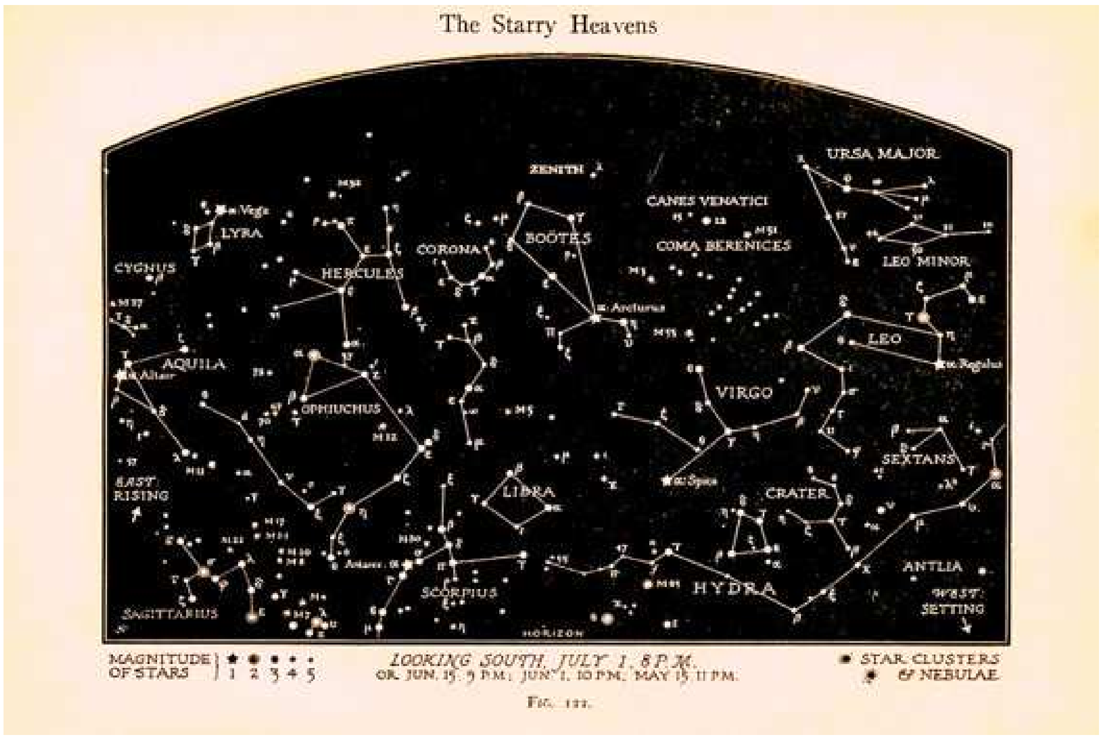
The Doodle Verse



1. Introduction

Each civilization has their own set of constellations, groups of stars ascribed to a particular piece of cultural memory. The ancient Greeks gave us most of our modern constellations: for example, Orion is a mythical hunter, but the Lakota constellation of the Hand is roughly coincident (and consists of many of the same stars). Could we take an arbitrary constellation, and then find a set of stars that fit it? In this lab, we'll explore how to take a set of about 1000 bright stars, and construct spectacular celestial displays of your own design. Welcome to the Doodle Verse!

2. Overview

The problem can be broken up into two main tasks: (1) taking the input image and extracting the ideal feature points, and (2) finding the ideal set of stars whose formation matches this set of feature points.

