Project Report CS 217

3D Face/Object Surface Reconstruction Using Calibrated and Uncalibrated Photometric Stereo

Name: Shagoto Rahman
Student ID: 65044968
Email: shagotor@uci.edu

Table of Contents

Overview and Objectives

Photometric stereo is a method in computer vision that estimates an object's surface normals by investigating its appearance under different lighting conditions while keeping the camera position fixed. So broadly we can divide photometric stereo into two major parts [1]. They are:

- **Calibrated**
  - The directions and intensities of the light sources lighting the object are known.
- **Uncalibrated**
  - Phenomenon where the directions and intensities of the light sources are unknown.

For this project, the objectives are divided into two parts. One with Calibrated and one with Uncalibrated. So the objectives are:

Part 1: Calibrated

- Capture (dataset) several images of the face or object illuminated from various known light directions.
- Utilize the calibrated lighting data to precisely calculate the surface normals for every pixel.
- Use the computed normals to reconstruct the 3D shape (mesh) of the object.

Figure 1 depicts the objective of Part 1. So, reconstructing 3D shapes can help in many applications like facial recognition, medical imaging, robotics etc.

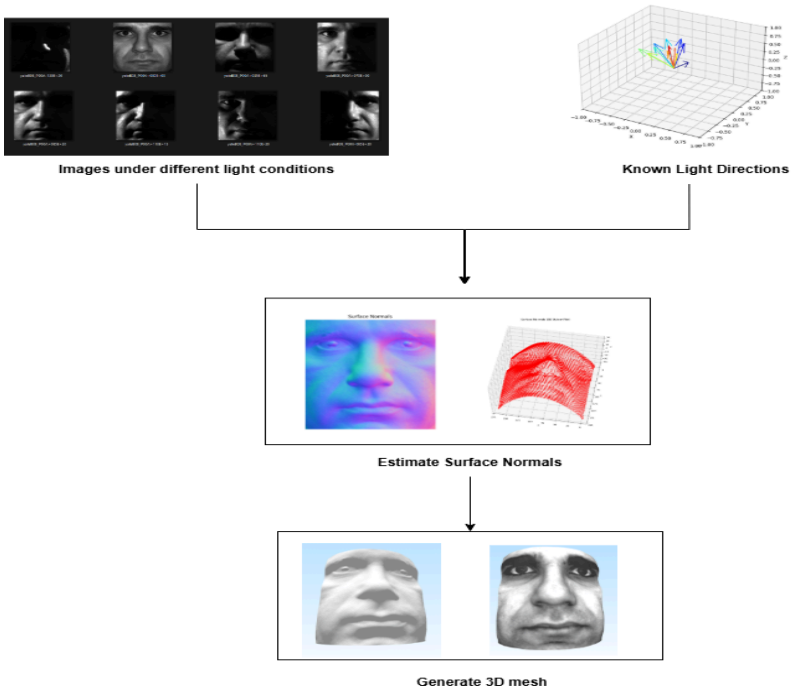

Figure 1. Part 1 objective

Part 2: Uncalibrated

- Capture (dataset) several images of the face or object illuminated from various unknown light directions.
- Estimate the surface normals for every pixel using various methods
- Use the computed normals to reconstruct the 3D shape (mesh) of the object.

Figure 2 depicts the objective of Part 2. Uncalibrated photometric stereo is more challenging and reality based as the light directions are unknown.

