Fitting a Template to Striate Cortex

Noah C. Benson, Omar H. Butt, Ritobrato Datta, Petya D. Radoeva, David H. Brainard, Geoffrey K. Aguirre

Supplemental Material for the paper "The retinotopic organization of striate cortex is well predicted by surface topology"

Introduction

The purpose of this file

This file is an interactive Mathematica notebook designed to walk the curious reader through all of the steps required to generate and export a V1 template, as is documented in the paper to which this file is attached as supplemental materials. If one correctly follows the instructions in the Introduction/Parameters subsection (below), it should be possible to evaluate every subsequent command without error.

If you are using this file as a tutorial or explanation, you can set the following variable to True; this will enable automatic figure generation. If this is left as False, then figures made along the way are not displayed (only saved to disk).

```
$ShowFigures = True;
```

A note regarding lazy-programming style

Virtually all of the calculations in this file use, directly or indirectly, some form of lazy programming. This is to say that, when a variable (such as the polar-angle data for a single subject) is declared, the data are not actually loaded and processed. Rather, the variable to hold the data is told how to load and process itself so that the first time it is referenced, the loading and processing takes place invisibly, and the data gets cached in the variable. Because of how *Mathematica*'s evaluation engine works, this is entirely seamless. If the variable x is initialized with lazy programming rather than actual data, you can still use it exactly as a normal variable:
x := (x = SomeSlowFunction[]); (* This does not load or calculate anything *)
x (* This forces x to run SomeSlowFunction[] and be assigned the result *)

What this means in practice is that you can evaluate the entire introduction and declara-

tion section of this notebook in just a few seconds because these parts of the notebook only tell the rest of the notebook how to load and process the data rather than actually loading it. If you evaluate the introduction then create a figure by evaluating one of the figure code blocks, the fugure-block will seem to take unnecessarily long, but this is only because it is loading and caching all of the data it needs for that particular figure. Subsequent evaluations of the same block will happen much more quickly. The main utility of this strategy is that if, for example, you only want to examine the eccentricity data plots, you don't have to load all of the anatomical data or find fits for the polar angle data just to make an eccentricity figure.

Dependencies

This section merely includes Mathematica dependencies that we may need further down the road in this file.

```
Needs["PlotLegends`"];
Needs["ErrorBarPlots`"];
Needs["MultivariateStatistics`"];
Needs["ComputationalGeometry`"];
```

This cell (below) is included only because of a small bug in some versions of Mathematica, which can cause infinite recursion if a FittedModel is loaded into the notebook rather than constructed from a function like NonlinearModelFit[]. This seems to prevent the error.

```
Block[{q, x}, NonlinearModelFit[{1, 2, 3, 5, 7, 9, 13, 17}, x^q, q, x]];
```

Local Functions and Parameters

Local functions include any function that is used during the execution of this file. This does not include λ-functions, but should include anything called frequently in this file. Additionally, parameters that will be used by all datasets should be defined in this section with a description of how they are used.

Filesystem Information

The template base directory is used for changing the directory from which you read all of your data. You don't need to use this, but if you organize your data the way we did, it will make it easier to fix/modify the various parameters in this file.
Note that the default value of this variable is a website; this means that *Mathematica* will pull our data from our public site for analysis. You may have to accept the certificate of the UPenn CFN (Center for Neuroscience) to do this. Alternately, you may download the zip file of the data directory and unzip it anywhere; this notebook

should work with a directory or a URL.
The zip file can be found here: https://cfn.upenn.edu/aguirreg/public/V1/BensonNC_2012_CurrBiol.tar.gz

```
$TemplateDirectory = "http://cfn.upenn.edu/aguirreg/public/V1/data";
```

When creating figures, we may want to export them to a specific directory. If you don't want to do this, just set this to None; otherwise, make it a directory.

```
$FigureExportDirectory = "~/Documents/Figures";
```

This directory is the target for any vtk file exported (there are routines for exporting VTK files with, e.g., the template mapped onto the cortical surface)

```
$VTKExportDirectory = "~/Documents/VTKs";
```

This shouldn't be edited; it's a convenient way for a figure to determine it's export filename.

```
FigureExport[filename_String, fig_,
   type_String, opts : OptionsPattern[Export]] := Which[
   $FigureExportDirectory === None, Undefined,
   ! StringQ[$FigureExportDirectory],
   (Print["$FiguresExportDirectory is not a string!"]; Abort[]),
   ! DirectoryQ[$FigureExportDirectory],
   (Print["$FiguresExportDirectory is not a directory!"]; Abort[]),
   True, Export[$FigureExportDirectory <> "/" <> filename, fig, type, opts]];
FigureExport[filename_String, fig_, opts : OptionsPattern[Export]] := Which[
   $FigureExportDirectory === None, Undefined,
   ! StringQ[$FigureExportDirectory],
   (Print["$FiguresExportDirectory is not a string!"]; Abort[]),
   ! DirectoryQ[$FigureExportDirectory],
   (Print["$FiguresExportDirectory is not a directory!"]; Abort[]),
   True, Export[$FigureExportDirectory <> "/" <> filename, fig, opts]];
```

- **Error Handling**

This function prints an error message and generates an Abort[] when things don't add up. It can be used to check progress for sanity. The ifTrue and errorMessage arguments are evalueated only when required.

```
SanityCheck[value_, test_, errorMessage_] := If[test[value],
   value,
   (Print[errorMessage]; Abort[])];
```

- **Color Scales**

This function provides the default color scale for use in figures in this notebook. Our default color scale for use in the paper was Blend[{Cyan,Blue,Purple,Red,Yellow},#]& (for datasets with >10° of data, the colors Black,Grey are appended to the blended color list). This color

scale is different, but can be edited to any color schema.

```
$ColorScaleColors = {Blue, Cyan, Green, Yellow, Red};
$ColorScaleFunction = (Blend[$ColorScaleColors, #] &);
```

- **Convert to Spherical Coordinates**

These functions will convert 3D coordinates into spherical coordinates assuming that the data lies on a sphere and we don't need to r/ρ coordinate (as we generally assume in this file with spherical brains). These also automatically handle lists of coord->value rules since we deal with them frequently while reading in VTK files.

```
CartesianToSpherical[dat : {{_, _, _} ..}] := Map[
   {If[#[[1]] == 0. && #[[2]] == 0., 0, ArcTan[#[[1]], #[[2]]]],
     ArcSin[#[[3]] / Norm[#]]} &,
   dat];
CartesianToSpherical[dat : {Rule[{_, _, _}, _] ..}] := Map[
   Rule[
     {If[#[[1, 1]] == 0. && #[[1, 2]] == 0., 0, ArcTan[#[[1, 1]], #[[1, 2]]]],
      ArcSin[#[[1, 3]] / Norm[#[[1]]]]},
     #[[2]]] &,
   dat];
CartesianToSpherical[dat : {_, _, _}] := List[
   If[dat[[1]] == 0. && dat[[2]] == 0., 0, ArcTan[dat[[1]], dat[[2]]]],
   ArcSin[dat[[3]] / Norm[dat]]];
CartesianToSpherical[dat : Rule[{_, _, _}, _]] := Rule[
   {If[dat[[1, 1]] == 0. && dat[[1, 2]] == 0., 0, ArcTan[dat[[1, 1]], dat[[1, 2]]]],
    ArcSin[dat[[1, 3]] / Norm[dat]]},
   dat[[2]]];
CartesianToSpherical[dat : {Rule[{_, _, _}, _] ..}] := Map[
   Rule[
     {If[#[[1, 1]] == 0. && #[[1, 2]] == 0., 0, ArcTan[#[[1, 1]], #[[1, 2]]]],
      ArcSin[#[[1, 3]] / Norm[#[[1]]]]},
     #[[2]]] &,
   dat];
```

- **Reading VTK Files**

Mathematica natively reads VTK file vertex and polygon data, but does not read the field data. This function will do this.

```
ReadVTK[filename_String, OptionsPattern[]] :=
  Module[
   {fl = StringToStream[
      (* we do all this instead of a normal
       stream because only import correctly handles URLs *)
      StringJoin @@ Import[
        filename,
        "Character8"]], (* The stream for this particular file *)
    p, n, m, dat, (* used below as temporaries *)
```

```
    toSpherical = SanityCheck[OptionValue["ConvertToSpherical"],
      # == True || # == False &,
      "ConvertToSpherical option to ReadVTK must be True or False"],
    unmask = SanityCheck[OptionValue["UnmaskData"],
      # == True || # == False &,
      "UnmaskData option to ReadVTK must be True or False"]},
   dat = Reap[
      For[p = Find[fl, "FIELD"], ! (p === EndOfFile), p = Find[fl, "Field"],
       {p, n, m, p} = Read[fl, {Word, Number, Number, Word}];
       Sow[Partition[ReadList[fl, Number, n * m], n]]];
      Close[fl]][[2, 1]];
   dat = Thread[
     Import[filename, "VertexData"] → Map[Flatten, Transpose[dat]]];
   (* At this point, dat is the data in the form of a list of {x,y,z}→
    value rules. We may want to clean these up according to
     the options: if the "ConvertToSpherical" option is true,
   we convert everything into spherical coordinates.  If the
    "UnmaskData" option is true, we filter out 0-valued
     data members. *)
   Which[
    (* We want to trim out those elements of dat that have zero values
     and convert the coordinates to spherical coordinates*)
    unmask && toSpherical, Flatten[
     Reap[
       Replace[dat, ({x_, y_, z_} → {value_}) :↦
         If[value ≠ 0., Sow[CartesianToSpherical[{x, y, z}]]], {1}]
      ][[2]],
     1],
    (* We want to unmask the data
     but not convert it into spherical coordinates *)
    unmask, Flatten[
     Reap[
       Replace[dat, (coord_ → {value_}) :↦ If[value ≠ 0., Sow[coord]], {1}]
      ][[2]],
     1],
    (* We want to convert to spherical
     coordinates but keep the values and not filter *)
    toSpherical, CartesianToSpherical[dat],
    (* or we just leave dat alone and return it *)
    True, dat]];
ReadVTK[__] :=
  (Print["Error: ReadVTK[] must be called with a filename"]; Abort[]);
(* These options allow ReadVTK to unmask data and/or
 convert it to spherical coordinates *)
Options[ReadVTK] = {"ConvertToSpherical" → False, "UnmaskData" → False};
```

- **FSAverage Geometric Parameters**

Freesurfer's FSAverage spherical brain has some quirks when it comes to the anatomical shape of V1; we correct for some of this here using a shear transformation. Note

that you won't want to use this (e.g., in your data-loading functions) if you are not using the FSAverage sphere.

```
$FSAverageShearMatrix = {{1, 0.65}, {0, 1}};
```

- **V1 and Expanded V1**

There must be a definition of V1 in order for this file to work properly. V1 should be an n x 2 matrix in which each row is the coordinate of a vertex in V1.

```
V1Filename = $TemplateDirectory <> "/metadata/V1-predict.mh.vtk";
V1Loader[] :=
  ReadVTK[V1Filename, "UnmaskData" → True, "ConvertToSpherical" → False];
```

The Expanded V1 is similar to V1 but should include points that surround V1; we use the OP Hinds et al. probabilistic V1 definition for this. It is assumed that V1Expanded is loaded by the same method V1, but this does not have to be true.

```
V1ExpandedFilename = $TemplateDirectory <> "/metadata/V1-prob.mh.vtk";
V1ExpandedLoader[] :=
  ReadVTK[V1ExpandedFilename, "UnmaskData" → True, "ConvertToSpherical" → False];
```

We also require anatomical information about V1; the following functions should load anatomical data for the equivalent regions of V1 and V1Extended.

```
AnatomyFilename =
  $TemplateDirectory <> "/anatomy/fs-average-template-curvature.vtk";
AnatomyLoader[] := ReadVTK[AnatomyFilename,
   "UnmaskData" → False, "ConvertToSpherical" → False];
```

The definitions for V1, V1Expanded, and Anatomy below all look complex, but they are basically just ways to make sure that their values are saved in variables such as V1Original before alignment with the V1 elliptical coordinate system. The data will automatically align themselves once the coordinate system is established.

```
V1Original := (V1Original = V1Loader[]; V1Original);
V1 := (V1 = V1Transform[V1Original]);

V1ExpandedOriginal :=
  (V1ExpandedOriginal = V1ExpandedLoader[]; V1ExpandedOriginal);
V1Expanded := (V1Expanded = V1Transform[V1ExpandedOriginal]);

AnatomyOriginal :=
  (AnatomyOriginal = Map[Join @@ # &, AnatomyLoader[]]; AnatomyOriginal);
Anatomy := (Anatomy = V1Transform[AnatomyOriginal]);
```

We also want to establish a region that is like the expanded V1 but with a smaller radius; ie, something that is still elliptical like V1 but dialated

```
V1Dialated := Module[
   {a = Block[{x, y}, FindRoot[V1EllipseEquation /. y → 0, {x, 1.0}][[1, 2]]],
    b = Block[{x, y}, FindRoot[V1EllipseEquation /. x → 0, {y, 1.0}][[1, 2]]]},
   V1Dialated = Reap[
      Scan[
       If[(#[[1]] / a)^2 + (#[[2]] / b)^2 ≤ 2.0, Sow[#]] &,
       V1Expanded]
     ][[2, 1]]];
```

Frequently we also want to plot the V1 convex hull (outline); this can be done with this variable...

```
V1Hull := (V1Hull = V1[[Append[#, First@#]]] &[ConvexHull[V1]]);
V1DialatedHull :=
  (V1DialatedHull = V1Dialated[[Append[#, First@#]]] &[ConvexHull[V1Dialated]]);
```

- **Coordinate Axes and V1 Ellipse**

Once data has been loaded, we want to be able to define a coordinate system that places the V1 ellipse around the origin with the major axis as the x-axis and the minor axis as the y-axis. To do this, we want to first fit the V1 ellipse to the V1 data then rotate and translate all of the imported data to coincide with this. In order to do this, we must first convert the V1 data into surface spherical coordinates--we would like to rotate it around so that the center of V1 is at (0°,0°) to assure that there is minimal spherical distortion in our projection. We then find a fit for the V1 ellipse that minimizes the number of points in V1Expanded inside the ellipse and maximizes the number of points in V1 inside the ellipse. We can do this by fitting an elliptical 2D Plateau to a rotated/transposed V1 and calling the ellipse border the curve where plateau(x,y) = 0.5.
For a plateau function, we use a modified (stretched/rotated) form of the sigmoid function 1/(1+exp(-t)) that has been rotated around the origin; we use a fixed gain (a high gain means the sigmoid has a very high derivative at 0).

```
BrainToMap[orig_List] := Module[
   {mu = Mean[V1Original] / Norm[Mean[V1Original]],
    rotationMatrix},
   (* we want the rotation that puts this vector at {1,0,0} *)
   rotationMatrix = RotationMatrix[{mu, {1, 0, 0}}];
   BrainToMap[dat_List] := MapThread[
     Join,
     {Transpose[
       $FSAverageShearMatrix.Transpose[
         CartesianToSpherical[
          Transpose[rotationMatrix.Transpose[dat[[All, 1 ;; 3]]]]]]],
      dat[[All, 4 ;; All]]}];
   BrainToMap[orig]];
```

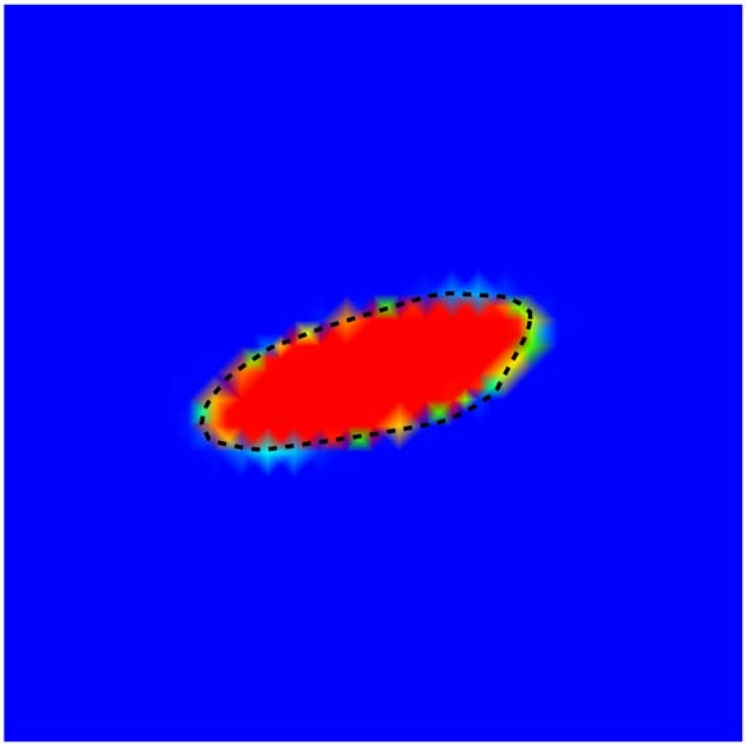
Now, we can do the sigmoid fitting (mentioned above) on the brain-to-map transformed V1 data.

```
V1SigmoidModel := (V1SigmoidModel = Block[{θ, x0, y0, σx, σy, x, y},
      Module[
       {V1Inside = BrainToMap[V1Original],
        V1Outside = BrainToMap[Complement[V1ExpandedOriginal, V1Original]],
        plateau = Function[{pt},
          1 - 1 / (1 +
               Exp[-20 *
                  (Sqrt[σx * ((pt[[1]] - x0) * Cos[θ] + (pt[[2]] - y0) * Sin[-θ])^2 +
                      σy *
                       ((pt[[1]] - x0) * Sin[θ] + (pt[[2]] - y0) * Cos[θ])^2] - 2)])]},
       (* Note that form uses an inverse rotation; this way θ will end
        up being the proper value for rotating the points in V1 rather
        than for rotating the Gaussian to the points.  We rotate/translate
        the Plateau during fitting in order to speed things up,
       but in the end we want to rotate/translate the points themselves. *)
       NonlinearModelFit[
        Join[
         Table[Append[x, 1.0], {x, V1Inside}],
         Table[Append[x, 0.0], {x, V1Outside}]],
        {1 - 1 / (1 + Exp[-20 * (Sqrt[σx * ((x - x0) * Cos[θ] + (y - y0) * Sin[-θ])^2 +
                    σy * ((x - x0) * Sin[θ] + (y - y0) * Cos[θ])^2] - 2)]),
         σy > 0 && σx > 0},
        {{x0, Mean[V1Inside[[All, 1]]]},
         {y0, Mean[V1Inside[[All, 2]]]},
         {θ, -Pi / 8},
         {σx, 5},
         {σy, 20}},
        {x, y},
        AccuracyGoal → 4]]);
```

If we wish, we can inspect the V1SigmoidModel to see what kind of fit we found. This forces evaluation of the fit, so in the interest of good lazy programming, it is commented out. The values that were fit when I ran it were:
{x0→-0.00622479, y0→-0.00138543, θ→-0.217459, σx→25.2198, σy→106.692}

```
(*V1SigmoidModel["BestFitParameters"]*)
```

```
If[$ShowFigures,
 Show[
  DensityPlot[
   V1SigmoidModel[x, y],
   {x, -1, 1},
   {y, -1, 1},
   ColorFunction → (Blend[{Blue, Cyan, Green, Yellow, Red}, #] &),
   PlotRange → Full,
   Frame → False],
  ListPlot[
   BrainToMap[V1Original][[
    Append[#, First@#] &@ConvexHull[BrainToMap@V1Original]]],
   PlotStyle → {Black, Dashed, Thick},
   Joined → True,
   ImageSize → 3.25 * 72,
   Frame → False]]]
```



Now that we have the model for the sigmoid, we can untranslate and unrotate all of the points in V1 by building a transformation (at which point V1 and all sets of spherical points already loaded will automatically translate themselves to our coordinate system).