For the first task, we must perform a series of image processing steps to separate an image into a set of feature points. We must (a) define the image contours as sets of k (x,y) coordinates that accurately depict the original image, and (b) reduce each contour into sets of $n < k$ (x,y) cooridinates that does not substantially degrade the original image representation. We determine this set of n points by testing points p_{x_i,y_i} for how much their removal would induce error between the new contour c_i and the original k point contour c^* by calculating the normalized absolute error E between c_i and c^*: $E = \frac{1}{||p_{x_{i-1},y_{i-1}} - p_{x_{i+1},y_{i+1}}||_2} \int\limits_{p_{x_{i-1},y_{i-1}}}^{p_{x_{i+1},y_{i+1}}} |l(x,y)| \; dx \; dy$ where $l(x,y)$ is the curve of c^* relative to the linear spline connecting $p_{x_{i-1},y_{i-1}}$ and $p_{x_{i+1},y_{i+1}}$. The goal thus becomes to choose only those points that, for some theshold T, do not induce error $E > T$.

For the second task, we start by framing it as an optimization problem. If $\mathbb{S}$ is our set of stars and F is the matrix defined by our feature points, then we are looking for the subset $S \subset \mathbb{S}$ that minimizes the following: $||TF - S||_F$. Here T is a map from the Cartesian "feature space" to the space that our stars lie in that preserves the shape of our feature points (i.e. doesn't distort angles and relative distances).

The number of stars in our $\mathbb{S}$ set (which has been restricted to stars visible with the naked eye) is around 3000, so for n feature points, there are around 3000 choose n possibilities for S. The search space is too large for a brute force method. We will proceed by breaking this problem into smaller problems to the point where a random search will yield sufficiently quick and accurate results.

3. Warmup

a. Getting Feature Points

In order to find a set of n feature points of a drawing, we first need to turn the drawing into a set of contours from which we will get our feature points. In order to accomplish this, we need some form of edge detector.

Algorithm used for finding contours involves a border following technique defined in this paper [1]

Next, in order to narrow down our possible feature points, we can use a corner detector. Similar to how edge detectors look for large changes in particular directions, corner dectors look for large changes in more than one direction. In particular, the Harris Corner Detector.

Harris Corner Detection

When we're designing computer programs to work with images, it is necessary to be able to map between two images of the same (or very similar) objects. In order to do this, we need to be able to reliably extract uniquely recognizable points in each image—we call these features. In particular, we'll talk about corners here. Corners can be thought of as the intersection of two edges, so the gradient of the image will be large in both directions. In particular, if the function $I(x, y)$ represents the intensity of an image at position (x, y), then we want to maximize $E(u, v) = \sum_{x,y} w(x, y)[I(x + u, y + v) - I(x, y)]^2$, where $w(x, y)$ is the window over which we're looking at position (x, y) and (u, v) are the half-widths of the window in the x and y directions. After a Taylor expansion and some algebra, we can express E as

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

where

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x \, I_y \\ I_x \, I_y & I_y^2 \end{bmatrix}$$

Now, for each window, a score R is calculated and compared to a threshold value, to determine if a corner is present:

$$R = [det(M) - k(Tr(M)]^2$$

Furthermore, $det(M)$ and $Tr(M)$ can be expressed in terms of the eigenvalues λ_1 and λ_2: $det(M) = \lambda_1 \times \lambda_2$ and $Tr(M) = \lambda_1 + \lambda_2$. In summary, your results for Harris corner detection may still vary, depending on your choice of u, v, and R.

The local maxima from this approach yields the corners found in the image. By thresholding the $m > n$ largest-valued corners, the original set of $k >> n$ points can be largely reduced to a more manageable set to fine tune by other methods, which is necessary because these m feature points are not necessarily the most representative features of the contour. Many shapes may not give features that may seem obvious to us, so it is often useful to include the top-most, right-most, bottom-most, and left-most features of the contour so that we are assured that at least four of the features will likely be useful. In adding these features, we now have a set of $m + 4$ features to narrow down.

Finally, in order to turn our $m + 4$ feature points into the n feature points that best represent the contour, we perform an algorithm which adds and removes points according to user-defined thresholds. The algorithm is defined below. Note that the order does not greatly affect the final result of the feature set and that the most important aspect is the selection of the threshold values t_{delete} and t_{add}.

