The Doodle Verse

1. Introduction

The DoodleVerse will make your dreams come true.

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 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

Section about getting feature points.

b. 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 $90°$.

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:

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

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

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:

```
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 points found in (something) to (x,y) coordinates centered at the point (???).

c. Preliminary Search

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.

Even with our substantially restricted search space ($30 - 50$ stars), the possibilities are still too large for a brute force approach. For example, if our feature set had 7 feature points, we would have on the order of 10^7 combinations of stars to try.

As is common when approaching non-convex problems like this, we decided to try a random search approach. Unfortunately, the probability of picking a set of stars whose interior angles match all k feature points is very low for $k > 3$. However, finding a match for just 3 of the feature points can be found rather quickly. This effectively matches a triangle in our feature set to a triangle of stars.

The following script finds interior angles formed by 3 points:

```
Angle script?
```

Now we have a possible match for a subset of our points. We need to 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 a 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

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))
    V = VT.T

    T = np.dot(U,V.T)

    return T
```

e. Evaluating the Match

Brightness stuff

4. Lab

1. Feature detection part
2. Generalize the orthogonal procrustes problem so that it will be useful for our case. That is, given the $(2 \times k)$ matrix of feature points F and a $(2 \times k)$ matrix of star points S (not necessarily 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.
3. Question 2...
4. Question 3....