The V1 transform must translate a point by (x0,y0) then must rutate a point by θ degrees. If this seems wrong, it's because we built the plateau with the wrong rotation formula, so it actually rotates -θ.

```
V1Transform[orig_List] := Block[{x0, y0, θ},
   Module[{params = V1SigmoidModel["BestFitParameters"], pt0, rotationMtx},
    {pt0, rotationMtx} = {{x0, y0}, RotationMatrix[θ]} //. params;
    V1Transform[dat_List] := Map[
      Join[
        (rotationMtx.(#[[1 ;; 2]] - pt0)),
        #[[3 ;; All]]] &,
      BrainToMap[dat]];
    V1Transform[orig]]];
```

Now, we would like the elliptical form of this model; we can solve the equation for values of 0.5 to find this particular equation.
Notably, since we want to find the ellipse parameters for the data post-rotation, what we really need at this point is to drop the x0, y0, and θ parameters, so this becomes much simpler.

```
(* Notably, we can use substitution to first show that Sqrt[σx*x^2+σy*y^2]==2;
this is our ellipse equation *)
V1EllipseEquation := Block[{σx, σy, x, y, u},
   V1EllipseEquation = Equal[
     σx * x^2 + σy * y^2 /. V1SigmoidModel["BestFitParameters"],
     u /. FindRoot[
       1 - 1 / (1 + Exp[-20 * (Sqrt[u] - 2)]) == 0.5,
       {u, 2}]]];

(* A similar technique gives us the Dialated
 ellipse (V1 ellipse with twice the radius) *)
V1DialatedEllipseEquation := Block[{σx, σy, x, y, u},
   V1DialatedEllipseEquation = Equal[
     σx * x^2 + σy * y^2 /. V1SigmoidModel["BestFitParameters"],
     2 * u /. FindRoot[
       1 - 1 / (1 + Exp[-20 * (Sqrt[u] - 2)]) == 0.5,
       {u, 2}]]];
```

Again, we can examine this if we wish to know the equation, but this will force evaluation. When I ran it, the result was:

$$106.693\ x^2 + 25.2195\ y^2 == 4.$$

```
(*V1EllipseEquation*)
```

Using this we can declare the V1 ellipse/test functions and inequalities...

```
InV1EllipseEquation := Block[{x, y},
   InV1EllipseEquation = Apply[LessEqual, V1EllipseEquation]];

InV1Q[a : {_, _, __}] :=
  (InV1Q[{x_, y_, rest__}] := Evaluate[InV1EllipseEquation]; InV1Q[a]);
InV1Q[a : {_, _}] := (InV1Q[{x_, y_}] := Evaluate[InV1EllipseEquation]; InV1Q[a]);
InV1Q[x0_, y0_] := (InV1Q[x_, y_] := Evaluate[InV1EllipseEquation]; InV1Q[x0, y0]);
```

We can also setup a similar set of equations/functions for the dialated V1 regions (see

previous section; this is an ellipse that has twice the radius as the V1 ellipse)

```
InV1DialatedEllipseEquation := Module[
   {v1a = Block[{x, y}, FindRoot[V1EllipseEquation /.y → 0, {x, 1.0}][[1, 2]]],
    v1b = Block[{x, y}, FindRoot[V1EllipseEquation /.x → 0, {y, 1.0}][[1, 2]]]},
   Block[{x, y},
    InV1DialatedEllipseEquation = ((x / v1a)^2 + (y / v1b)^2 ≤ 2)]];
InV1DialatedQ[a : {_, _, __}] := (InV1DialatedQ[{x_, y_, rest__}] :=
    Evaluate[InV1DialatedEllipseEquation]; InV1DialatedQ[a]);
InV1DialatedQ[a : {_, _}] := (InV1DialatedQ[{x_, y_}] :=
    Evaluate[InV1DialatedEllipseEquation]; InV1DialatedQ[a]);
InV1DialatedQ[x0_, y0_] := (InV1DialatedQ[x_, y_] :=
    Evaluate[InV1DialatedEllipseEquation]; InV1DialatedQ[x0, y0]);
```

Finally, we want some access to the parameters of the ellipse; most of these are pretty simple, but they are useful nonetheless

```
V1EllipseA := (V1EllipseA =
    Block[{x, y}, FindRoot[V1EllipseEquation /.y → 0, {x, 1.0}][[1, 2]]]);
V1EllipseB := (V1EllipseB = Block[{x, y},
     FindRoot[V1EllipseEquation /.x → 0, {y, 1.0}][[1, 2]]]);
V1EllipseX0 = 0;
V1Ellipsey0 = 0;
V1Ellipseθ = 0;

V1DialatedEllipseA := (V1DialatedEllipseA =
    Block[{x, y}, FindRoot[V1DialatedEllipseEquation /.y → 0, {x, 1.0}][[1, 2]]]);
V1DialatedEllipseB := (V1DialatedEllipseB =
    Block[{x, y}, FindRoot[V1DialatedEllipseEquation /.x → 0, {y, 1.0}][[1, 2]]]);
V1DialatedEllipseX0 = 0;
V1DialatedEllipsey0 = 0;
V1DialatedEllipseθ = 0;
```

- **Basic Plotting**

Here we define some basic plotting routines. These should make plotting considerably easier. All of them should be based on a particular mathematica plot and should simply do preprocessing or slightly modify the arguments while still accepting all the original arguments. Currently, there is only BrainPlot, which is a variant on ListDensityPlot.

```
(* BrainPlot is basically like ListDensityPlot, but with 2 extra
 arguments: Filling and Select (which can almost always be left as is) and
   with a few alternative default values that can easily be overridden. *)
Module[
  {defaultArgs = { (* the arguments for ListDensityPlot that are non-standard *)
     ImageSize → 3.25 * 72, (* by default images are 3.25 inches wide *)
     ColorFunction → $ColorScaleFunction,
     Frame → None,
     Axes → None,
     AspectRatio → 0.5,
```

```
    PlotRange → Automatic,
    LabelStyle → Directive[10, FontFamily → "Arial"]},
  ownArgs =
   { (* Arguments for BrainPlot that do not exist for ListDensityPlot *)
    Filling → LightGray, (* Option specifies the color
     to fill in for the V1Dialated region (or None) *)
    Select → InV1DialatedQ},
  (* used when filling in gray points *)
  binsz = 0.02},
 BrainPlot[data_List, opts : OptionsPattern[]] := Module[
  {sel = Function[{selectarg},
      If[selectarg === None, (# &), Select[#, selectarg] &]
     ][OptionValue[Select]],
   fillarg = OptionValue[Filling],
   fill = Module[
     {xs = Table[x,
        {x, -V1DialatedEllipseA, V1DialatedEllipseA + binsz - 0.00001, binsz}],
      ys = Table[y, {y, -V1DialatedEllipseB, V1DialatedEllipseB +
           binsz - 0.00001, binsz}]},
     Function[{fillarg},
      Function[{dat},
       Join[
        dat,
        Flatten[
         Reap[
           MapThread[
            Function[{bin, pt}, If[bin < 1 && InV1DialatedQ[pt], Sow[pt]]],
            {BinCounts[
              dat[[All, 1 ;; 2]],
              {First@xs, Last@xs, binsz},
              {First@ys, Last@ys, binsz}],
             Table[
              {x, y, -2000},
              {x, Most@xs},
              {y, Most@ys}]},
            2]
          ][[2]],
         1]]]]],
   drange = {Min[data[[All, 3]]], Max[data[[All, 3]]]},
   prolog = OptionValue[Prolog] /. {None → {}, Automatic → {}},
   prange = OptionValue[PlotRange],
   cfun = OptionValue[ColorFunction],
   cfs = OptionValue[ColorFunctionScaling],
   img},
  If[cfun === Automatic, cfun = OptionValue[defaultArgs, ColorFunction]];
  fill = fill[fillarg];
  prange = Which[
    prange === Automatic,
    Which[
     drange[[2]] > 91.0, {0, 180},
     drange[[2]] > 10.0, {0, 20},
```

```
      drange[[2]] > 0, {0, 10},
      True, {-180, 0}],
     prange === Full, drange,
     True, prange];
   ListDensityPlot[
    Evaluate[fill[sel[data]]],
    Evaluate[
     FilterRules[
      Join[
       {ColorFunctionScaling → False,
        ColorFunction → If[cfs == True || cfs === Automatic,
          If[# < -1000, fillarg, cfun[
             Median[{0, 1, (# - prange[[1]]) / (prange[[2]] - prange[[1]])}]]] &,
          If[# < -1000, fillarg, cfun[Median[{prange[[1]], prange[[2]], #}]]] &],
        Prolog → If[fillarg =!= None,
          Join[{fillarg,
            Disk[{0, 0}, {V1DialatedEllipseA, V1DialatedEllipseB}]}, prolog],
          prolog]},
       FilterRules[
        {opts},
        Except[
         {ColorFunctionScaling → False, PlotRange → Automatic, Prolog → None}]],
       FilterRules[
        defaultArgs,
        Except[{opts}]]],
      Options[ListDensityPlot]]]]];
Options[BrainPlot] = Join[
  ownArgs,
  defaultArgs,
  FilterRules[Options[ListDensityPlot], Except[defaultArgs]]];
```

- **Aggregating Subjects**

This is just a handy routine for aggregating subject data. It sums across points according to their x and y coordinate; anything landing within a bin of size 0.005, which is small enough that points only fall in the same bin when they're from different hemispheres but are the same vertex, is passed through the second argument. Only the list of the third coordinates are passed to this function.

```
AggregateSubjects[dataList_List, aggFunc_] := Flatten[
   Reap[
     Scan[
      If[Length@# > 0,
        Sow[{Mean[#[[All, 1]]], Mean[#[[All, 2]]], aggFunc[#[[All, 3]]]}]] &,
      Map[
       Flatten[#, 1] &,
       BinLists[
        Flatten[dataList, 1],
        0.005,
        0.005,
        100],
       {2}],
      {2}]
    ][[2]],
   1];
```

- **Fitting Eccentricity and Polar Angle**

We want to fit templates to the subjects' BOLD fMRI data; the form of the templates are defined here. Note that EccentricityCoordinate[] and PolarAngleCoordinate[] are functions that convert a point in V1 into a single number that should, according to models such as Schira et al and Balasubramanian et al, be equivalent to the axis along which the respective values lie.

```
Block[{q, x, y},
  $PolarAngleTemplateForm :=
   90.0 + 90.0 * (Sign[y] * Abs[PolarAngleCoordinate[x, y]]^q);
  $PolarAngleTemplateConstraints := (q > 0.001);
  $PolarAngleTemplateParameters := {{q, 1.5}};
 ];

Block[{q, x, y},
  $EccentricityTemplateForm :=  90.0 * Exp[q * (EccentricityCoordinate[x, y] - 1)];
  $EccentricityTemplateConstraints := (q > 0.001);
  $EccentricityTemplateParameters := {{q, 4.0}};
 ];

(* An alternate template; we don't actually use this in the paper,
but it demonstrates how one could change the fit function. *)
(*
Block[{q, x, y},
  $EccentricityTemplateForm :=
   90.0*(Exp[q*EccentricityCoordinate[x, y]] - 1)/(Exp[q] - 1);
  $EccentricityTemplateConstraints:=(q>0.001);
  $EccentricityTemplateParameters:={{q,4.0}};
 ];
*)
```

Fitting is done via equation forms using Mathematica's NonlinearModelFit routine. In order to facilitate this, we want to be able to define a coordinate system over the

V1 ellipse that uses decreasing ellipses and hyperbolas to give each vertex in V1 a value on the range of [0,1] for the hyperbolas (which map roughly to eccentricity) and [-1,1] for the ellipses (which map roughly to polar angle).
We call this the EllipticalCoordinateSystem.

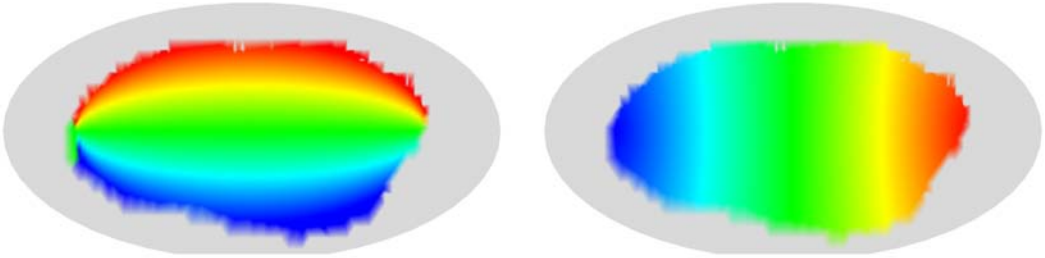
We can determine this coordinate system from the V1 ellipse parameters, which are drastically simplified since we've rotated it to be of a very simple form.

Eccentricity: We know from geometry that every ellipse of the form (x/a)^2 + (y/b)^2 = 1 is orthogonal to every hyperbola of the form x^2/(a^2-t) + y^2/(t - b^2) = 1. We want to solve for the hyperbola of that form which passes through a particular point in V1 and use that hyperbola's x-intercept as the eccentricity elliptical coordinate.

PolarAngle: We just want to define the ellipse with an a parameter equal to the V1 ellipse's a param and a b parameter that forces the ellipse to pass through the point in question.

```
Module[
 {form = Block[{a, b, x0, y0, ϴ, x, y, t},
  FullSimplify[ (* Here, we simplify for the hyperbola form *)
   ReplaceAll[
   Sqrt[a^2 - t] / a,
   Simplify[
    Solve[
     x^2 / (a^2 - t) - y^2 / (t - b^2) == 1,
     t,
     Reals],
    {a^2 > t > b^2 > 0, x > 0, y > 0}][[2, 1]]]]],
 norm = Block[{a, b}, Sqrt[a^2 - b^2] / a]},
 (* Use these to declare the actual eccentricity coordinate function... *)
 EccentricityCoordinate[λ0_, ϕ0_] := Module[
  {v1a =
      Block[{x, y}, FindRoot[V1EllipseEquation /. y -> 0, {x, 1.0}][[1, 2]]],
  v1b = Block[{x, y}, FindRoot[V1EllipseEquation /. x -> 0, {y, 1.0}][[1, 2]]]},
  (* above we solved for the x- and y-intercepts,
     which are the a/b parameters in the ellipse *)
  form = Block[{a, b}, ReplaceAll[form, {a -> v1a, b -> v1b}]];
  norm = Block[{a, b}, ReplaceAll[norm, {a -> v1a, b -> v1b}]];
  EccentricityCoordinate[λ_, ϕ_] := Block[{x, y},
   ReplaceAll[
    Piecewise[
     {{0.5 * (form / norm + 1), x > 0}},
     0.5 * (1 - form / norm)],
    {x -> λ * norm, y -> ϕ * norm}]];
  EccentricityCoordinate[λ0, ϕ0]];
 (* and the polar angle function *)
 PolarAngleCoordinate[λ0_, ϕ0_] := Module[
  {v1a =
      Block[{x, y}, FindRoot[V1EllipseEquation /. y -> 0, {x, 1.0}][[1, 2]]],
  v1b = Block[{x, y}, FindRoot[V1EllipseEquation /. x -> 0, {y, 1.0}][[1, 2]]]},
  PolarAngleCoordinate[λ_, ϕ_] := Piecewise[
   {{(ϕ * v1a / Sqrt[v1a^2 - λ^2]) / v1b, Abs[λ] < v1a}},
   0];
  PolarAngleCoordinate[λ0, ϕ0]]];
```

```
(* If figures are on, this will plot a nice image of the V1 region
 as indexed by their polar angle and eccentricity coordinates. *)
If[$ShowFigures,
 GraphicsRow[
 {BrainPlot[
  Map[
      Append[#, PolarAngleCoordinate[#[[1]], #[[2]]]] &,
      V1],
  AspectRatio -> 0.5,
     PlotRange → {-1, 1},
  ColorFunctionScaling -> False,
  ColorFunction -> (Blend[{Blue, Cyan, Green, Yellow, Red}, (# + 1) / 2] &)],
 BrainPlot[
  Map[
   Append[#, EccentricityCoordinate[#[[1]], #[[2]]]] &,
   V1],
  AspectRatio -> 0.5,
     PlotRange → {0, 1},
  ColorFunctionScaling -> False]}]]
```



Now that we have those functions setup, we can define the fitting functions!
These should take polar angle or eccentricity data for a subject and produce a FittedModel.

```
FitEccentricity[dat_List, OptionsPattern[]] := Block[{q, x, y},
 Module[
  {fit = q /. FindFit[
    Select[
     dat,
     OptionValue[Select] /. {Automatic -> InV1Q,
            {min_, max_} :> (InV1Q[#] && min <= #[[3]] <= max &)}],
    {$EccentricityTemplateForm, $EccentricityTemplateConstraints},
    $EccentricityTemplateParameters,
    {x, y},
    NormFunction -> Function[{residual}, Norm[Log[Abs[residual] + 1]]],
    Method -> "NMinimize",
    AccuracyGoal -> 6]},
  NonlinearModelFit[
   dat,
   {$EccentricityTemplateForm,
    fit - 0.00001 < q <= fit},
   {{q, fit}},
   {x, y},
   AccuracyGoal -> 6]]];
Options[FitEccentricity] = {Select -> InV1Q};

FitPolarAngle[dat_List, OptionsPattern[]] := Block[{q, x, y},
 Module[
  {fit = q /. FindFit[
    Select[
     dat,
     OptionValue[Select] /. {Automatic → InV1Q}],
    {$PolarAngleTemplateForm, $PolarAngleTemplateConstraints},
    $PolarAngleTemplateParameters,
    {x, y},
    NormFunction -> Function[{residual}, Norm[Log[Abs[residual] + 1]]],
    Method -> "NMinimize",
    AccuracyGoal -> 6]},
  NonlinearModelFit[
   dat,
   {$PolarAngleTemplateForm,
    fit - 0.00001 < q <= fit},
   {{q, fit}},
   {x, y}]]];
Options[FitPolarAngle] = {Select -> InV1Q};
```