N-feature selection algorithm:

1. For all points p_i in the set of $m + 4$ features, find normalized error E_i between p_{i-1} and p_{i+1}. If $E_i < t_{delete}$, remove E_i from the set of feature points. Else, move on to the next point. until all redundant points j have been removed and your new set is composed of $y = m - j$ points.

2. For all points p_i in the set of y features, find normalized error E_i between p_i and p_{i+1}. If $E_i > t_{add}$, then add a new feature point along the contour at the bisecting point and test this new point to check if it meets the t_{add} criterion. Else, move on to the next point. When finished, the new set of feature points is composed of $n = y + r$ points, where r is the number of points added by this step.

Finally, because our approach for matching feature points to stars is based upon a version of triangular mapping, we must determine a good set of three points of the n features that will give us a good chance at finding a match. This set of points will have well defined angles and large enough interpoint distances such that the three points do not line up with each other for form a line or near-line. Thus, the problem becomes a search for which three adjacent feature points along the contour best meet these requirements. In other words, which point solves the following: $p^* = max_i[(||p_i - p_{i-1}||_2 + ||p_i - p_{i+1}||_2)sin(\theta_i)]$ where θ_i is the angle spanning p_{i-1} and p_{i+1} going through p_i.

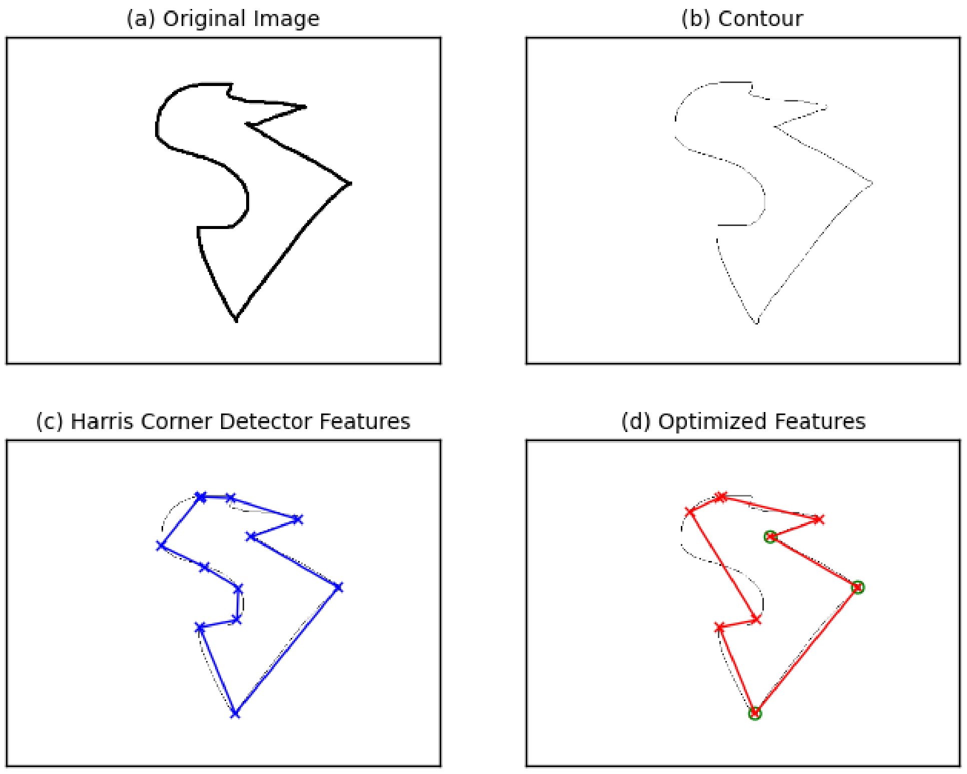


Figure 2

b. Cartesian and Spherical Coordinates: The Mollweide Projection

Next we want to begin our search to see if we can find a set of stars that fits these feature points. The first issue we encounter is that our extracted feature points have Cartesian coordinates in the $(x, y)-$plane and our star data set is in spherical coordinates. Here is a readout of the first few stars of our raw data set:

```
    RA          Dec         Mag
  float64     float64     float64
 ---------- ------------ -------
 0.02662528 -77.06529438    4.78
 0.03039927  -3.02747891    5.13
 0.03266433  -6.01397169    4.37
 0.03886504 -29.72044805    5.04
 0.06232565 -17.33597002    4.55
```

```
0.07503398  −10.50949443    4.99
0.08892938   −5.70783255    4.61
0.13976888   29.09082805    2.07
0.15280269   59.15021814    2.28
0.15583908  −27.9879039     5.42
```

