates by the square size in millimeters divided by the square size in pixels.

```
> scale_px2mm <- square_size_mm / square_size_px
```

13. Read in the landmarks and curve points as matrices.

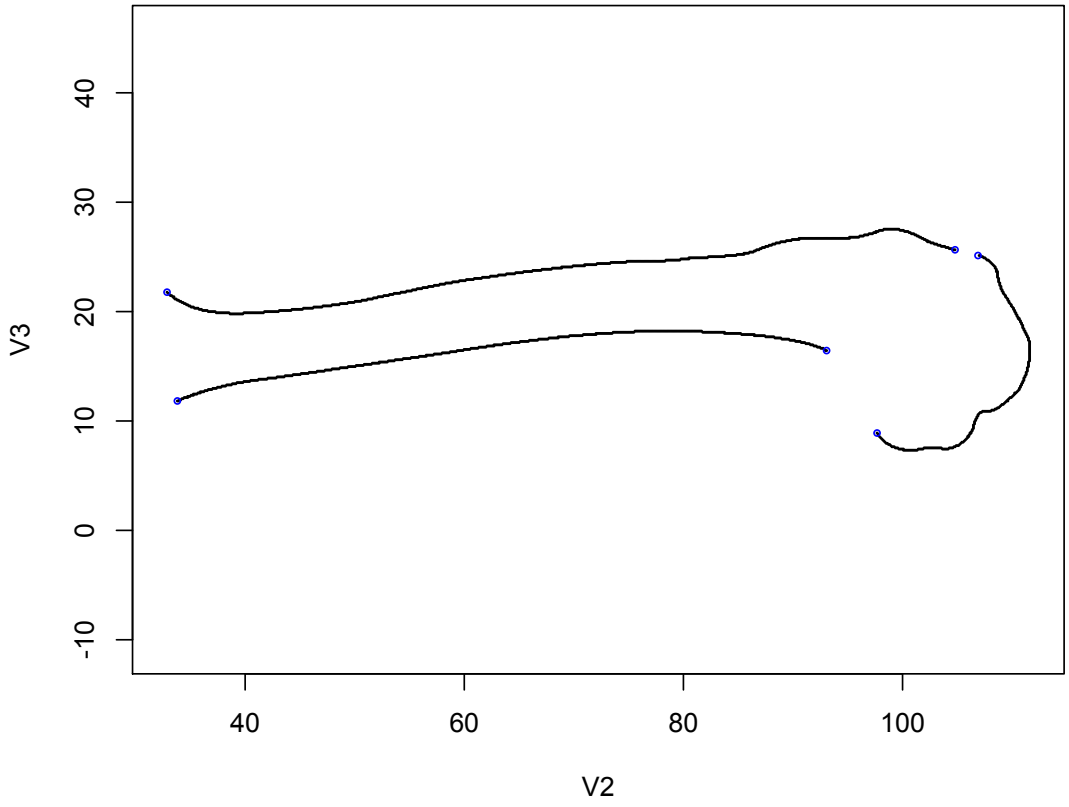
```
> landmarks_px <- as.matrix(read.table('Landmarks px/RHumerus.txt',
  row.names=1))
> curves_px <- as.matrix(read.table('Curves px/RHumerus.txt',
  row.names=1))
```

14. Convert the landmarks and curves from pixel coordinates to millimeters using the conversion factor saved to `scale_px2mm` in step 12.

```
> landmarks_mm <- landmarks_px * scale_px2mm
> curves_mm <- curves_px * scale_px2mm
```

15. To visualize the scaled landmarks and curve points, use `plot()` and `points()`.

```
> plot(curves_mm, cex=0.1, asp=1)
> points(landmarks_mm, cex=0.5, col='blue')
```



The curve points (black) and landmarks (blue) scaled to millimeters and visualized using `plot()`.

16. Save the landmarks and curve points to text files as matrices with row names.

```
> write.table(landmarks_mm , file = "Landmarks mm/RHumerus.txt",
  quote=F, sep="\t", col.names=F, row.names=T)
> write.table(curves_mm , file = "Curves mm/RHumerus.txt", quote=F,
  sep="\t", col.names=F, row.names=T)
```

These steps can be repeated over several images. Once you have digitized landmarks and curves from a series of images and assuming the folder structure in the tutorial folder '2D Morphometrics', the following will perform the same steps as above for all of the digitized files.

```
> nx <- 5
> ny <- 3
> find_corners <- findCheckerboardCorners(image.file='Object images',
  nx=nx, ny=ny, corner.file='Corners', verify.file='Corners verify')
> filenames <- gsub('.jpg|.jpeg', '.txt', list.files('Object images'),
  ignore.case=TRUE)
> square_size_mm <- 10.001
> for(filename in filenames){
    corners <- as.matrix(read.table(paste0('Corners/', filename)))
    measure <- measureCheckerboardSize(corners, nx=nx)
    scale_px2mm <- square_size_mm / measure$square.size.px
    landmarks_px <- as.matrix(read.table(paste0('Landmarks px/',
      filename), row.names=1))
    curves_px <- as.matrix(read.table(paste0('Curves px/', filename),
      row.names=1))
    landmarks_mm <- landmarks_px * scale_px2mm
    curves_mm <- curves_px * scale_px2mm
    write.table(landmarks_mm, file = paste0('Landmarks mm/',filename),
      quote=F, sep='\t', col.names=F, row.names=T)
    write.table(curves_mm, file = paste0('Curves mm/', filename),
      quote=F, sep='\t', col.names=F, row.names=T)
  }
```