- **Datasets**

The DeclareDataset[] function allows us to setup a dataset for use in this file. A dataset should generally be a series of subjects whose data were collected according to the same procedure and can be analyzed as a unit.
When a dataset is declared, a large number of lazy-evaluation data items are defined along with it. As soon as a dataset is defined (e.g. dataset = DeclareDataset[...];), one can already call dataset[Template,Eccentricity], for example, to get the eccentricity

template associated with the that dataset. Requesting this item will force the dataset to load all of the subjects' data, parse/transform it to the V1 elliptical system, and fit the templates, but the results of each of these processes will be saved along the way so that subsequent requests for any of them will be immediate.

```
DeclareDataset[
   doc_String,
   subjects : {_String ..},
   loadFunction_,
   OptionsPattern[]] :=
  Module[
   {preproc = OptionValue[Preprocessor],
    postproc = OptionValue[Postprocessor],
    idvert = OptionValue[IdenticalVertices],
    stimulusRange = OptionValue[StimulusRange],
    pasel = OptionValue[PolarAngleSelect],
    eccsel = OptionValue[EccentricitySelect],
    rhreverse = OptionValue[ReverseRightHemisphere],
    data},

   data[Description] = doc;
   data[Subjects] = subjects;
   data[StimulusRange] = stimulusRange;
   data[Preprocessor] = If[preproc === None, (None &), preproc];
   data[Postprocessor] = If[postproc === None, (#2 &), postproc];
   data[IdenticalVertices] = idvert;
   data[LoadFunction] = loadFunction;
   data[PolarAngleSelect] = pasel;
   data[EccentricitySelect] = eccsel;

   (* How we load the data--this auto-caches itself *)
   data[sub_?(StringQ[#] || IntegerQ[#] &)][Data] := Module[
     {subject = If[StringQ[sub], sub, subjects[[sub]]],
      subjectID =
       If[StringQ[sub], First[Flatten[Position[subjects, sub]]], sub]},
     If[
      MemberQ[subjects, subject],
      Module[
       {pre = data[Preprocessor][subjectID], tmp},
       tmp = Map[
         V1Transform,
         data[Postprocessor][subject, loadFunction[subject, pre], pre],
         {2}];
       If[(rhreverse === Automatic && Mean[tmp[[2, 1, All, 3]]] < 0) ||
         rhreverse === True,
        tmp[[2, 1]] = tmp[[2, 1]].{{1, 0, 0}, {0, 1, 0}, {0, 0, -1}}];
       data[subjectID][Data] = tmp;
       data[subject][Data] = tmp;
       tmp],
      (Print["Provided subjectID is not a subject in this dataset"]; Abort[])]];
```

```
(* How we calculate the fits... will also auto-cache itself *)
data[sub_?(StringQ[#] || IntegerQ[#] &)][Fit] := Module[
  {subject = If[StringQ[sub], sub, subjects[[sub]]],
   subjectID =
    If[StringQ[sub], First[Flatten[Position[subjects, sub]]], sub]},
  If[
   MemberQ[subjects, subject],
   (data[subjectID][Fit] = List[
      (* LH first *)
      {FitPolarAngle[data[subjectID][LeftHemisphere, PolarAngle],
        Select → pasel],
       FitEccentricity[data[subjectID][LeftHemisphere, Eccentricity],
        Select → eccsel]},
      (* Then RH *)
      {FitPolarAngle[data[subjectID][RightHemisphere, PolarAngle],
        Select → pasel],
       FitEccentricity[data[subjectID][RightHemisphere, Eccentricity],
        Select → eccsel]}];
    data[subject][Fit] = data[subjectID][Fit];
    data[subjectID][Fit]),
   (Print[
     "Provided subjectID is not a subject in this dataset"]; Abort[])]];

(* We also want access to leave-one-out template fits;
for a single subject, this is the template fit from everyone else *)
data[sub_?(StringQ[#] || IntegerQ[#] &)][LeaveOneOut,
  type : (PolarAngle | Eccentricity)] := Module[
  {subject = If[StringQ[sub], sub, subjects[[sub]]],
   subjectID =
    If[StringQ[sub], First[Flatten[Position[subjects, sub]]], sub]},
  If[
   MemberQ[subjects, subject], Module[
    {pts = Select[
       Flatten[Delete[data[type], subjectID], 2],
       data[If[type === PolarAngle, PolarAngleSelect, EccentricitySelect]]]},
    If[type === PolarAngle,
     data[subjectID][LeaveOneOut, type] = FitPolarAngle[pts],
     data[subjectID][LeaveOneOut, type] = FitEccentricity[pts]];
    data[subject][LeaveOneOut, type] = data[subjectID][LeaveOneOut, type]],
   (Print["Error: " <> subject <> " is not a subject in dataset"]; Abort[])]];
data[sub_?(StringQ[#] || IntegerQ[#] &)][
  LeaveOneOut, hem : (RightHemisphere | LeftHemisphere),
  type : (PolarAngle | Eccentricity)] := Module[
  {subject = If[StringQ[sub], sub, subjects[[sub]]],
   subjectID =
    If[StringQ[sub], First[Flatten[Position[subjects, sub]]], sub]},
  If[
   MemberQ[subjects, subject], Module[
    {pts = Select[
       Flatten[Delete[data[hem, type], subjectID], 1],
```

```
        data[If[type === PolarAngle, PolarAngleSelect, EccentricitySelect]]},
     If[type === PolarAngle,
      data[subjectID][LeaveOneOut, hem, type] = FitPolarAngle[pts],
      data[subjectID][LeaveOneOut, hem, type] = FitEccentricity[pts]];
     data[subject][LeaveOneOut, hem, type] =
      data[subjectID][LeaveOneOut, hem, type]],
    (Print["Error: " <> subject <> " is not a subject in dataset"]; Abort[])]];
data[LeaveOneOut] :=
 (data[LeaveOneOut] = Map[data[#][LeaveOneOut] &, subjects]);
data[LeaveOneOut, hem : (RightHemisphere | LeftHemisphere)] :=
 (data[LeaveOneOut, hem] = Map[data[#][LeaveOneOut, hem] &, subjects]);
data[LeaveOneOut, type : (PolarAngle | Eccentricity)] :=
 (data[LeaveOneOut, type] = Map[data[#][LeaveOneOut, type] &, subjects]);
data[LeaveOneOut, hem : (RightHemisphere | LeftHemisphere),
  type : (PolarAngle | Eccentricity)] := (data[LeaveOneOut, hem, type] =
   Map[data[#][LeaveOneOut, hem, type] &, subjects]);
data[LeaveOneOut, type : (PolarAngle | Eccentricity),
  hem : (RightHemisphere | LeftHemisphere)] := (data[LeaveOneOut, hem, type] =
   Map[data[#][LeaveOneOut, hem, type] &, subjects]);

(* All data: will autocache itself *)
data[Data] := (data[Data] = Map[data[#][Data] &, subjects]);
data[LeftHemisphere] := data[Data][[All, 1]];
data[RightHemisphere] := data[Data][[All, 2]];
data[PolarAngle] := data[Data][[All, All, 1]];
data[Eccentricity] := data[Data][[All, All, 2]];
data[LeftHemisphere, PolarAngle] := data[Data][[All, 1, 1]];
data[LeftHemisphere, Eccentricity] := data[Data][[All, 1, 2]];
data[RightHemisphere, PolarAngle] := data[Data][[All, 2, 1]];
data[RightHemisphere, Eccentricity] := data[Data][[All, 2, 2]];
data[PolarAngle, LeftHemisphere] := data[Data][[All, 1, 1]];
data[Eccentricity, LeftHemisphere] := data[Data][[All, 1, 2]];
data[PolarAngle, RightHemisphere] := data[Data][[All, 2, 1]];
data[Eccentricity, RightHemisphere] := data[Data][[All, 2, 2]];
data[Fit] := (data[Fit] = Map[data[#][Fit] &, subjects]);
data[FitPolarAngle] :=
 (data[FitPolarAngle] = Map[data[#][FitPolarAngle] &, subjects]);
data[FitEccentricity] := (data[FitEccentricity] =
   Map[data[#][FitEccentricity] &, subjects]);
data[FitPolarAngle, LeftHemisphere] := (data[FitPolarAngle, LeftHemisphere] =
   Map[data[#][FitPolarAngle, LeftHemisphere] &, subjects]);
data[FitPolarAngle, RightHemisphere] := (data[FitPolarAngle, RightHemisphere] =
   Map[data[#][FitPolarAngle, RightHemisphere] &, subjects]);
data[FitEccentricity, RightHemisphere] :=
 (data[FitEccentricity, RightHemisphere] =
   Map[data[#][FitEccentricity, RightHemisphere] &, subjects]);
data[FitEccentricity, LeftHemisphere] :=
 (data[FitEccentricity, LeftHemisphere] =
   Map[data[#][FitEccentricity, LeftHemisphere] &, subjects]);
data[FitParameters] := (data[FitParameters] = Map[data[#][Fit] &, subjects]);
MapThread[
```

```
Function[{sID, s},
 data[sID][RightHemisphere] := data[s][Data][[2]];
 data[sID][LeftHemisphere] := data[s][Data][[1]];
 data[sID][PolarAngle] := data[s][Data][[All, 1]];
 data[sID][Eccentricity] := data[s][Data][[All, 2]];
 data[sID][RightHemisphere, PolarAngle] := data[s][Data][[2, 1]];
 data[sID][PolarAngle, RightHemisphere] := data[s][Data][[2, 1]];
 data[sID][LeftHemisphere, PolarAngle] := data[s][Data][[1, 1]];
 data[sID][PolarAngle, LeftHemisphere] := data[s][Data][[1, 1]];
 data[sID][RightHemisphere, Eccentricity] := data[s][Data][[2, 2]];
 data[sID][Eccentricity, RightHemisphere] := data[s][Data][[2, 2]];
 data[sID][LeftHemisphere, Eccentricity] := data[s][Data][[1, 2]];
 data[sID][Eccentricity, LeftHemisphere] := data[s][Data][[1, 2]];
 (* Here, we declare the fits *)
 data[sID][FitPolarAngle] := data[s][Fit][[All, 1]];
 data[sID][FitEccentricity] := data[s][Fit][[All, 2]];
 data[sID][RightHemisphere, FitPolarAngle] := data[s][Fit][[2, 1]];
 data[sID][FitPolarAngle, RightHemisphere] := data[s][Fit][[2, 1]];
 data[sID][LeftHemisphere, FitPolarAngle] := data[s][Fit][[1, 1]];
 data[sID][FitPolarAngle, LeftHemisphere] := data[s][Fit][[1, 1]];
 data[sID][RightHemisphere, FitEccentricity] := data[s][Fit][[2, 2]];
 data[sID][FitEccentricity, RightHemisphere] := data[s][Fit][[2, 2]];
 data[sID][LeftHemisphere, FitEccentricity] := data[s][Fit][[1, 2]];
 data[sID][FitEccentricity, LeftHemisphere] := data[s][Fit][[1, 2]];
 (* We also want an interface to fit parameters... *)
 data[sID][FitParameters] := (data[sID][FitParameters] =
    Map[#["BestFitParameters"][[1, 2]] &, data[sID][Fit], {2}]);
 data[sID][FitParameters, Eccentricity] :=
  data[sID][FitParameters][[All, 2]];
 data[sID][FitParameters, PolarAngle] := data[sID][FitParameters][[All, 1]];
 data[sID][FitParameters, LeftHemisphere] := data[sID][FitParameters][[1]];
 data[sID][FitParameters, RightHemisphere] := data[sID][FitParameters][[2]];
 data[sID][FitParameters, LeftHemisphere, PolarAngle] :=
  data[sID][FitParameters][[1, 1]];
 data[sID][FitParameters, LeftHemisphere, Eccentricity] :=
  data[sID][FitParameters][[1, 2]];
 data[sID][FitParameters, RightHemisphere, PolarAngle] :=
  data[sID][FitParameters][[2, 1]];
 data[sID][FitParameters, RightHemisphere, Eccentricity] :=
  data[sID][FitParameters][[2, 2]];
 data[sID][FitParameters, PolarAngle, LeftHemisphere] :=
  data[sID][FitParameters][[1, 1]];
 data[sID][FitParameters, Eccentricity, LeftHemisphere] :=
  data[sID][FitParameters][[1, 2]];
 data[sID][FitParameters, PolarAngle, RightHemisphere] :=
  data[sID][FitParameters][[2, 1]];
 data[sID][FitParameters, Eccentricity, RightHemisphere] :=
  data[sID][FitParameters][[2, 2]];
],
{Join[Range[Length@subjects], subjects], Join[subjects, subjects]}];
```

```
(* We want to be able to produce aggregates... *)
data[Aggregate, hem : (LeftHemisphere | RightHemisphere),
  type : (PolarAngle | Eccentricity)] := Module[
  {binnedData = Transpose[(* Here,
     we collect the points into individual bins *)
     Map[
      Flatten[#, 1] &,
      BinLists[
       Select[
        Flatten[data[hem, type], 1],
        InV1DialatedQ],
       {Table[x, {x, -V1DialatedEllipseA, V1DialatedEllipseA, 0.005}]},
       {Table[y, {y, -V1DialatedEllipseB, V1DialatedEllipseB, 0.005}]},
       {{-0.01, 180.01}}],
      {2}]]},
  (* Having done that, we can go through and reap the medians! *)
  data[Aggregate, hem, type] = Reap[
     Scan[
      If[Length@# > 2 &&
         StandardDeviation[#[[All, 3]]] < If[type === PolarAngle, 60, 3.3],
        Sow[{Mean[#[[All, 1]]], Mean[#[[All, 2]]], Median[#[[All, 3]]]}]] &,
      binnedData,
      {2}]
    ][[2, 1]]];
data[Aggregate, type : (PolarAngle | Eccentricity)] := Module[
  {binnedData =
    Transpose[(* Here, we collect the points into individual bins *)
     Map[
      Flatten[#, 1] &,
      BinLists[
       Select[
        Flatten[data[type], 2],
        InV1DialatedQ],
       {Table[x, {x, -V1DialatedEllipseA, V1DialatedEllipseA, 0.005}]},
       {Table[y, {y, -V1DialatedEllipseB, V1DialatedEllipseB, 0.005}]},
       {{-0.01, 180.01}}],
      {2}]]},
  (* Having done that, we can go through and reap the medians! *)
  data[Aggregate, type] = Reap[
     Scan[
      If[Length@# > 2 &&
         StandardDeviation[#[[All, 3]]] < If[type === PolarAngle, 60, 3.3],
        Sow[{Mean[#[[All, 1]]], Mean[#[[All, 2]]], Median[#[[All, 3]]]}]] &,
      binnedData,
      {2}]
    ][[2, 1]]];
data[Aggregate] :=
 {data[Aggregate, PolarAngle], data[Aggregate, Eccentricity]};

(* We also want to be able to produce templates;
simultaneous fits to all data points; note that we pre-
```

```
    select here instead of allowing the NonlinearModelFit to include all points;
   this is because there are just so many when you fit the
    aggregate that we don't want it thinking about the extras *)
   data[Template, hem : (RightHemisphere | LeftHemisphere), PolarAngle] := (
     data[Template, hem, PolarAngle] = FitPolarAngle[
       Select[
        Flatten[data[hem, PolarAngle], 1],
        data[PolarAngleSelect]]]);
   data[Template, hem : (RightHemisphere | LeftHemisphere), Eccentricity] := (
     data[Template, hem, Eccentricity] = FitEccentricity[
       Select[
        Flatten[data[hem, Eccentricity], 1],
        data[EccentricitySelect]]]);
   data[Template, PolarAngle] := (
     data[Template, PolarAngle] = FitPolarAngle[
       Select[
        Flatten[data[PolarAngle], 2],
        data[PolarAngleSelect]]]);
   data[Template, Eccentricity] := (
     data[Template, Eccentricity] = FitEccentricity[
       Select[
        Flatten[data[Eccentricity], 2],
        data[EccentricitySelect]]]);
   data[Template] := {data[Template, PolarAngle], data[Template, Eccentricity]};

   (* Finally, return the symbol to which this is all attached. *)
   data];
Options[DeclareDataset] = {
   (* The range of the stimulus used during the experiment;
   by default this is out to 10° *)
   StimulusRange → {0, 10},
   (* The preprocessor function is called immediately before a
    subject is loaded.  It is passed the subject id.  Its result is
    passed to the loader function and the postprocessor function *)
   Preprocessor → None,
   (* The postprocessor is called immediately after loading and
    transformations.  It is passed the subject id, the loaded/transformed data,
   and the return value of the preprocessor (None if there is no preprocessor).
       It should return the data after performing any post-processing:
      data = postprocessor[id, data, preprocessor_return] *)
   Postprocessor → None,
   (* If the eccentricity and polar angle
    data contain identical vertices (ie, they were fit jointly,
     perhaps with the PRF method), this should be true *)
   IdenticalVertices → False,
   (* These options are ways to pass
    arguments directly to FitEccentricity and FitPolarAngle *)
   EccentricitySelect → InV1Q,
   PolarAngleSelect → InV1Q,
   (* Some datasets have RH data that is in negative degrees;
   we want to convert these (done automatically),
```

```
    but you can turn this off or force it to invert using this option *)
    ReverseRightHemisphere → Automatic
   };
 (* make sure our options are protected *)
 Scan[
   (Unprotect[#]; # = #; Protect[#];) &,
   Join[
    Options[DeclareDataset][[All, 1]],
    {Description, Subjects, StimulusRange, Eccentricity,
     PolarAngle, RightHemisphere, LeftHemisphere, LoadFunction,
     Data, FitParameters, Aggregate, Template, LeaveOneOut}]];
```

- **Joining Dataset Templates**

This function will let us join the templates of two datasets into a single template

```
JointTemplate[datasets_List, hemi : (LeftHemisphere | RightHemisphere | None),
   type : (Eccentricity | PolarAngle)] := Module[
   {qq = Median[
      Map[
       If[hemi === None,
        (#[Template, type]["BestFitParameters"][[1, 2]] &),
        (#[Template, hemi, type]["BestFitParameters"][[1, 2]] &)],
       datasets]]},
   Switch[type,
    PolarAngle, Function[{x0, y0},
     Evaluate[Block[{x = x0, y = y0, q = qq}, $PolarAngleTemplateForm]]],
    Eccentricity, Function[{x0, y0}, Evaluate[
      Block[{x = x0, y = y0, q = qq}, $EccentricityTemplateForm]]]]];
```

Dataset Declaration

This section includes parameters that specify where we load the data. Each dataset is a distinct set of data (eg, different retinotopy experiments) that has been collected. For our purposes, we have labeled the datasets by how many degrees of eccentricity were mapped (10 and 20). Datasets must be declared via the DeclareDataset function, which is documented in the Introduction/Local Functions section.
Note that the datasets do not load instantaneously; instead, they load themselves once the data is requested. If you want to force the dataset to load immediately, evaluate
data[<dataset-name>][Data]
immediately after declaring it.