The "RA" or "right ascension" measures the distance from a central meridian and ranges from 0 hours to 24 hours (which corresponds to $0° - 360°$). To convert to degrees we need to multiply this column by 15. The "Dec" or "declination" measures the distances above or below the equator and ranges from $-90°$ to $90°$. The "Mag" or "magnitude" corresponds to the stars brightness. The scale is reversed: *lower numbers correspond to brighter stars.*

To convert from spherical coordinates to Cartesian coordinates, we need to pick a center point and project the stars in the neighborhood of this point onto a plane. If we attempt to project all or most of the stars, this will distort the stars far from our center point, which would be undesirable considering we are trying to match shapes. This restricts our search to small neighborhoods at a time.

The Mollweide projection of a star with spherical coordinates (λ, ϕ) centered around (λ_c, ϕ_c) can be obtained as follows [2]:

$$x = R\frac{2\sqrt{2}}{\pi}(\lambda - \lambda_c)cos(\theta)$$

$$y = R\sqrt{2}sin(\theta)$$

Where θ is the angle defined by: $2\theta + sin(2\theta) = \pi sin(\phi - \phi_c)$. Since θ is implicitly defined, we cannot solve for it directly. But the following iteration will converge to its value after a few iterations (it converges slow for points far from our center point, but that is not a concern for us).

$$\theta_0 = \phi - \phi_c$$

$$\theta_{k+1} = \theta_k + \frac{2\theta_k + sin(2\theta_k) - \pi(sin\phi - \phi_c)}{2 + 2cos(2\theta_k)}$$

Here is a Python script which will compute the Mollweide projection for given spherical coordinates:

```python
def project(ra, dec, c_ra, c_dec):
        '''
        Finds the Mollweide projection coordinates (x,y) for the point (ra,dec) around
        point (c_ra,c_dec).
        '''

        # Find theta
```

```
    theta_0 = dec − c_dec
    epsilon = 10**−6
    error = 1+epsilon

    while error > epsilon:
        m = (2*theta_0+np.sin(2*theta_0)−np.pi*np.sin(dec − c_dec))/(2+2*np.cos(2*theta_0))
        theta_1 = theta_0 − m
        error = np.abs(theta_1 − theta_0)
        theta_0 = theta_1

    # Compute (x,y) coordinates
    x = 2*np.sqrt(2)*(ra−c_ra)*np.cos(theta_0)/np.pi
    y = np.sqrt(2)*np.sin(theta_0)

    return [x,y]
```

Use this script to convert the 79 stars found in *stars.mat* to (x,y) coordinates centered at the $(\lambda_c, \phi_c) = (293, -2.8)$.

c. Preliminary Search: A Clustering Approach

We wish to determine if there is possible good match for our feature points in the set obtained from the previous section. Since we are only trying to match the shape and not the scale of our feature points, one approach is to find a set of stars that forms the same set of interior angles.