After declaring a dataset, we can use it in a few ways. Suppose we've just declared a dataset and stored it in the variable set1:
set1[Data] => the full list of {{LH polar angle, LH eccentricity}, {RH polar angle}, {RH eccentricity}} for each subject

set1[LeftHemisphere] => the full list of {LH polar angle, LH eccentricity} for each subject
set1[Eccentricity] => the full list of {LH eccentricity, RH eccentricity} for each subject
set1[RightHemisphere, PolarAngle] => the full list of RH polar angle for each subject
set1[“subject1”][RightHemisphere] => the {RH polar angle, RH eccentricity} data for subject “subject1” assuming this subject exists

- **Declare Dataset 1 (10° stimulus)**

Here we declare our first dataset, which we call dataset 10 (because the stimulus goes out to 10° of eccentricity). The

```
(* Our first dataset: data10; for our file-load function,
we use an fstat cutoff of 8 *)
With[
 {$FStatCutoff = 8},

 data10 = DeclareDataset[
  "Our dataset of subjects with retinotopy measured out to 10°",

  (* We find our subjects from a file... *)
  Import[$TemplateDirectory <> "/subjects/subjects-dataset10.txt", "Words"],

  (* The function that, given a subject id, loads the data *)
  Function[{id},
   Module[
    {lfl = $TemplateDirectory <> "/subjects/" <> id <> "-lh.dat",
     rfl = $TemplateDirectory <> "/subjects/" <> id <> "-rh.dat",
     (* Here's how we collect a hemisphere's data... *)
     import = Function[{file},
      Flatten[
       Last[
        Reap[
         Replace[
          Import[file, "Table"],
          {x_, y_, z_, polarAngle_, eccentricity_, fstat_} :> (
           If[fstat >= $FStatCutoff,
            Sow[
             Append[{x, y, z}, polarAngle], "PolarAngle"];
            Sow[
             Append[{x, y, z}, eccentricity], "Eccentricity"]]),
          {1}],
         {"PolarAngle", "Eccentricity"}]],
        1]]},
     (* First, read in the left *)
     {import[lfl], import[rfl]}]],

  (* This dataset has RH data that goes from -180->0, so we correct for this *)
  ReverseRightHemisphere -> True,

  (* Selections for fitting *)
  EccentricitySelect ->
      (InV1Q[#] && 2.5 <= #[[3]] <= 8.0 && #[[1]] < V1EllipseA / 3 &),
  PolarAngleSelect -> (InV1Q[#] && #[[1]] < V1EllipseA / 3 &),

  (* Other details *)
  StimulusRange -> {{0, 180}, {0, 10}},
  IdenticalVertices -> True]];
```

- **Declare Half-length Datasets**

These datasets are the same as the above dataset, but they are each for half the length; we ca use these to look at reproducibility errors.

```
With[
 {$FStatCutoff = 1},

 data10half1 = DeclareDataset[
  "Our dataset of subjects with retinotopy measured out to 10°",

  (* We find our subjects from a file... *)
  Import[
      $TemplateDirectory <> "/subjects/subjects-dataset10half1.txt", "Words"],

  (* The function that, given a subject id, loads the data *)
  Function[{id},
   Module[
    {lfl = $TemplateDirectory <> "/subjects/" <> id <> "-lh.dat",
     rfl = $TemplateDirectory <> "/subjects/" <> id <> "-rh.dat",
     (* Here's how we collect a hemisphere's data... *)
     import = Function[{file},
      Flatten[
       Last[
        Reap[
         Replace[
          Import[file, "Table"],
          {x_, y_, z_, polarAngle_, eccentricity_, fstat_} :> (
           If[fstat >= $FStatCutoff,
            Sow[
             Append[{x, y, z}, polarAngle], "PolarAngle"];
            Sow[
             Append[{x, y, z}, eccentricity], "Eccentricity"]]),
          {1}],
         {"PolarAngle", "Eccentricity"}]],
        1]]},
     (* First, read in the left *)
     {import[lfl], import[rfl]}]],

  (* This dataset has RH data that goes from -180->0, so we correct for this *)
  ReverseRightHemisphere -> True,

  (* Selections for fitting *)
  EccentricitySelect ->
      (InV1Q[#] && 2.5 <= #[[3]] <= 8.0 && #[[1]] < V1EllipseA / 3 &),
  PolarAngleSelect -> (InV1Q[#] && #[[1]] < V1EllipseA / 3 &),

  (* Other details *)
  StimulusRange -> {{0, 180}, {0, 10}},
  IdenticalVertices -> True]];
```

```
With[
 {$FStatCutoff = 1},

 data10half2 = DeclareDataset[
  "Our dataset of subjects with retinotopy measured out to 10°",

  (* We find our subjects from a file... *)
  Import[
      $TemplateDirectory <> "/subjects/subjects-dataset10half2.txt", "Words"],

  (* The function that, given a subject id, loads the data *)
  Function[{id},
   Module[
    {lfl = $TemplateDirectory <> "/subjects/" <> id <> "-lh.dat",
     rfl = $TemplateDirectory <> "/subjects/" <> id <> "-rh.dat",
     (* Here's how we collect a hemisphere's data... *)
     import = Function[{file},
      Flatten[
       Last[
       Reap[
        Replace[
        Import[file, "Table"],
        {x_, y_, z_, polarAngle_, eccentricity_, fstat_} :> (
         If[fstat >= $FStatCutoff,
          Sow[
           Append[{x, y, z}, polarAngle], "PolarAngle"];
          Sow[
           Append[{x, y, z}, eccentricity], "Eccentricity"]]),
        {1}],
       {"PolarAngle", "Eccentricity"}]],
      1]]},
    (* First, read in the left *)
    {import[lfl], import[rfl]}]],

  (* This dataset has RH data that goes from -180->0, so we correct for this *)
  ReverseRightHemisphere -> True,

  (* Selections for fitting *)
  EccentricitySelect ->
      (InV1Q[#] && 2.5 <= #[[3]] <= 8.0 && #[[1]] < V1EllipseA / 3 &),
  PolarAngleSelect -> (InV1Q[#] && #[[1]] < V1EllipseA / 3 &),

  (* Other details *)
  StimulusRange -> {{0, 180}, {0, 10}},
  IdenticalVertices -> True]];
```

- **Declare Dataset 2 (20° stimulus)**

Here we declare our second dataset, which we call dataset 20 (because the stimulus goes out to 20° of eccentricity). The

```
(* Our second dataset: Dataset[20];
for our file-load function, we use an fstat cutoff of 5 *)
With[
  {$FStatCutoff = 3},

  data20 = DeclareDataset[
    "Our dataset of subjects with retinotopy measured out to 20°",

    (* We find our subjects from a file... *)
    Import[$TemplateDirectory <> "/subjects/subjects-dataset20.txt", "Words"],

    (* The function that, given a subject id, loads the data *)
    Function[{id},
     Module[
      {lfl = $TemplateDirectory <> "/subjects/" <> id <> "-lh.dat",
       rfl = $TemplateDirectory <> "/subjects/" <> id <> "-rh.dat",
       (* Here's how we collect a hemisphere's data... *)
       import = Function[{file},
         Flatten[
          Last[
           Reap[
            Replace[
             Import[file, "Table"],
             {x_, y_, z_, polarAngle_,
               eccentricity_, fstatAngle_, fstatEccen_} :> (
               If[fstatAngle ≥ $FStatCutoff,
                Sow[
                 Append[{x, y, z}, polarAngle], "PolarAngle"]];
               If[fstatEccen > $FStatCutoff,
                Sow[
                 Append[{x, y, z}, eccentricity], "Eccentricity"]]),
             {1}],
            {"PolarAngle", "Eccentricity"}]],
          1]]},
      (* First, read in the left *)
      {import[lfl], import[rfl]}]],

    (* This dataset has RH data that goes from -180→0,
    so we correct for this *)
    ReverseRightHemisphere → False,

    (* Selections for fitting *)
    EccentricitySelect → (InV1Q[#] && 2.5 ≤ #[[3]] ≤ 18.0 &&
        EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &),
    (* It seems that the fit qualities are better when we
     include extrastriate regions for polar angle;
    this seems to be because some of the subjects have splotchy data
     right inside the V1 ellipse, leaving poor extrema to fit the model. *)
    PolarAngleSelect → (InV1Q[#] &&
        EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &),
```

```
(* Other details *)
StimulusRange → {{0, 180}, {0, 20}},
IdenticalVertices → False]];
```

- **Declare Repeat-run Dataset**

This dataset is for a subject who did several runs of 16 minutes each; we use him to examing how MRI quality falls off with time in the scanner.

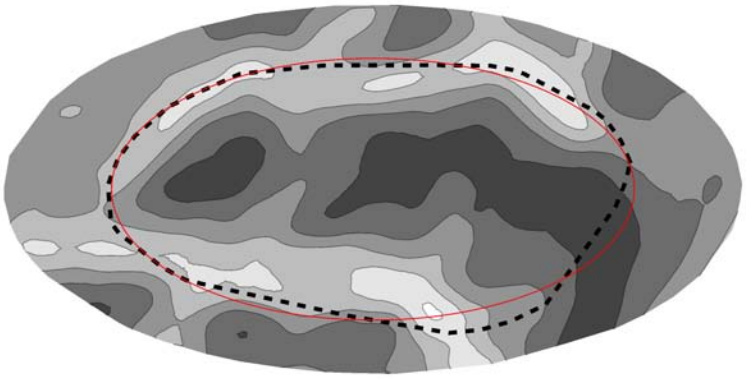
```
With[{$FStatCutoff = 8},
  dataLR = DeclareDataset[
    "Our single subject with long repeated runs; retinotopy measured out to 10°",

    (* We find our subjects from a file... *)
    Import[
     $TemplateDirectory <> "/subjects/longrun/subjects-longrun.txt", "Words"],

    (* The function that, given a subject id, loads the data *)
    Function[{id},
     Module[
      {lfl = $TemplateDirectory <> "/subjects/longrun/" <> id <> "-lh.dat",
       rfl = $TemplateDirectory <> "/subjects/longrun/" <> id <> "-rh.dat",
       (* Here's how we collect a hemisphere's data... *)
       import = Function[{file},
         Flatten[
          Last[
           Reap[
            Replace[
             Import[file, "Table"],
             {x_, y_, z_, polarAngle_, eccentricity_, fstat_} :> (
               If[fstat ≥ $FStatCutoff,
                Sow[
                 Append[{x, y, z}, polarAngle], "PolarAngle"];
                Sow[
                 Append[{x, y, z}, eccentricity], "Eccentricity"]]),
             {1}],
            {"PolarAngle", "Eccentricity"}]],
          1]]},
      (* First, read in the left *)
      {import[lfl], import[rfl]}]],

    (* This dataset has RH data that goes from -180→0,
    so we correct for this *)
    ReverseRightHemisphere → True,

    (* Selections for fitting *)
    EccentricitySelect → (InV1Q[#] && 2.5 ≤ #[[3]] ≤ 8.0 &&
        EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &),
    PolarAngleSelect → (InV1Q[#] && EccentricityCoordinate[
          #[[1]], #[[2]]] < 0.75 &),

    (* Other details *)
    StimulusRange → {{0, 180}, {0, 10}},
    IdenticalVertices → True]];
```

Images

Anatomical Plot

Here, we create a plot of the cortical surface anatomy with the Hinds V1 border in a dotted black line and the V1 ellipse in red.

```
Module[
 {img = Show[
    {ListContourPlot[
      Select[Anatomy, InV1DialatedQ],
      ColorFunction → (GrayLevel[1 - 0.8 * #] &),
      AspectRatio → 0.5,
      Frame → None,
      Axes → None],
     ListPlot[
      V1Hull,
      PlotStyle → {Thick, Black, Dashed},
      Joined → True],
     Plot[
      #[[2]] & /@ List @@ Quiet[Reduce[V1EllipseEquation, y]],
      {x, -V1EllipseA, V1EllipseA},
      PlotStyle → {Red, Red}]}]},
 FigureExport["anatomy.pdf", img, "PDF", ImageResolution → 600];
 If[$ShowFigures, img]]
```

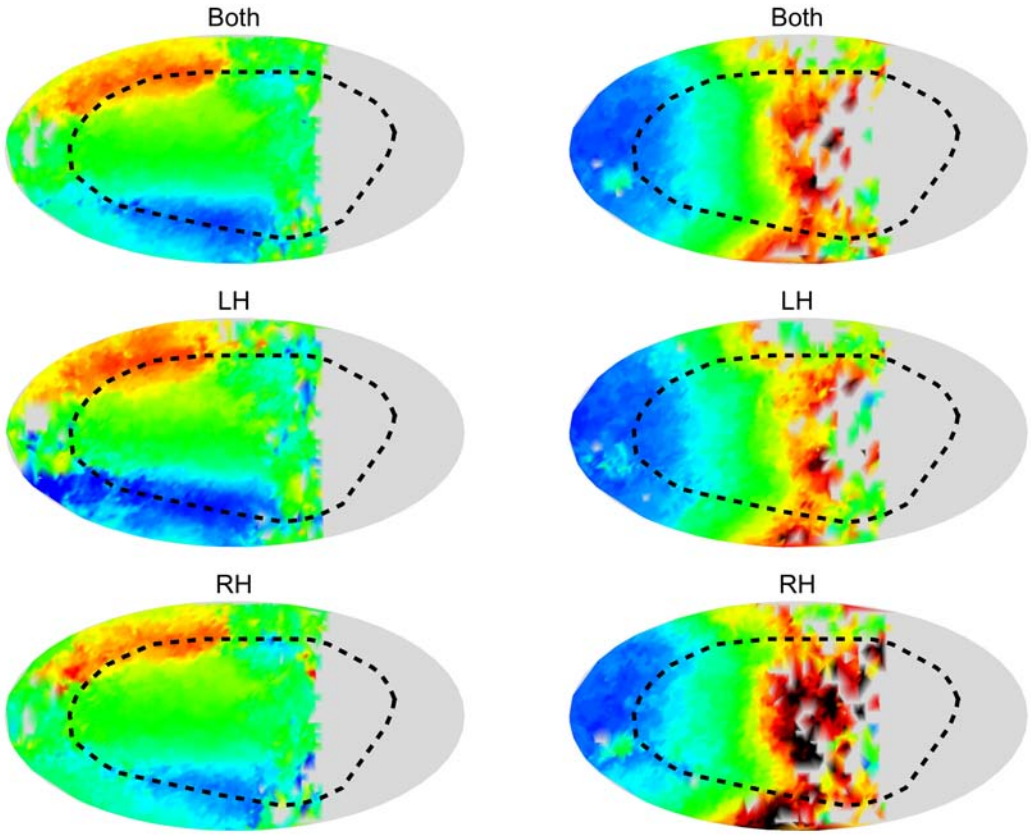


Aggregate Plots

Aggregates of polar angle and eccentricity. We haven't plotted these per hemisphere, but the results look remarkably similar.

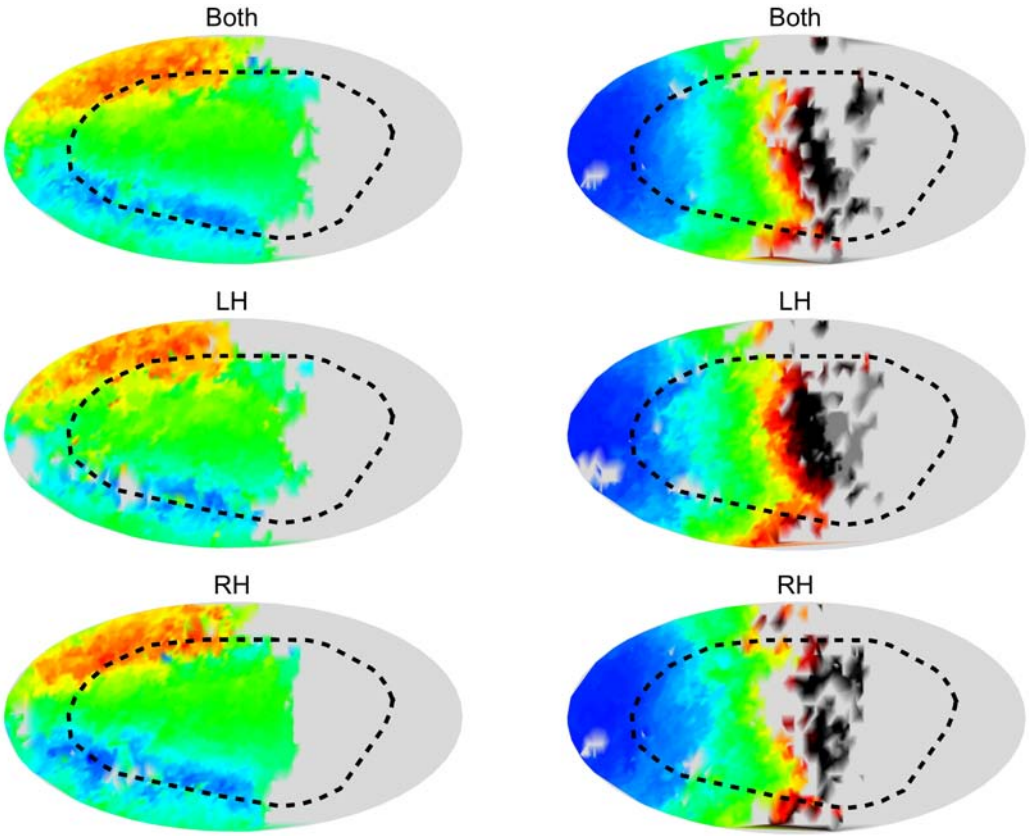
```
tmp = With[{data = data10, type = PolarAngle},
   Module[
    {binnedData =
      Transpose[(* Here, we collect the points into individual bins *)
       Map[
        Flatten[#, 1] &,
        BinLists[
         Select[
          Flatten[data[type], 2],
          InV1DialatedQ],
         {Table[x, {x, -V1DialatedEllipseA, V1DialatedEllipseA, 0.005}]},
         {Table[y, {y, -V1DialatedEllipseB, V1DialatedEllipseB, 0.005}]},
         {{-0.01, 180.01}}],
        {2}]]},
    {binnedData,
     Reap[
      Scan[
       If[Length@# > 2 &&
          StandardDeviation[#[[All, 3]]] < If[type === PolarAngle, 60, 3.3],
         Sow[{Mean[#[[All, 1]]], Mean[#[[All, 2]]], Median[#[[All, 3]]]}]] &,
       binnedData,
       {2}]
      ][[2, 1]]}]];
```

```
plotAggregates = Function[{data, exportName},
   Module[
    {img = GraphicsGrid[
       Map[
        Show[
          {#,
           ListPlot[V1Hull, PlotStyle → {Thick, Black, Dashed}, Joined → True]}] &,
        Map[
         {BrainPlot[
            Select[
             If[# === None,
              data[Aggregate, PolarAngle],
              data[Aggregate, #, PolarAngle]],
             EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &],
            PlotRange → {0, 180},
            Filling → LightGray,
            PlotLabel → Style[
              Which[# === LeftHemisphere,
               "LH", # === RightHemisphere, "RH", True, "Both"],
              FontFamily → "Arial",
              FontSize → 12]],
           BrainPlot[
            Select[
             If[# === None,
              data[Aggregate, Eccentricity],
              data[Aggregate, #, Eccentricity]],
             EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &],
            PlotRange → {0, 20},
            ColorFunction → Function[{v},
              Blend[
               Join[
                $ColorScaleColors,
                Take[
                 Flatten[
                  {Table[Black, {Floor[(Length[$ColorScaleColors]) / 2]}],
                   Table[Gray, {Floor[(Length[$ColorScaleColors]) / 2]}]}],
                 Length[$ColorScaleColors] - 1]],
               v]],
            Filling → LightGray,
            PlotLabel → Style[
              Which[# === LeftHemisphere,
               "LH", # === RightHemisphere, "RH", True, "Both"],
              FontFamily → "Arial",
              FontSize → 12]]} &,
         {None, LeftHemisphere, RightHemisphere}],
        {2}]]},
     FigureExport[exportName <> ".pdf", img, "PDF", ImageResolution → 600];
     If[$ShowFigures, img]]];
plotAggregates[data10, "aggregates10"]
```



This is the same for dataset 20

```
plotAggregates[data20, "aggregates20"]
```

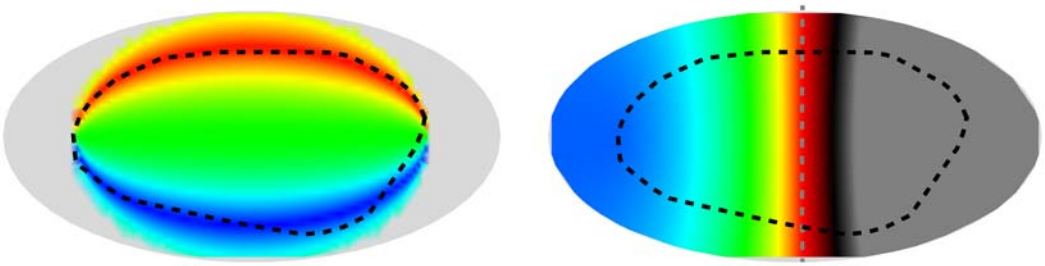


Template Plots

This code will produce plots of the template predictions; in the case of Polar Angle, it projects the template beyond the elliptical borders a bit.

```
Function[{data},
  Module[
   {img = GraphicsRow[
      Map[
       Show[
        {#,
         ListPlot[V1Hull, PlotStyle → {Thick, Black, Dashed}, Joined → True]}] &,
       {BrainPlot[
         Map[
          Module[{pacoord = PolarAngleCoordinate[#[[1]], #[[2]]]},
           Append[#,
            Which[
             Abs[pacoord] ≤ 1, data[Template, PolarAngle][#[[1]], #[[2]]],
             pacoord > 1, data[Template, PolarAngle][
              0,
              V1EllipseB * (2.0 - pacoord)],
             pacoord < -1, data[Template, PolarAngle][
              0,
```

```
            V1EllipseB * (-2.0 - pacoord)]]]] &,
        Select[
         Flatten[
          Table[{x, y},
           {x, -V1DialatedEllipseA, V1DialatedEllipseA, 0.01},
           {y, -V1DialatedEllipseB, V1DialatedEllipseB, 0.01}],
          1],
         And[
           InV1DialatedQ[#],
           Abs[PolarAngleCoordinate[#[[1]], #[[2]]]] ≤ Sqrt[2.0],
           Abs[#[[1]]] < V1EllipseA] &]],
       PlotRange → {0, 180}],
      (* Eccentricity template plot... *)
      Show[
       {BrainPlot[
         Map[
          Append[#, data[Template, Eccentricity][#[[1]], #[[2]]]] &,
          Select[
           Flatten[
            Table[{x, y},
             {x, -V1DialatedEllipseA, V1DialatedEllipseA, 0.01},
             {y, -V1DialatedEllipseB, V1DialatedEllipseB, 0.01}],
            1],
           InV1DialatedQ]],
         PlotRange → {0, 20},
         ColorFunction → (If[# ≤ 0.5,
             Blend[$ColorScaleColors, 2 * #],
             Blend[{Last@$ColorScaleColors, Black, Gray}, 2 * (# - 0.5)]] &)],
        (* Add a gray dashed line to mark the 10° boundary *)
        ListPlot[
         Sort[
          Block[{x, y},
           Module[
            {e0 =
              FindRoot[data[Template, Eccentricity][x, 0] == 10.0, {x, -0.05}],
             x0},
            x0 = EccentricityCoordinate[x /. e0, 0];
            Select[
             Flatten[
              Table[
               Map[
                {x, y /. #} &,
                Solve[EccentricityCoordinate[x, y] == x0, y]],
               {x, e0[[1, 2]] - 0.06, e0[[1, 2]] + 0.06, 0.001}],
              1],
             Length@# == 2 && Im[#[[2]]] == 0 &]]],
          #1[[2]] < #2[[2]] &],
         PlotStyle → {Gray, Thick, Dashed},
         Joined → True]}]}]],
   FigureExport["templates.pdf", img, "PDF", ImageResolution → 600];
   If[$ShowFigures, img]]
 ]@data10
```

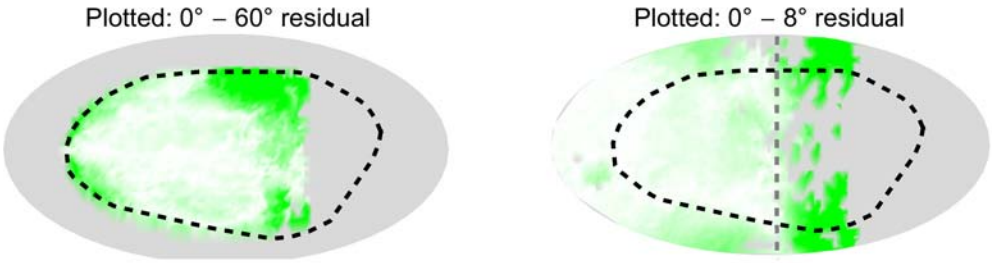


V1 Residual Plots

This code will produce plots of the residual error of the template to the aggregate mapped onto V1.

```
Function[{data},
  Module[
   {img = GraphicsRow[
      Map[
       Show[
         {#,
          ListPlot[V1Hull, PlotStyle → {Thick, Black, Dashed}, Joined → True]}] &,
       {BrainPlot[
         Map[
          {#[[1]], #[[2]],
            Abs[#[[3]] - data[Template, PolarAngle][#[[1]], #[[2]]]]} &,
          Select[
           data[Aggregate, PolarAngle],
           InV1Q[#] && EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &]],
         PlotRange → {0, 60},
         PlotLabel → Style[
           "Plotted: 0° - 60° residual",
           FontFamily → "Arial",
           FontSize → 12],
         ColorFunction → (Blend[{White, Green}, #] &)],
        Show[
         {BrainPlot[
           Map[
            {#[[1]], #[[2]],
              Abs[#[[3]] - data[Template, Eccentricity][#[[1]], #[[2]]]]} &,
            Select[
             data[Aggregate, Eccentricity],
             InV1DialatedQ[#] &&
               EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &]],
           PlotRange → {0, 8},
           PlotLabel → Style[
             "Plotted: 0° - 8° residual",
             FontFamily → "Arial",
             FontSize → 12],
           ColorFunction → (Blend[{White, Green}, #] &)],
          ListPlot[
```

```
            Sort[
             Block[{x, y},
              Module[
               {e0 =
                 FindRoot[data[Template, Eccentricity][x, 0] == 10.0, {x, -0.05}],
                x0},
               x0 = EccentricityCoordinate[x /. e0, 0];
               Select[
                Flatten[
                 Table[
                  Map[
                   {x, y /. #} &,
                   Solve[EccentricityCoordinate[x, y] == x0, y]],
                  {x, e0[[1, 2]] - 0.06, e0[[1, 2]] + 0.06, 0.001}],
                 1],
                Length@# == 2 && Im[#[[2]]] == 0 &]]],
              #1[[2]] < #2[[2]] &],
             PlotStyle → {Gray, Thick, Dashed},
             Joined → True]}]}]]},
      FigureExport["vertex-residuals.pdf", img, "PDF", ImageResolution → 600];
      If[$ShowFigures, img]]
    ]@data10
```



■ Prediction Accuracy Plots

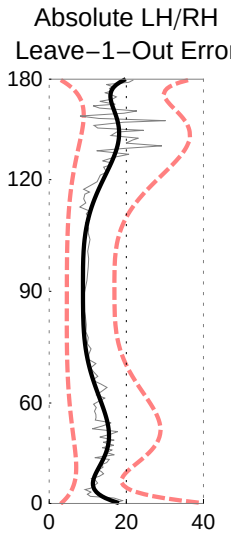
These plots show the accuracy of the template predictions as a function of polar angle and eccentricity.

To do this, we collect the eccentricity or polar angle coordinate for each vertex as well as it's residual to the vertex's subject's leave-one-out template. We then plot the median and quartiles for these with a best fit line through them.

```
Function[{data, binsz},
  Module[
   {errors = Flatten[
      MapThread[
       Function[{dat, tmp},
        Map[
         {#[[1]], #[[2]], Abs[#[[3]] - tmp[#[[1]], #[[2]]]]} &,
         Select[
```

```
        dat,
        data[PolarAngleSelect][#] &&
          2.5 ≤ data[Template, Eccentricity][#[[1]], #[[2]]] < 8 &]],
     {Flatten[data[PolarAngle], 1],
      Flatten[
       Transpose[
        {data10[LeaveOneOut, PolarAngle, LeftHemisphere],
         data10[LeaveOneOut, PolarAngle, RightHemisphere]}],
       1]}],
    1],
 medianErr, lowerErr, upperErr, medianFit,
 img},
{lowerErr, medianErr, upperErr} = Map[
  Flatten[#, 1] &,
  Reap[
    Scan[
     If[Length@# > 0,
       (Sow[{Mean[#[[All, 1]]], Mean[#[[All, 3]]]}, 2];
        Sow[{Mean[#[[All, 1]]], Mean[#[[All, 2]]]}, 1];
        Sow[{Mean[#[[All, 1]]], Mean[#[[All, 4]]]}, 3];)] &,
     Flatten[
      BinLists[
       Map[{#[[1]], #[[2, 1]], #[[2, 2]], #[[2, 3]]} &,
        Reap[
          Scan[
           If[Length@# ≥ 3,
             Sow[
              {data[Template, PolarAngle][
                Mean[#[[All, 1]]], Mean[#[[All, 2]]]],
               Quartiles[#[[All, 3]]]}]] &,
           Map[
            Flatten[#, 1] &,
            BinLists[
             errors,
             {Table[x, {x, -V1DialatedEllipseA, V1DialatedEllipseA, 0.005}]},
             {Table[y, {y, -V1DialatedEllipseB, V1DialatedEllipseB, 0.005}]},
             {{-0.01, 180.01}}],
            {2}],
           {2}]
         ][[2, 1]]],
       binsz,
       1000, 1000, 1000],
      3]],
     {1, 2, 3}
    ][[2]]];
(* We want to convert lower and upper error
 into best fits as well as make a best fit for the median *)
{lowerErr, medianFit, upperErr} = Map[
  Function[{dat},
   Module[
    {fit = Block[{x}, LinearModelFit[dat, {x, x^2, x^3, x^4, x^5, x^6}, x]]},
```

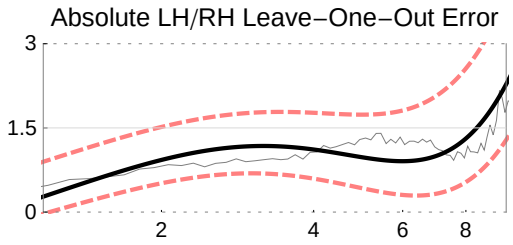
```
      Map[{#[[1]], fit[#[[1]]]} &, dat]]],
    {lowerErr, medianErr, upperErr}];
  img = ListPlot[
    Map[
     ReplaceAll[
      #,
      {pa_, err_} :> {err, FindRoot[data[Template, PolarAngle][0, y] == pa,
           {y, 0.2}][[1, 2]] / V1EllipseB}] &,
     {lowerErr, medianErr, upperErr, medianFit}],
    LabelStyle -> Directive[10, FontFamily -> "Arial"],
    ImageSize -> 1.5 * 72,
    PlotLabel -> Style[
      "Absolute LH/RH\nLeave-1-Out Error",
      FontFamily -> "Arial",
      FontSize -> 12],
    Joined -> True,
    AxesOrigin -> {20, -2},
    AxesStyle -> {LightGray, Dotted},
    Frame -> True,
    FrameStyle -> {{{Dotted, Gray}, {Dotted, Gray}}, {Gray, Gray}},
    FrameTicks -> {
      {Prepend[
        Table[{FindRoot[data[Template, PolarAngle][0, y] == t, {y, 0.2}][[1, 2]] /
           V1EllipseB, t},
         {t, {60, 90, 120, 180}}],
        {-1, 0}], None},
      {{0, 20, 40}, None}},
    PlotRange -> {{0, 40}, {-1, 1}},
    PlotStyle -> {{Pink, Thick, Dashed}, {Gray, Dashing[{}], Thin},
      {Pink, Thick, Dashed}, {Thick, Dashing[{}], Black}},
    FrameTicksStyle -> Black,
    AspectRatio -> 2.75];
  FigureExport["pa-leave-one-out-error.pdf", img, "PDF", ImageResolution -> 600];
  If[$ShowFigures, img]
][data10, 1.5]
```




```
Function[{data, binsz},
  Module[
   {errors = Flatten[
      MapThread[
       Function[{dat, tmp},
        Map[
         {#[[1]], #[[2]], Abs[#[[3]] - tmp[#[[1]], #[[2]]]]} &,
         Select[
          dat,
          InV1Q[#] &&
            EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 && #[[3]] < 10.0 &]]],
       {Flatten[data[Eccentricity], 1],
        Flatten[
         Transpose[
          {data[LeaveOneOut, Eccentricity, LeftHemisphere],
           data[LeaveOneOut, Eccentricity, RightHemisphere]}],
         1]}],
      1],
    medianErr, lowerErr, upperErr, medianFit,
    minecc = data[Template, Eccentricity][-V1EllipseA, 0],
    img},
   {lowerErr, medianErr, upperErr} = Map[
     Flatten[#, 1] &,
     Reap[
       Scan[
        If[Length@# > 0,
          (Sow[{Mean[#[[All, 1]]], Mean[#[[All, 3]]]}, 2];
           Sow[{Mean[#[[All, 1]]], Mean[#[[All, 2]]]}, 1];
           Sow[{Mean[#[[All, 1]]], Mean[#[[All, 4]]]}, 3];)] &,
        Flatten[
         BinLists[
          Map[{#[[1]], #[[2, 1]], #[[2, 2]], #[[2, 3]]} &,
           Reap[
             Scan[
              If[Length@# ≥ 3,
```

```
              Sow[
               {data[Template, Eccentricity][
                 Mean[#[[All, 1]]], Mean[#[[All, 2]]]],
                Quartiles[#[[All, 3]]]}] &,
             Map[
              Flatten[#, 1] &,
              BinLists[
               errors,
               {Table[x,
                 {x, -V1DialatedEllipseA, V1DialatedEllipseA, 0.0025}]},
               {Table[y, {y, -V1DialatedEllipseB, V1DialatedEllipseB,
                  0.0025}]},
               {{-0.01, 180.01}}],
              {2}],
             {2}]
           ][[2, 1]]],
       binsz,
       1000, 1000, 1000],
      3]],
    {1, 2, 3}
   ][[2]]];
(* We want to convert lower and upper error
 into best fits as well as make a best fit for the median *)
{lowerErr, medianFit, upperErr} = Map[
  Function[{dat},
   Module[
    {fit = Block[{x}, LinearModelFit[dat, {x, x^2, x^3, x^4, x^5, x^6}, x]]},
    Map[{#, fit[#]} &, Range[0, 12, 0.05]]]],
  {lowerErr, medianErr, upperErr}];
img = ListPlot[
  Map[
   ReplaceAll[
     #,
     {ecc_, err_} :> {
       If[ecc ≤ minecc, -V1EllipseA,
        (FindRoot[data[Template, Eccentricity][x, 0] == ecc, {x, -0.25},
              AccuracyGoal → 4][[1, 2]] + V1EllipseA) / (2 * V1EllipseA)],
       err}] &,
   {lowerErr, medianErr, upperErr, medianFit}],
  LabelStyle → Directive[10, FontFamily → "Arial"],
  PlotLabel → Style[
    "Absolute LH/RH Leave-One-Out Error",
    FontFamily → "Arial",
    FontSize → 12],
  ImageSize → 3.25 * 72,
  Joined → True,
  AxesOrigin → {-2, 1.5},
  AxesStyle → {LightGray, Dotted},
  Frame → True,
  FrameStyle → {{Gray, Gray}, {{Dotted, Gray}, {Dotted, Gray}}},
  FrameTicks → {{{0, 1.5, 3}, None},
```

```
    {Table[{(FindRoot[data[Template, Eccentricity][x, 0] == t, {x, -0.25},
              AccuracyGoal → 4][[1, 2]] + V1EllipseA) / (2 * V1EllipseA), t},
      {t, {2, 4, 6, 8, 10}}], None}},
   PlotRange → {{0.05, 0.5109}, {0, 3}},
   PlotStyle → {{Pink, Thick, Dashed}, {Gray, Dashing[{}], Thin},
     {Pink, Thick, Dashed}, {Thick, Dashing[{}], Black}},
   FrameTicksStyle → Black,
   AspectRatio → 1 / 2.75];
 FigureExport[
  "ecc-leave-one-out-error.pdf", img, "PDF", ImageResolution → 600];
 If[$ShowFigures, img]]
][data10, 0.125]
```