In essence, this is a clustering problem. First, let's simplify the problem by considering the case where our feature points consist of just 3 points: (p_1, p_2, p_3). The shape of these 3 points is uniquely determined by their interior angles (i.e. the angles of the triangle they form), call them (a_1, a_2, a_3). Finding these angles is a simple geometric formula:

The angle formed by p_1, p_2, p_3 with a vertex at p_1 is given by:

$$\arccos\left(\frac{P_{12}^T P_{13}}{||P_{12}||||P_{13}||}\right)$$

Where P_{ij} is the vector formed by $(p_i - p_j)$. The following script computes and the angles as we need them:

```
def GetAngles(verts):
    '''
    Gets angles specified by points in verts,
    returns as a sorted array in radians.
    '''

    d = np.zeros((3))
    ab = verts[0]-verts[1]
    ac = verts[0]-verts[2]
    cb = verts[1]-verts[2]
    d[0] = np.linalg.norm(ab)
    d[1] = np.linalg.norm(ac)
    d[2] = np.linalg.norm(cb)

    a = np.zeros((3))
    a[0] = np.arccos(np.dot(ab,ac)/(d[0]*d[1]))
    a[1] = np.arccos(np.dot(ab,cb)/(d[0]*d[2]))
    a[2] = np.arccos(np.dot(ac,cb)/(d[1]*d[2]))

    return np.sort(a)
```

If we think of every set of 3 points in our search space as a specified by their interior angles (sorting the angles in ascending order uniquely determines the set's "angle representation"), we are basically looking for the 3-point sets that lie in the same cluster as our feature points. This cluster would be our "possible match" cluster.

Form this as a clustering problem $X = DW$. What do X, D, W look like? Say our search set contains 50 stars. If D is $n \times k$, what is n? What would be the effect of choosing smaller or larger values of k?

Unfortunately, even with our substantially restricted search space ($30 - 50$ stars), the number of possible size 3 subsets is still quite large (on the order of 10^4), and the clustering algorithms we know require you to iterate through every point. Fortunately, it turns out that solving this cluster problem in its entirety is not necessary. We are not only looking for the *best fit* sets of stars, we are looking for the best fit *and brightest* sets of stars. By sorting the stars by magnitude, we can prioritize clustering the brightest stars so we find the brightest, best fits.

So if we are looking for m possible matches in our subset, we can start with a much smaller subset of brightest stars and keep adding the next-brightest star until the cluster that contains our feature points has m members.

Unfortunately, this approach doesn't scale well when our feature set is more than 3 points (most notably, the search space becomes way too large). However, matching 3 feature points to 3 stars does uniquely determine a transformation from the $(x, y)-$plane that the features lie in to the $(x, y)-$plane that the stars lie in. So rather than finding a match for the remaining points using this clustering approach, we can simply check to see if there are stars in the appropriate places to form the rest of our shape. To do this, we need to find the transformation from the $(x, y)-$coordinates of our features to the $(x, y)-$coordinates of our stars. To find this transformation, we must solve what is called the Procrustes problem.

d. The Orthogonal Procrustes Problem[3]

Here we address the problem of finding the ideal transformation T between our set of feature points F and a predetermined subset of stars $S \subset \mathbb{S}$. Here we are assuming F and S are the same size $(2 \times k)$ and represent their respective collections of points in Cartesian coordinates (each column is a point). Finding this transformation will be necessary to evaluate the fitness of a possible match.

Other than minimizing the distance between points, the primary goal is preserving the shape of F. For now, let's also constrain the problem to where S and F are normalized, so we don't need to change the scale of F: our transformation will preserve distances and angles between points (and thus the shape). This simplifies the problem because it constrains T to be orthogonal. Also, let's assume that both sets of points are centered around the origin (i.e. their mean is $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$). Basically, we are looking for the best rotation of F that minimizes the distance between its points and the points in S. Precisely, we are looking for the orthogonal T that minimizes:

$$||TF - S||_F^2 = \langle TF - S, TF - S \rangle_F$$

Where $tr(AB^T) = \langle A, B \rangle_F = \langle B, A \rangle_F = tr(BA^T)$.

Let $U\Sigma V^T$ be the SVD of the matrix SF^T. Show that the orthogonal T that minimizes $||TF - S||_F^2$ is equal to UV^T.
Hint: T, U, and V are orthogonal, and multiplying a vector by an orthogonal matrix doesn't change its Frobenius norm.