- **Exact Mean and Median Leave-one-out Errors**

Here, we can calculate (and store) the exact mean leave-one-out error over our dataset.

```
LeaveOneOutError[datasets_List, hemi : (None | LeftHemisphere | RightHemisphere),
   type : (Eccentricity | PolarAngle),
   OptionsPattern[]] := Module[
   {collect = Transpose[
      Flatten[
       Transpose[
        Map[
         If[hemi === None,
          {Flatten[Transpose[#[type]], 1],
            Join[#[LeaveOneOut, type, LeftHemisphere],
             #[LeaveOneOut, type, RightHemisphere]],
            Table[#, {2 * Length@#[Subjects]}]} &,
          {#[type, hemi],
            #[LeaveOneOut, type, hemi],
            Table[#, {Length@#[Subjects]}]} &],
         datasets]],
       1]],
    signed = OptionValue[Sign],
    tmp},
   collect = MapThread[
     Function[{pts, tmpl, dataset},
      Map[
       {#[[3]], tmpl[#[[1]], #[[2]]]} &,
       Select[
        pts,
        If[type === PolarAngle,
         dataset[PolarAngleSelect][#] &&
           2.5 ≤ dataset[Template, Eccentricity][#[[1]], #[[2]]] ≤ 8 &,
         dataset[EccentricitySelect][#] && 2.5 ≤ dataset[Template,
              Eccentricity][#[[1]], #[[2]]] ≤ 8 &]]]],
     Transpose[collect]];
   tmp = Flatten[collect, 1];
   (* here we deal with offdiag and sign if necessary *)
   tmp = If[signed,
     Map[(#[[1]] - #[[2]]) &, tmp],
     Map[Abs[#[[1]] - #[[2]]] &, tmp]];
   {Mean → Mean[tmp], Median → Median[tmp], Data → collect}];
Options[LeaveOneOutError] = {Sign → False};
```

We can calculate (and store) these for any dataset or combination of them

```
Module[{tmp = LeaveOneOutError[{data10}, None, PolarAngle]},
 data10["MeanLOOErrorPA"] = (Mean /. tmp);
 data10["MedianLOOErrorPA"] = (Median /. tmp);
 {"Both" → tmp[[1 ;; 2]],
  "LH" → LeaveOneOutError[{data10}, LeftHemisphere, PolarAngle][[1 ;; 2]],
  "RH" → LeaveOneOutError[{data10}, RightHemisphere, PolarAngle][[1 ;; 2]]}]
```

```
{Both → {Mean → 16.7496, Median → 11.431},
 LH → {Mean → 17.5271, Median → 12.3772}, RH → {Mean → 15.9721, Median → 10.3236}}
```

```
Module[{tmp = LeaveOneOutError[{data10}, None, Eccentricity]},
 data10["MeanLOOErrorEcc"] = (Mean /. tmp);
 data10["MedianLOOErrorEcc"] = (Median /. tmp);
 {"Both" → tmp[[1 ;; 2]],
  "LH" → LeaveOneOutError[{data10}, LeftHemisphere, Eccentricity][[1 ;; 2]],
  "RH" → LeaveOneOutError[{data10}, RightHemisphere, Eccentricity][[1 ;; 2]]}]
```

```
{Both → {Mean → 1.12146, Median → 0.907442},
 LH → {Mean → 1.09886, Median → 0.92564}, RH → {Mean → 1.14936, Median → 0.883037}}
```

The signed versions of the same measurements...

```
Module[{tmp = LeaveOneOutError[{data10}, None, PolarAngle, Sign → True]},
 data10["MeanSignedLOOErrorPA"] = (Mean /. tmp);
 data10["MedianSignedLOOErrorPA"] = (Median /. tmp);
 {"Both" → tmp[[1 ;; 2]],
  "LH" →
   LeaveOneOutError[{data10}, LeftHemisphere, PolarAngle, Sign → True][[1 ;; 2]],
  "RH" → LeaveOneOutError[{data10}, RightHemisphere,
     PolarAngle, Sign → True][[1 ;; 2]]}]
```

```
{Both → {Mean → -3.63991, Median → -0.931891},
 LH → {Mean → -4.55596, Median → -1.70121},
 RH → {Mean → -2.72393, Median → -0.423031}}
```

```
Module[{tmp = LeaveOneOutError[{data10}, None, Eccentricity, Sign → True]},
 data10["MeanSignedLOOErrorEcc"] = (Mean /. tmp);
 data10["MedianSignedLOOErrorEcc"] = (Median /. tmp);
 {"Both" → tmp[[1 ;; 2]],
  "LH" →
   LeaveOneOutError[{data10}, LeftHemisphere, Eccentricity, Sign → True][[1 ;; 2]],
  "RH" → LeaveOneOutError[{data10}, RightHemisphere,
     Eccentricity, Sign → True][[1 ;; 2]]}]
```

```
{Both → {Mean → 0.367013, Median → 0.394207},
 LH → {Mean → 0.514878, Median → 0.50933}, RH → {Mean → 0.184436, Median → 0.244406}}
```

These two blocks will calculate the median error range across subjects (i.e., min and max over all subjects of the median absolute error of all vertices in a single subject)

```
Module[
  {dat = Table[
     Map[
      Median[Abs[#[[All, 1]] - #[[All, 2]]]] &,
      LeaveOneOutError[{data10}, type, PolarAngle][[3, 2]]],
     {type, {LeftHemisphere, RightHemisphere, None}}]},
  Prepend[
   MapThread[
    {#2, Min[#1], Max[#1]} &,
    {dat,
     {"LH", "RH", "BH"}}],
   {"\\", "Min", "Max"}]] // MatrixForm
```

\	Min	Max
LH	5.48582	23.3434
RH	4.78199	33.8342
BH	4.78199	33.8342

```
Module[
  {dat = Table[
     Map[
      Median[Abs[#[[All, 1]] - #[[All, 2]]]] &,
      LeaveOneOutError[{data10}, type, Eccentricity][[3, 2]]],
     {type, {LeftHemisphere, RightHemisphere, None}}]},
  Prepend[
   MapThread[
    {#2, Min[#1], Max[#1]} &,
    {dat,
     {"LH", "RH", "BH"}}],
   {"\\", "Min", "Max"}]] // MatrixForm
```

\	Min	Max
LH	0.603633	1.5603
RH	0.461047	2.67835
BH	0.461047	2.67835

■ Repeated-runs/Time-per-accuracy Plot

We want to look at how 16, 32, and 48 minuts of scanning compares to 48 minuts of separate scanning; to do so, we compare all unique tupes of size 3 against all unique disjoint tuples of size 3, 2, and 1 scans (each scan is 16 minutes).

```
rrSets = Module[{subs = Range[Length@dataLR[Subjects]]},
   Split[
    Flatten[
     Map[
      Function[{s},
       Map[{s[[1]], #} &, s[[2]]]],
      Flatten[
       Table[
        Map[{#, Subsets[Complement[subs, #], {3}]} &, Subsets[subs, {k}]],
        {k, 1, 3}],
       1]],
     1],
    Length[#1[[1]]] == Length[#2[[1]]] &]];
```

These sets can then be used to pick out the residuals that you get from comparing the aggregate of one group of to the other. This looks a little complex because it's mapped across polar angle and eccentricity, but really it's just a transformation of {set1, set2} -> residuals you get when aggregate-of-set1 is compared to aggregate-of-set2

```
rrResiduals = Map[
   (* This just reorganizes the data so that first list level is the time in
    scanner, second list level is pa/ecc, and third list level is hemisphere *)
   {{#[[All, 1, 1]], #[[All, 2, 1]]}, {#[[All, 1, 2]], #[[All, 2, 2]]}} &,
   Map[
    Function[{rrCmp},
     Module[
      {d1 = Map[
         Map[
           Select[#, InV1Q] &,
           dataLR[#][Data],
           {2}] &,
         rrCmp[[1]]],
       d2 = Map[
         Map[
           Select[#, InV1Q] &,
           dataLR[#][Data],
           {2}] &,
         rrCmp[[2]]]},
      Table[
       Select[
        AggregateSubjects[
         {AggregateSubjects[d1[[All, i, j]], Mean[#] &],
          AggregateSubjects[d2[[All, i, j]], -Mean[#] &]},
         If[Length@# ≠ 2 || Sign[#[[1]]] == Sign[#[[2]]],
           Delete,
           If[#[[1]] > 0,
            {#[[1]], -#[[2]]},
            {#[[2]], -#[[1]]}]] &],
        #[[3]] =!= Delete &],
       {i, 1, 2},
       {j, 1, 2}]]],
    rrSets,
    {2}]];
```

And make a plot...

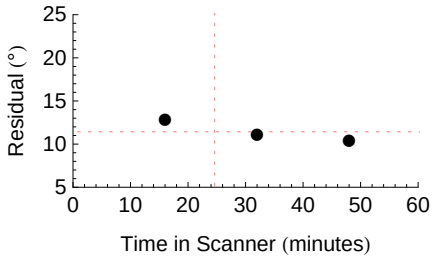
```
(* aggfun should be Mean or Median *)
Function[{data, aggfun},
  Module[
   {rrStats = Transpose[
      Map[
       (* This will map over the ecc/pa sets for each scan length *)
       Module[{dat = Flatten[#[[All, All, All, 3]], 2], res},
         res = dat[[All, 1]] - dat[[All, 2]];
         {aggfun@Abs@res, StandardDeviation@Abs@res, Length@res}] &,
       rrResiduals,
       {2}]],
    datPA =
     If[aggfun === Median, data["MedianLOOErrorPA"], data["MeanLOOErrorPA"]],
    datEc = If[aggfun === Median, data["MedianLOOErrorEcc"],
      data["MeanLOOErrorEcc"]],
```

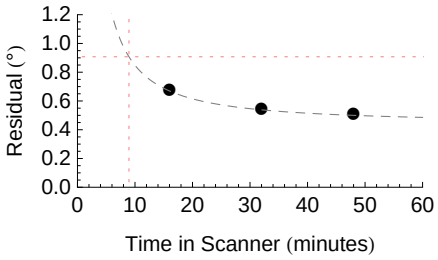
```
 img},
img = GraphicsRow[
  {Module[
    {dat = MapThread[
       {{#1, #2[[1]]}, ErrorBar[Sqrt[#2[[2]]^2 / #2[[3]]]]} &,
       {{16, 32, 48}, rrStats[[1]]}],
     mdl, pamin},
    mdl = Block[{x, c, k, k0},
      (*LinearModelFit[
       {{20,dat[[1,1,2]]},{40,dat[[2,1,2]]},{60,dat[[3,1,2]]}},
       {x},
       x]*)
      NonlinearModelFit[
       {{16, dat[[1, 1, 2]]}, {32, dat[[2, 1, 2]]}, {48, dat[[3, 1, 2]]}},
       {k0 + k * x^(-c), c > 0.001 && k0 > 0 && k > 0.001},
       {{k0, 1}, {c, 1}, {k, 1}},
       x,
       Method → "NMinimize"]];
    tmp = mdl;
    Print[mdl["BestFitParameters"]];
    pamin = Quiet@FindRoot[
        mdl[x] == datPA,
        {x, 10.0}
       ][[1, 2]];
    Show[
     {ErrorListPlot[
       dat,
       ImageSize → 3.25 * 72,
       PlotRange → {{0, 60}, {5, 25}},
       PlotMarkers → {{●, Small}},
       PlotStyle → Black,
       LabelStyle → Directive[10, FontFamily → "Arial"],
       Axes → None,
       AspectRatio → 0.5,
       Frame → {{True, False}, {True, False}},
       FrameLabel →
        {{"Residual (°)", None}, {"Time in Scanner (minutes)", None}},
       Prolog → {Pink, Dotted,
         Line[{{pamin, -1}, {pamin, 400}}],
         Line[{{-1, datPA}, {400, datPA}}]}],
      Plot[mdl[x], {x, -180, 180},
       PlotStyle → {Gray, Dashed},
       PlotRange → Full]}]],
   Module[
    {dat = MapThread[
       {{#1, #2[[1]]}, ErrorBar[Sqrt[#2[[2]]^2 / #2[[3]]]]} &,
       {{16, 32, 48},
        rrStats[[2]]}],
     mdl, eccmin},
    mdl = Block[{x, c, k, k0},
      (*LinearModelFit[
```

```
        {{20,dat[[1,1,2]]},{40,dat[[2,1,2]]},{60,dat[[3,1,2]]}},
        {x},
        x]*)
       NonlinearModelFit[
        {{16, dat[[1, 1, 2]]}, {32, dat[[2, 1, 2]]}, {48, dat[[3, 1, 2]]}},
        {k0 + k * x^(-c), c > 0.001 && k0 > 0 && k > 0.001},
        {{k0, 1}, {c, 1}, {k, 1}},
        x,
        Method → "NMinimize"]];
      Print[mdl["BestFitParameters"]];
      eccmin = FindRoot[
        mdl[x] == datEc,
        {x, 10.0}
       ][[1, 2]];
      Show[
       {ErrorListPlot[
        dat,
        ImageSize → 3.25 * 72,
        PlotRange → {{0, 60}, {0, 1.2}},
        PlotMarkers → {{●, Small}},
        PlotStyle → Black,
        LabelStyle → Directive[10, FontFamily → "Arial"],
        Axes → None,
        AspectRatio → 0.5,
        Frame → {{True, False}, {True, False}},
        FrameLabel →
         {{"Residual (°)", None}, {"Time in Scanner (minutes)", None}},
        Prolog → {Pink, Dotted,
          Line[{{eccmin, -1}, {eccmin, 400}}],
          Line[{{-1, datEc}, {400, datEc}}]}],
       Plot[Evaluate[mdl["BestFit"]], {x, 0.1, 60},
        PlotStyle → {Gray, Dashed},
        PlotRange → {{0, 60}, Full},
        Exclusions → None]}]]}];
    FigureExport[
     If[aggfun === Median, "sj-summary-median.pdf", "sj-summary-mean.pdf"],
     img,
     "PDF",
     ImageResolution → 600];
    If[$ShowFigures, img]]
  ][data10, Median]
```

```
{k0 → 8.54939, c → 0.803827, k → 37.8527}
```

```
{k0 → 0.440559, c → 1.22744, k → 6.88198}
```





■ Template in terms of Normalized V1 Distance

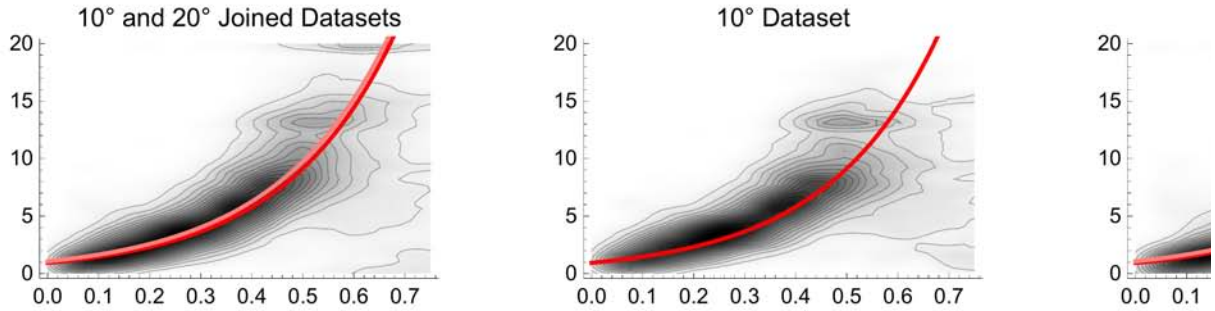
Here, we want to make a plot of the template in terms of the eccentricity coordinate (or a normalized version of V1 length, basically). We make this with the aggregate eccentricity data of both dataset 10° and dataset 20°.

```
plotEccHist = Function[{hemi, name},
   Module[
    {img = MapThread[
       Show[
         {#1,
          ListPlot[
           Evaluate[
            If[hemi === None,
             If[#2,
              {Table[{x, data10[Template, Eccentricity][
                   V1EllipseA * 2 * x - V1EllipseA, 0]}, {x, 0, 1, 0.01}],
               Table[{x, data20[Template, Eccentricity][V1EllipseA * 2 * x -
                    V1EllipseA, 0]}, {x, 0, 1, 0.01}]},
              Table[{x, data10[Template, Eccentricity][V1EllipseA * 2 * x -
                   V1EllipseA, 0]}, {x, 0, 1, 0.01}]],
             If[#2,
              {Table[{x, data10[Template, hemi, Eccentricity][
                   V1EllipseA * 2 * x - V1EllipseA, 0]}, {x, 0, 1, 0.01}],
               Table[{x, data20[Template, hemi, Eccentricity][
                   V1EllipseA * 2 * x - V1EllipseA, 0]}, {x, 0, 1, 0.01}]},
              Table[{x, data10[Template, hemi, Eccentricity][
                  V1EllipseA * 2 * x - V1EllipseA, 0]}, {x, 0, 1, 0.01}]]]],
           PlotRange → {{0, 0.75}, {0, 20}},
           Joined → True,
           PlotStyle → If[#2, {{Thick, Red}, {Thick, Pink}}, {Thick, Red}],
           ImageSize → 3.25 * 72,
           LabelStyle → Directive[10, FontFamily → "Arial"],
           Axes → None,
           AspectRatio → 0.6,
           Frame → {{True, False}, {True, False}}]}] &,
       {{SmoothDensityHistogram[
          Map[
           {EccentricityCoordinate[#[[1]], #[[2]]], #[[3]]} &,
           If[hemi === None,
            Join[
```

```
        Select[Flatten[data10[Eccentricity], 2], InV1Q],
        Select[Flatten[data20[Eccentricity], 2], InV1Q]],
       Join[
        Select[Flatten[data10[hemi, Eccentricity], 1], InV1Q],
        Select[Flatten[data20[hemi, Eccentricity], 1], InV1Q]]]],
     ColorFunction → (GrayLevel[1 - #] &),
     ImageSize → 3.25 * 72,
     PlotRange → {{0, 0.75}, {0, 20}},
     LabelStyle → Directive[10, FontFamily → "Arial"],
     PlotLabel → Style[
       "10° and 20° Joined Datasets", FontFamily → "Arial", FontSize → 12],
     Axes → None,
     AspectRatio → 0.6,
     Frame → {{True, False}, {True, False}}],
    SmoothDensityHistogram[
     Map[
      {EccentricityCoordinate[#[[1]], #[[2]]], #[[3]]} &,
      If[hemi === None,
       Select[Flatten[data10[Eccentricity], 2], InV1Q],
       Select[Flatten[data10[hemi, Eccentricity], 1], InV1Q]]],
     ColorFunction → (GrayLevel[1 - #] &),
     ImageSize → 3.25 * 72,
     PlotLabel → Style["10° Dataset", FontFamily → "Arial", FontSize → 12],
     PlotRange → {{0, 0.75}, {0, 20}},
     LabelStyle → Directive[10, FontFamily → "Arial"],
     Axes → None,
     AspectRatio → 0.6,
     Frame → {{True, False}, {True, False}}],
    SmoothDensityHistogram[
     Map[
      {EccentricityCoordinate[#[[1]], #[[2]]], #[[3]]} &,
      If[hemi === None,
       Select[Flatten[data20[Eccentricity], 2], InV1Q],
       Select[Flatten[data20[hemi, Eccentricity], 1], InV1Q]]],
     ColorFunction → (GrayLevel[1 - #] &),
     ImageSize → 3.25 * 72,
     PlotLabel → Style["20° Dataset", FontFamily → "Arial", FontSize → 12],
     PlotRange → {{0, 0.75}, {0, 20}},
     LabelStyle → Directive[10, FontFamily → "Arial"],
     Axes → None,
     AspectRatio → 0.6,
     Frame → {{True, False}, {True, False}}]},
    {True, False, True}]];
FigureExport[name <> ".pdf", GraphicsRow[img], "PDF", ImageResolution → 600];
(* we also want high resolution png's for this *)
FigureExport[name <> ".png", GraphicsRow[img], "PNG", ImageResolution → 600];
(* and individual versions... *)
FigureExport[name <> "-both.pdf", img[[1]], "PDF", ImageResolution → 600];
FigureExport[name <> "-10.pdf", img[[2]], "PDF", ImageResolution → 600];
FigureExport[name <> "-20.pdf", img[[3]], "PDF", ImageResolution → 600];
If[$ShowFigures, GraphicsRow[img]]];
```

Actually make the plot for eccentricity...