Here is the Python code that solves the orthogonal Procrustes problem:

```python
def Procrustes(F, S):
    '''
    Given F, S (same size, unit norm, and centered around the origin), finds the
    ideal transformation that maps the points of F to best fit the points of S.
    That is, ||TF-S|| is minimized.
    '''

    [U,s,VT] = np.linalg.svd(np.dot(S,F.T))

    T = np.dot(U,VT)

    return T
```

Use the Procrustes algorithm to find the transformation that best maps the A to B matrix found in the *procrustes.mat* file.

e. Evaluating the Match

Now we have sufficient machinery to evaluate our possible match. We know we have a decent match for three of our feature points, and we know how to transform these feature points so they best align with the match. We just have to determine if there is decent match for the remaining feature points. To do this, we apply the same transformation to these feature points, and search for bright stars nearby the transformed points. In the simplest case, we could just pick the closest star to each transformed point and implement some cost function that penalizes dim stars.

Now we have our set of feature points $[f_1, \ldots, f_n] = F$, our possible match set of stars $[s_1, \ldots, s_n] = S$, with $[m_1, \ldots, m_n] = M$ being the set of magnitudes for the stars in S.

Suppose we had k such possible matches $\{S_i, M_i\}$, and we want to determine the *best* one. Set up a cost/penalizer function that we are trying to minimize.

4. Lab

You have been supplied: diamond.png and a star dataset. Perform the following:

1. Import diamond.png to your program of choice (MATLAB or Python). What is a good way to get a set of k feature points from this image? How important is the edge detector your choose for this circumstance? Implement the edge detector of your choice to get a set of k feature points.

2. k is clearly too many feature points, how can you narrow it down to n points that accurately represent the original image? Implement a method of your choosing.

3. If the shape's four sides had small amplitude sinusoids along the flat parts, how would this affect your choice of feature points? Large amplitude sinusoids?

4. Pick 3 of your n points and compute the angles between them. Now find a good potential match in (star-subset) for the triangle formed by these 3 points. Iterate through all possible combinations of stars in descending order of brightness until 3 "good" matches have been found.

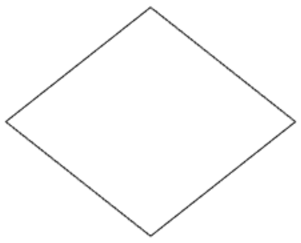


Figure 3: diamond.png

5. Generalize the orthogonal procrustes problem so that it will be useful for this case. That is, given the $(2 \times k)$ matrix of feature points F and a $(2 \times k)$ matrix of star points S (which are not unit norm or centered around the origin), find ideal transformation T such that $||T(F) - S||_F$ is minimized. Note that it is easy to reduce to the previous case by computing the following:

$$\hat{F} = \frac{F}{||F||_F} - mean(F) \quad \text{and} \quad \hat{S} = \frac{S}{||S||_F} - mean(S)$$

   and solving the problem for $\hat{F}$ and $\hat{S}$. Modify the previous Procrustes problem code to solve the general Procrustes problem, and use it to map all of your n feature points to the subspace containing the star projections.

6. For the feature points that weren't part of your triangle, find the closest stars to their transformed positions. Repeat this process for your other good matches and evaluate the overall fit of your matches using your cost function from the previous section. Which match is best? If your cost function had a weighted penalizer for the star magnitude, how does changing this weight affect your evaluation?

References

[1] S. Suzuki and K. Be, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, pp. 32–46, apr 1985.

[2] J. P. Snyder, “Map Projections: A Working Manual,” *U.S. Geological Survey Professional Paper 1395*, pp. 154–163, 1987.

[3] R. Everson, “Orthogonal , but not Orthonormal , Procrustes Problems,” *Convergence*, no. 4, pp. 1–11, 1997.