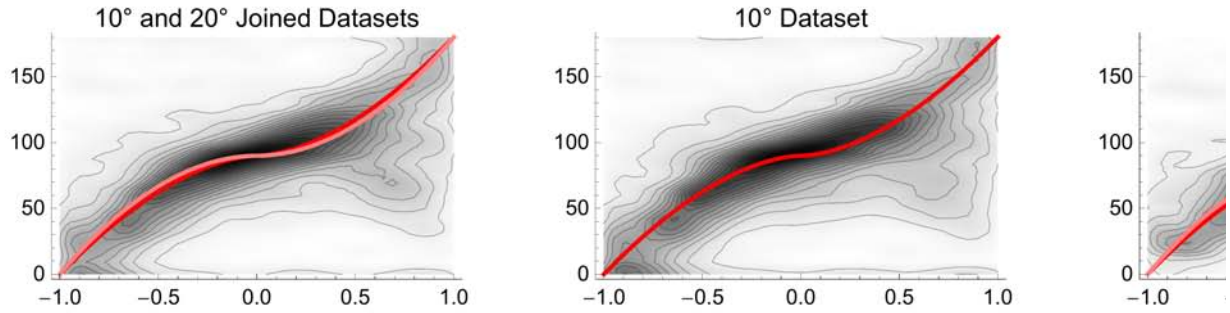
```
Module[
 {img = plotEccHist[None, "eccen-histogram-both"]},
 If[$ShowFigures, img]]
```



And we make the same plot for polar angle.

```
plotPAHist = Function[{hemi, name},
   Module[
    {img = MapThread[
       Show[
         {#1,
          Plot[
           Evaluate[
            If[hemi === None,
             If[#2,
              {data10[Template, PolarAngle][0, V1EllipseB * y],
               data20[Template, PolarAngle][0, V1EllipseB * y]},
              data10[Template, PolarAngle][0, V1EllipseB * y]],
             If[#2,
              {data10[Template, hemi, PolarAngle][0, V1EllipseB * y],
               data20[Template, hemi, PolarAngle][0, V1EllipseB * y]},
              data10[Template, hemi, PolarAngle][0, V1EllipseB * y]]]],
           {y, -1, 1},
           PlotRange → {{-1, 1}, {0, 180}},
           PlotStyle → If[#2, {{Thick, Red}, {Thick, Pink}}, {Thick, Red}],
           ImageSize → 3.25 * 72,
           LabelStyle → Directive[10, FontFamily → "Arial"],
           Axes → None,
           AspectRatio → 0.6,
           Frame → {{True, False}, {True, False}}]}] &,
       {{SmoothDensityHistogram[
          Map[
           {PolarAngleCoordinate[#[[1]], #[[2]]], #[[3]]} &,
           If[hemi === None,
            Join[
             Select[Flatten[data10[PolarAngle], 2], InV1Q],
             Select[Flatten[data20[PolarAngle], 2], InV1Q]],
            Join[
             Select[Flatten[data10[hemi, PolarAngle], 1], InV1Q],
```

```
            Select[Flatten[data20[hemi, PolarAngle], 1], InV1Q]]]],
          ColorFunction → (GrayLevel[1 - #] &),
          ImageSize → 3.25 * 72,
          PlotLabel → Style[
            "10° and 20° Joined Datasets", FontFamily → "Arial", FontSize → 12],
          PlotRange → {{-1, 1}, {0, 180}},
          LabelStyle → Directive[10, FontFamily → "Arial"],
          Axes → None,
          AspectRatio → 0.6,
          Frame → {{True, False}, {True, False}}],
         SmoothDensityHistogram[
          Map[
           {PolarAngleCoordinate[#[[1]], #[[2]]], #[[3]]} &,
           If[hemi === None,
            Select[Flatten[data10[PolarAngle], 2], InV1Q],
            Select[Flatten[data10[hemi, PolarAngle], 1], InV1Q]]],
          ColorFunction → (GrayLevel[1 - #] &),
          ImageSize → 3.25 * 72,
          PlotLabel → Style["10° Dataset", FontFamily → "Arial", FontSize → 12],
          PlotRange → {{-1, 1}, {0, 180}},
          LabelStyle → Directive[10, FontFamily → "Arial"],
          Axes → None,
          AspectRatio → 0.6,
          Frame → {{True, False}, {True, False}}],
         SmoothDensityHistogram[
          Map[
           {PolarAngleCoordinate[#[[1]], #[[2]]], #[[3]]} &,
           If[hemi === None,
            Select[Flatten[data20[PolarAngle], 2], InV1Q],
            Select[Flatten[data20[hemi, PolarAngle], 1], InV1Q]]],
          ColorFunction → (GrayLevel[1 - #] &),
          ImageSize → 3.25 * 72,
          PlotLabel → Style["20° Dataset", FontFamily → "Arial", FontSize → 12],
          PlotRange → {{-1, 1}, {0, 180}},
          LabelStyle → Directive[10, FontFamily → "Arial"],
          Axes → None,
          AspectRatio → 0.6,
          Frame → {{True, False}, {True, False}}]},
         {True, False, True}]];
    FigureExport[name <> ".pdf", GraphicsRow[img], "PDF", ImageResolution → 600];
    (* we also want high resolution png's for this *)
    FigureExport[name <> ".png", GraphicsRow[img], "PNG", ImageResolution → 600];
    (* and individual versions... *)
    FigureExport[name <> "-both.pdf", img[[1]], "PDF", ImageResolution → 600];
    FigureExport[name <> "-10.pdf", img[[2]], "PDF", ImageResolution → 600];
    FigureExport[name <> "-20.pdf", img[[3]], "PDF", ImageResolution → 600];
    If[$ShowFigures, GraphicsRow[img]]]];

Module[
 {img = plotPAHist[None, "polarangle-histogram-both"]},
 If[$ShowFigures, img]]
```

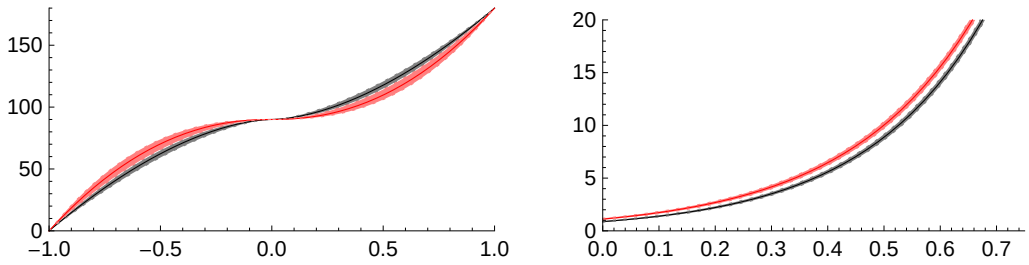


■ Comparison of Fits

Here, we create a plot of the polar angle and eccentricity fits +/- standard error across all subjects

```
Module[
 {fits = Map[
    #["BestFitParameters"][[1, 2]] &,
    Join[data10[Fit], data20[Fit]],
    {3}],
  mu, SE, img},
 mu = Table[
   Mean[
    Select[
     fits[[All, i, j]],
     # < 10 &]],
   {i, 1, 2}, {j, 1, 2}];
 SE = Table[
   (StandardDeviation[#] / Sqrt[Length@#]) &@Select[
     fits[[All, i, j]],
     # < 10 &],
   {i, 1, 2}, {j, 1, 2}];
 img = Block[{q, x, y},
   GraphicsRow[
    {Plot[
      {$PolarAngleTemplateForm /. {q → mu[[1, 1]], x → 0, y → (V1EllipseB * (xx))},
       $PolarAngleTemplateForm /.
        {q → mu[[1, 1]] + SE[[1, 1]], x → 0, y → (V1EllipseB * (xx))},
       $PolarAngleTemplateForm /. {q → mu[[1, 1]] - SE[[1, 1]],
         x → 0, y → (V1EllipseB * (xx))},
       $PolarAngleTemplateForm /. {q → mu[[2, 1]], x → 0, y → (V1EllipseB * (xx))},
       $PolarAngleTemplateForm /.
        {q → mu[[2, 1]] + SE[[2, 1]], x → 0, y → (V1EllipseB * (xx))},
       $PolarAngleTemplateForm /. {q → mu[[2, 1]] - SE[[2, 1]],
         x → 0, y → (V1EllipseB * (xx))}},
      {xx, -1, 1},
      PlotRange → {{-1, 1}, {0, 180}},
      PlotStyle → {{Black}, {Gray, Dotted}, {Gray, Dotted},
        {Red, Dashing[{}]}, {Pink, Dotted}, {Pink, Dotted}},
```

```
      Filling → {2 → {{1}, Gray}, 3 → {{1}, Gray}, 5 → {{4}, Pink}, 6 → {{4}, Pink}},
      LabelStyle → Directive[10, FontFamily → "Arial"],
      ImageSize → 3.25 * 72,
      Frame → Table[{True, False}, {2}],
      Axes → False,
      AspectRatio → 0.5],
     Plot[
      {$EccentricityTemplateForm /.
        {q → mu[[1, 2]], y → 0, x → (V1EllipseA * (2 * xx - 1))},
       $EccentricityTemplateForm /. {q → mu[[1, 2]] + SE[[1, 2]],
        y → 0, x → (V1EllipseA * (2 * xx - 1))},
       $EccentricityTemplateForm /. {q → mu[[1, 2]] - SE[[1, 2]],
        y → 0, x → (V1EllipseA * (2 * xx - 1))},
       $EccentricityTemplateForm /. {q → mu[[2, 2]], y → 0,
        x → (V1EllipseA * (2 * xx - 1))},
       $EccentricityTemplateForm /. {q → mu[[2, 2]] + SE[[2, 2]],
        y → 0, x → (V1EllipseA * (2 * xx - 1))},
       $EccentricityTemplateForm /. {q → mu[[2, 2]] - SE[[2, 2]],
        y → 0, x → (V1EllipseA * (2 * xx - 1))}},
      {xx, 0, 1},
      PlotRange → {{0, 0.75}, {0, 20}},
      PlotStyle → {{Black}, {Gray, Dotted}, {Gray, Dotted},
        {Red, Dashing[{}]}, {Pink, Dotted}, {Pink, Dotted}},
      Filling → {2 → {{1}, Gray}, 3 → {{1}, Gray}, 5 → {{4}, Pink}, 6 → {{4}, Pink}},
      LabelStyle → Directive[10, FontFamily → "Arial"],
      ImageSize → 3.25 * 72,
      Frame → Table[{True, False}, {2}],
      Axes → False,
      AspectRatio → 0.5]}]];
FigureExport[
 "hemisphere-comparison.pdf",
 img,
 "PDF",
 ImageResolution → 600];
If[$ShowFigures, img]
```



■ Split-Halves Reliability Plots

Here, we create a function that will easily/quickly plot the reliability data as a density histogram with a distribution on top.

```
plotRepeatHist = Function[{data, type, hem},
   Module[
```

```
{tmp = Flatten[
   Module[{A, B},
    Map[
     Map[
       {#[[1, 1]], #[[1, 2]], {#[[1, 3]], #[[2, 3]]}} &,
       Select[
        Split[
         Sort[
          Join[
           Map[
            Apply[A, #] &,
            Select[
             #[[1]],
             InV1Q[#] && EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &]],
           Map[
            Apply[B, #] &,
            Select[
             #[[2]],
             InV1Q[#] && EccentricityCoordinate[#[[1]], #[[2]]] < 0.75 &]]],
          Which[
           #1[[1]] < #2[[1]], True,
           #1[[1]] > #2[[1]], False,
           #1[[2]] < #2[[2]], True,
           #1[[2]] > #2[[2]], False,
           Head[#1] === A, True,
           True, False] &],
         (#1[[1]] == #2[[1]] && #1[[2]] == #2[[2]]) &],
        (Length[#] == 2 && Head[#[[1]]] === A && Head[#[[2]]] === B) &]] &,
     Transpose[
      If[hem === None,
       {Join[data[[1]][LeftHemisphere, type],
         data[[1]][RightHemisphere, type]], Join[data[[2]][
          LeftHemisphere, type], data[[2]][RightHemisphere, type]]},
       {data[[1]][hem, type],
        data[[2]][hem, type]}]]]],
   1],
  pdat, pts,
  binsz = If[type === PolarAngle, 0.01, 0.005],
  m, M, μ, qtl},
 pdat = Map[
   {If[type === PolarAngle,
      PolarAngleCoordinate[#[[1]], #[[2]]],
      EccentricityCoordinate[#[[1]], #[[2]]]],
     Abs[#[[3, 1]] - #[[3, 2]]]} &,
   tmp];
 pts = Flatten[
   Reap[
     Scan[
      If[Length@# > 4,
        (Sow[{Mean[#[[All, 1]]], Quartiles[#[[All, 2]]][[1]]}, 1];
         Sow[{Mean[#[[All, 1]]], Quartiles[#[[All, 2]]][[2]]}, 2];
```

```
        Sow[{Mean[#[[All, 1]]], Quartiles[#[[All, 2]]][[3]]}, 3])] &,
      Split[
       Sort[
        pdat,
        #1[[1]] < #2[[1]] &],
       Floor[#1[[1]] / binsz] == Floor[#2[[1]] / binsz] &]],
     {1, 2, 3}
    ][[2]],
   1];
Print[
 {type, hem} → {
   Mean → Mean[tmp[[All, 3, 1]] - tmp[[All, 3, 2]]],
   Median → Median[tmp[[All, 3, 1]] - tmp[[All, 3, 2]]],
   Mean[Abs] → Mean[Abs[tmp[[All, 3, 1]] - tmp[[All, 3, 2]]]],
   Median[Abs] → Median[Abs[tmp[[All, 3, 1]] - tmp[[All, 3, 2]]]]}];
qtl = Quartiles[pdat[[All, 2]]];
Block[{x, terms},
 terms = {x, x^2, x^3, x^4, x^5, x^6};
 m = LinearModelFit[
   pts[[1]],
   terms,
   x];
 μ = LinearModelFit[
   pts[[2]],
   terms,
   x];
 M = LinearModelFit[
   pts[[3]],
   terms,
   x]];
{SmoothDensityHistogram[
  tmp[[All, 3]],
  ColorFunction → (Blend[{White, Black}, #] &),
  PlotRange →
   If[type === Eccentricity, {{0, 10}, {0, 10}}, {{0, 180}, {0, 180}}],
  ImageSize → 2 * 72,
  AspectRatio → 1,
  Frame → Table[{True, False}, {2}],
  Axes → None,
  FrameLabel →
   {{"2nd half data [degrees]", None}, {"1st half data [degrees]", None}},
  LabelStyle → Directive[7, FontFamily → "Arial"],
  PlotLabel → Style[
    Which[
     hem === None, "Both",
     hem === LeftHemisphere, "LH",
     True, "RH"],
    FontSize → 7,
    FontFamily → "Arial"]],
 Block[{x},
  Show[
```

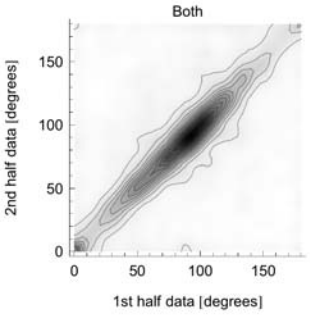
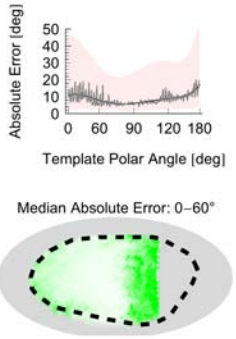
```
  {Plot[
    {m[x], μ[x], M[x]},
    Evaluate[
     If[type === Eccentricity,
      {x, 0, EccentricityCoordinate[FindRoot[10 ==
           data10[Template, Eccentricity][x, 0], {x, -0.1}][[1, 2]], 0]},
      {x, -1, 1}]],
    PlotStyle → {LightPink, Black, LightPink},
    PlotRange → {Automatic, If[type === PolarAngle, {0, 50}, {0, 3}]},
    Filling → {1 → {{3}, LightPink}},
    ImageSize → 1.75 * 72,
    Frame → Table[{True, False}, {2}],
    FrameTicks → {{Automatic, None},
      {Quiet[
        If[type === PolarAngle,
         Table[
          {PolarAngleCoordinate[0, FindRoot[y ==
               data10[Template, PolarAngle][0, x], {x, -0.1}][[1, 2]]], y},
          {y, {0, 60, 90, 120, 180}}],
         Table[
          {EccentricityCoordinate[FindRoot[y == data10[Template,
                 Eccentricity][x, 0], {x, -0.1}][[1, 2]], 0], y},
          {y, {2, 4, 6, 8}}]]],
       None}},
    Axes → None,
    FrameLabel → {{"Absolute Error [deg]", None},
      {"Template " <> If[type === PolarAngle,
         "Polar Angle", "Eccentricity"] <> " [deg]", None}},
    LabelStyle → Directive[7, FontFamily → "Arial"]],
   ListPlot[
    pts[[2]],
    Joined → True,
    PlotStyle → Gray]}]],
Show[
 {BrainPlot[
   Map[
    If[Length@# == 1, #[[1]], {#[[1, 1]], #[[1, 2]], Median[#[[All, 3]]]}] &,
    Split[
     Sort[
      Map[
       {#[[1]], #[[2]], Abs[#[[3, 1]] - #[[3, 2]]]} &,
       tmp],
      Which[
        #1[[1]] < #2[[1]], True,
        #1[[1]] > #2[[1]], False,
        #1[[2]] < #2[[2]], True,
        #1[[2]] > #2[[2]], False,
        True, True] &],
     (#1[[1]] == #2[[1]] && #1[[2]] == #2[[2]]) &]],
   PlotLabel → Style[
     "Median Absolute Error: 0-" <> If[type === PolarAngle, "60°", "8°"],
```

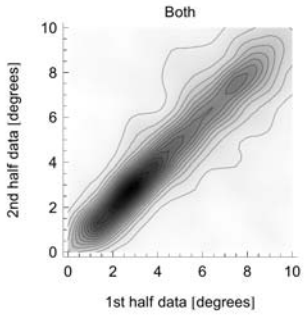
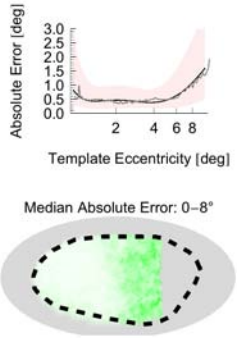
```
      FontFamily → "Arial",
      FontSize → 7],
    ColorFunction → (Blend[{White, Green}, #] &),
    PlotRange → If[type === PolarAngle, {0, 60}, {0, 8}],
    ImageSize → 1.75 * 72],
  ListPlot[V1Hull, Joined → True, PlotStyle → {Thick, Black, Dashed}]}]}]];
```

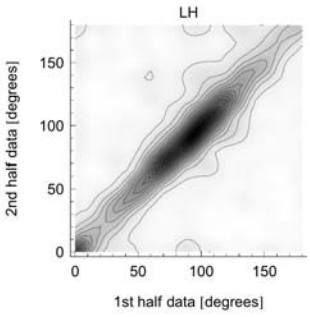
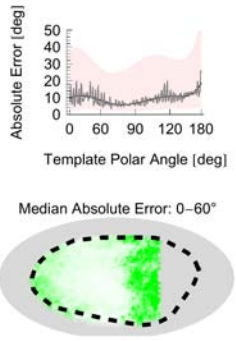
And actually produce the image of these together; this also prints out statistics (here and generally throughout, None, when used as a hemisphere, means both hemisphere)

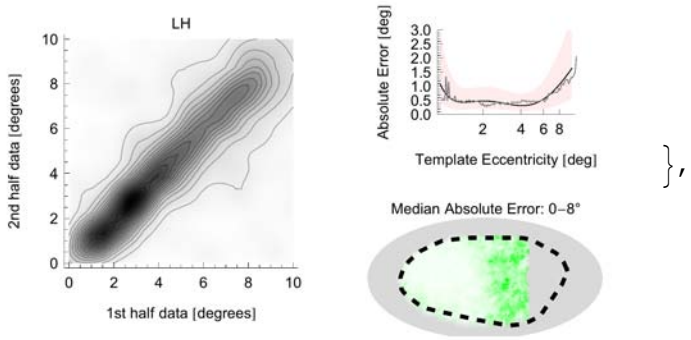
```
Module[
 {img = Table[
    {type, hem} → plotRepeatHist[{data10half1, data10half2}, type, hem],
    {hem, {None, LeftHemisphere, RightHemisphere}},
    {type, {PolarAngle, Eccentricity}}]},
 Map[
  Function[{d},
   Module[
    {type = d[[1, 1]], hem = d[[1, 2]], ims = d[[2]],
     im = GraphicsRow[{d[[2, 1]], GraphicsColumn[d[[2, {2, 3}]]]}],
     stype = If[d[[1, 1]] === Eccentricity, "ecc", "pa"],
     shem = Which[d[[1, 2]] === None,
       "BH", d[[1, 2]] === LeftHemisphere, "LH", True, "RH"]},
    FigureExport["split-halves-" <> stype <> "-" <> shem <> ".pdf", im, "PDF"];
    FigureExport[
     "split-halves-" <> stype <> "-" <> shem <> "-hist.pdf", d[[2, 1]], "PDF"];
    FigureExport["split-halves-" <> stype <> "-" <> shem <> "-err.pdf",
     d[[2, 2]], "PDF"];
    FigureExport["split-halves-" <> stype <> "-" <> shem <> "-vertex.pdf",
     d[[2, 3]], "PDF"];
    If[$ShowFigures, im]]],
  img,
  {2}]]
```

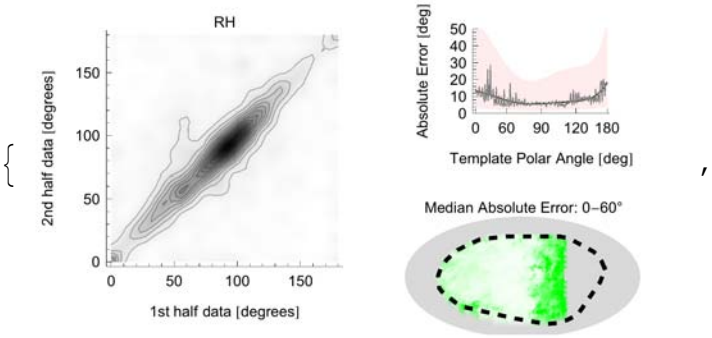
```
{PolarAngle, None} →
 {Mean → -2.19281, Median → 0., Mean[Abs] → 24.2665, Median[Abs] → 7.761}

{Eccentricity, None} →
 {Mean → 0.148539, Median → 0.0137, Mean[Abs] → 1.84895, Median[Abs] → 0.7516}

{PolarAngle, LeftHemisphere} →
 {Mean → -3.16697, Median → 0., Mean[Abs] → 25.7617, Median[Abs] → 8.583}

{Eccentricity, LeftHemisphere} →
 {Mean → 0.169318, Median → 0.0295, Mean[Abs] → 1.81321, Median[Abs] → 0.7207}

{PolarAngle, RightHemisphere} →
 {Mean → -1.1975, Median → 0., Mean[Abs] → 22.7389, Median[Abs] → 7.0255}

{Eccentricity, RightHemisphere} →
 {Mean → 0.127308, Median → 0., Mean[Abs] → 1.88546, Median[Abs] → 0.7871}
```

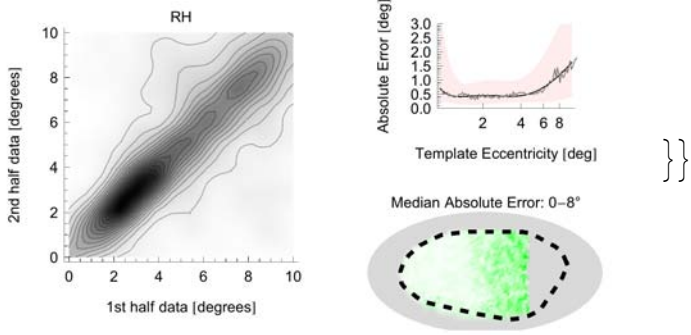
{{ 
Both
2nd half data [degrees]
1st half data [degrees]
0
50
100
150

Absolute Error [deg]
Template Polar Angle [deg]
0
10
20
30
40
50
0
60
90
120
180
Median Absolute Error: 0–60°
,


Both
2nd half data [degrees]
1st half data [degrees]
0
2
4
6
8
10

Absolute Error [deg]
Template Eccentricity [deg]
0.0
0.5
1.0
1.5
2.0
2.5
3.0
2
4
6
8
Median Absolute Error: 0–8°
},

{ 
LH
2nd half data [degrees]
1st half data [degrees]
0
50
100
150

Absolute Error [deg]
Template Polar Angle [deg]
0
10
20
30
40
50
0
60
90
120
180
Median Absolute Error: 0–60°
,







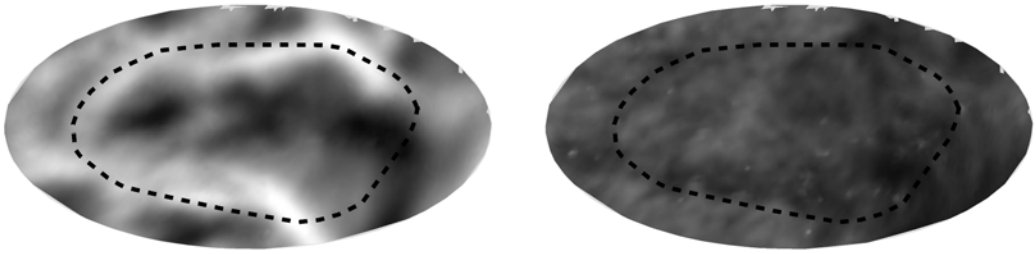
■ Anatomical Data Plots

Here, we look at the anatomical data of each subject by reading in anatomical VTK files, which measure cortical elevation over the spherical brain.
First we prepare them for reading, then we do the calculations

```
data10["Anatomy"] := (data10["Anatomy"] = Map[
      V1Transform /@ {
         ReadVTK[$TemplateDirectory <> "/anatomy/" <> # <> "-lh-curvature.vtk"] /.
          x_Rule :> Join[x[[1]], x[[2]]],
         ReadVTK[$TemplateDirectory <> "/anatomy/" <> # <> "-rh-curvature.vtk"] /.
          x_Rule :> Join[x[[1]], x[[2]]]} &,
      data10[Subjects]]);
data20["Anatomy"] := (data20["Anatomy"] = Map[
      V1Transform /@ {
         ReadVTK[$TemplateDirectory <> "/anatomy/" <> # <> "-lh-curvature.vtk"] /.
          x_Rule :> Join[x[[1]], x[[2]]],
         ReadVTK[$TemplateDirectory <> "/anatomy/" <> # <> "-rh-curvature.vtk"] /.
          x_Rule :> Join[x[[1]], x[[2]]]} &,
      data20[Subjects]]);
$TemplateAnatomy := ($TemplateAnatomy = V1Transform[
      ReadVTK[
        $TemplateDirectory <> "/anatomy/fs-average-template-curvature.vtk"] /.
       x_Rule :> Join[x[[1]], x[[2]]]]);
```

```
Function[{tmp},
  Module[
   {bins, surf, img},
   bins = Map[
     Flatten[#, 1] &,
     BinLists[
      Reap[
        Scan[
         If[InV1DialatedQ[#], Sow[#]] &,
         tmp,
         {3}]
       ][[2, 1]],
      0.001, 0.001, 10 000],
     {2}];
   surf = Reap[
      Scan[
       If[Length@# > 0,
         Sow[{Mean[#[[All, 1]]], Mean[#[[All, 2]]],
           Mean[#[[All, 3]]], StandardDeviation[#[[All, 3]]]}] &,
       bins,
       {2}]
     ][[2, 1]];
   Print[{Min@surf[[All, 3]], Max@surf[[All, 3]]}];
   img = GraphicsRow[
     {Show[
       {BrainPlot[
         Map[{#[[1]], #[[2]], 0.23 - #[[3]]} &, surf[[All, 1 ;; 3]]],
         PlotRange → {0, 0.5},
         ColorFunction → (Blend[{Black, White}, #] &)],
        ListPlot[V1Hull, Joined → True, PlotStyle → {Thick, Black, Dashed}]}],
      Show[
       {BrainPlot[
         surf[[All, {1, 2, 4}]],
         PlotRange → {0, 0.5},
         ColorFunction → (Blend[{Black, White}, #] &)],
        ListPlot[V1Hull, Joined → True, PlotStyle → {Thick, Black, Dashed}]}]}];
   FigureExport[
    "surface.pdf",
    img,
    "PDF",
    ImageResolution → 600];
   If[$ShowFigures, img]]
 ]@Join[data10["Anatomy"], data20["Anatomy"]]
```

```
{-0.298719, 0.208807}
```



We can compare the anatomy of individual subjects from that of the template...

```
Module[
 {anat = Sort[
     Select[$TemplateAnatomy, InV1DialatedQ],
     (#1[[1]] < #2[[1]] || (#1[[1]] == #2[[1]] && #1[[2]] < #2[[2]])) &
    ][[All, 3]]},
 DistributionFitTest[
  Map[
   Sqrt[Mean[(# - anat)^2]] &,
   Map[
    Sort[
       Select[#, InV1DialatedQ],
       (#1[[1]] < #2[[1]] || (#1[[1]] == #2[[1]] && #1[[2]] < #2[[2]])) &
      ][[All, 3]] &,
    Join[
     Flatten[data10["Anatomy"], 1],
     Flatten[data20["Anatomy"], 1]]]],
  Automatic,
  "FittedDistribution"]]
```

```
NormalDistribution[0.135572, 0.013105]
```

Exporting VTK Files

- **Function for exporting VTK files**

We can export a VTK file by reading in another VTK file, mapping each of its points to a value, and writing out the resulting data. This allows us to create a spherical brain VTK file despite the fact that our data has been transformed. In order to facilitate this, we require that the function that maps points to a value accept, as input arguments, 3 lists: the original (x,y,z) coordinates of the points in the VTK file, the converted (θ,ϕ) coordinates of the flattened points, and the values of the points in the vtk file. It must return a list of values to be written out.

```
VTKExport[flnm_String, convert_Function, outnm_String] := Module[
   {v = ReadVTK[flnm],
    vals,
    e = Import[flnm, "PolygonData"],
    f = OpenWrite[outnm]},
   vals = convert[
     v[[All, 1]],
     V1Transform[v[[All, 1]]],
     v[[All, 2, 1]]];
   WriteString[f, "# vtk DataFile Version 1.0\n"];
   WriteString[f, "vtk output\n"];
   WriteString[f, "ASCII\n"];
   WriteString[f, "DATASET POLYDATA\n"];
   WriteString[f, "POINTS ", Length@v, " float\n"];
   Scan[
    WriteString[f,
      NumberForm[#[[1]], {8, 5}, NumberPadding → {"", "0"}], "  ",
      NumberForm[#[[2]], {8, 5}, NumberPadding → {"", "0"}], "  ",
      NumberForm[#[[3]], {8, 5}, NumberPadding → {"", "0"}], "\n"] &,
    v[[All, 1]]];
   WriteString[f, "POLYGONS ",
    Length@e, " ", Length@e + Length[Flatten[e]], "\n"];
   Scan[
    WriteString[f, Length@#,
      Apply[StringJoin, Flatten[Map[{" ", ToString[#]} &, # - 1]]], "\n"] &,
    e];
   WriteString[f, "POINT_DATA ", Length@v, "\n"];
   WriteString[f, "FIELD FieldData 1\n"];
   WriteString[f, "curv 1 ", Length@v, " double\n"];
   Scan[
    WriteString[f, NumberForm[#, {8, 5}, NumberPadding → {"", "0"}], "\n"] &,
    vals];
   Close[f]];
```

- **Export Templates**

For example, we can export the eccentricity and polar angle values

```
VTKExport[
  $TemplateDirectory <> "/metadata/V1-predict.mh.vtk",
  Function[{pts3d, pts2d, vals},
   MapThread[
    If[! InV1Q[#1], 0, data10[Template, PolarAngle][#1[[1]], #1[[2]]]] &,
    {pts2d, vals}]],
  $VTKExportDirectory <> "/angle-template.vtk"];
```

```
VTKExport[
  $TemplateDirectory <> "/metadata/V1-predict.mh.vtk",
  Function[{pts3d, pts2d, vals},
   MapThread[
    If[! InV1Q[#1], 0, data10[Template, Eccentricity][#1[[1]], #1[[2]]]] &,
    {pts2d, vals}]],
  $VTKExportDirectory <> "/eccen-template.vtk"];
```

- **Export Aggregates**

Or we can export aggregates; this is a bit trickier as we have to figure out the aggregate value at a point the hard way

```
Module[
  {paAgg3 = data10[Aggregate, PolarAngle][[All, 3]],
   paAgg12 = data10[Aggregate, PolarAngle][[All, 1 ;; 2]],
   pos},
  VTKExport[
   $TemplateDirectory <> "/metadata/V1-prob.mh.vtk",
   Function[{pts3d, pts2d, vals},
    MapThread[
     Which[
       #2 == 0, 0,
       (pos = Position[paAgg12, pt : {_, _} /; Norm[pt - #1] < 0.01]) == {}, 0,
       True, paAgg3[[pos[[1, 1]]]]] &,
     {pts2d, vals}]],
   $VTKExportDirectory <> "/angle-aggregate.vtk"]];

Module[
  {ecAgg3 = data10[Aggregate, Eccentricity][[All, 3]],
   ecAgg12 = data10[Aggregate, Eccentricity][[All, 1 ;; 2]],
   pos},
  VTKExport[
   $TemplateDirectory <> "/metadata/V1-prob.mh.vtk",
   Function[{pts3d, pts2d, vals},
    MapThread[
     Which[
       #2 == 0, 0,
       (pos = Position[ecAgg12, pt : {_, _} /; Norm[pt - #1] < 0.01]) == {}, 0,
       True, ecAgg3[[pos[[1, 1]]]]] &,
     {pts2d, vals}]],
   $VTKExportDirectory <> "/eccen-aggregate.vtk"]];
```

- **Export Coordinate Systems**

Or the polar angle and eccentricity coordinates

```
VTKExport[
  $TemplateDirectory <> "/metadata/V1-prob.mh.vtk",
  Function[{pts3d, pts2d, vals},
   MapThread[
    If[! InV1Q[#1], 0, PolarAngleCoordinate[#1[[1]], #1[[2]]]] &,
    {pts2d, vals}]],
  $VTKExportDirectory <> "/angle-coordinate.vtk"];
```

```
VTKExport[
  $TemplateDirectory <> "/metadata/V1-predict.mh.vtk",
  Function[{pts3d, pts2d, vals},
   MapThread[
    If[! InV1Q[#1], 0, EccentricityCoordinate[#1[[1]], #1[[2]]]] &,
    {pts2d, vals}]],
  $VTKExportDirectory <> "/eccen-coordinate.vtk"];
```