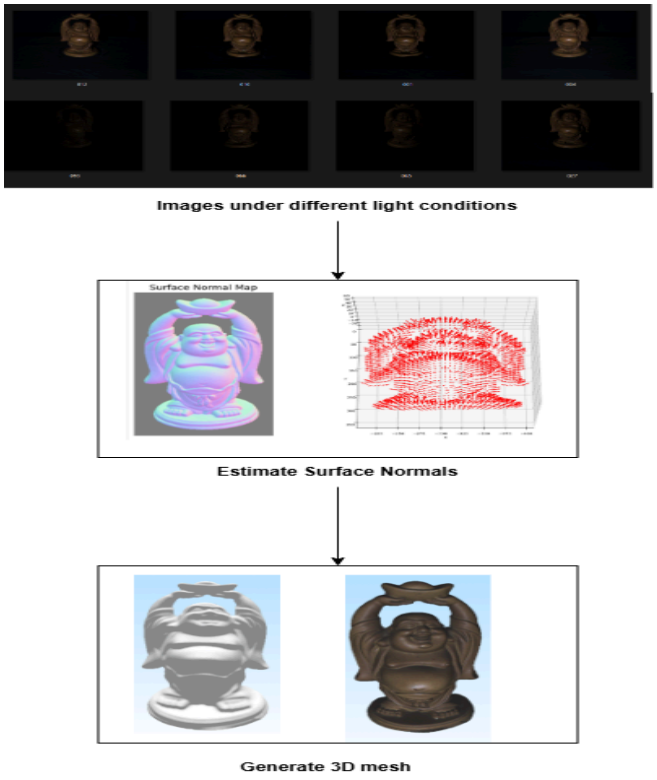

Figure 2. Part 2 objective

Now, there has been lots of work on both calibrated and uncalibrated scenarios. Photometric stereo was first introduced in [2], where surface normals were estimated from light directions, but this method struggled with shadows and non-Lambertian surfaces. In [3], authors developed robust techniques to account for shadows and outliers and improved normal estimation. In [4], authors developed PS-FCN, a Convolutional Neural Network based model to estimate normals from input images and light directions. In [5], authors developed SDPS-Net to predict surface normals in a two stage framework where they predicted light intensities and directions first and then predicted the normals.

Thus, for this project the main objective is to explore calibrated and uncalibrated photometric stereo methods and generate 3D surfaces from images and then compare the performances between the methods.

Datasets

As for the datasets, for this project two datasets have been enabled. They are described in the following:

The Yale Face Dataset [6]

The Yale Face Database B is a facial image dataset comprising 5,850 grayscale images of 10 subjects captured under different conditions with 9 specific head poses and 64 illumination settings. Frontal pose images include detailed facial landmark coordinates. However, other poses provide face center positions. Captured using a strobe-based illumination rig at 30 frames per second, the dataset enables research in face recognition, illumination modeling, and 3D reconstruction under different lighting and pose conditions. For this analysis, a single pose (frontal) has been utilized. Figure 3 depicts some images from the Yale dataset.

Figure 3. Yale dataset.

DiLiGenT [7]

The DiLiGenT (Dataset for Light and Geometry from Images) benchmark is a comprehensive resource for evaluating photometric stereo methods. It includes high-resolution (612×512) images of ten real-world objects, each photographed under 96 known lighting directions. Each object's folder contains data such as a binary mask, calibrated lighting directions, RGB light intensities. Ground-truth surface normals are also provided. DiLiGenT a robust and standardized benchmark for both calibrated and uncalibrated photometric stereo research. Figure 4 depicts some images from the DiLiGenT dataset.

Figure 4. DiLiGenT dataset.

Technical Approach

Part 1: Calibrated

To get the mesh from images taken under different light conditions the procedure is divided into several steps. The steps for calibrated part are described as follows:

Step 1: Reflectance Equation (Lambertian Model)

From Lambertian model, we know that,

$$I = \rho \cdot (\mathbf{N} \cdot \mathbf{L})$$

Where,

I is the observed intensity at a pixel.

ρ is the albedo (the surface's inherent brightness).

$N \in R^3$ is the unit normal vector at the surface point.

$L \in R^3$ is the unit vector representing the light direction.

The dot product $N \cdot L = \cos(\theta)$, where θ is the angle between the normal and the light.

So, intensity is highest when light hits the surface straight on, and weaker at shallow angles.

This theory helps us to extract the normal of an object from the collections of images.

Step 2: Stacking Multiple Images

Since N images are available under N different known lighting directions $L_1, L_2, \ldots, L_N$. *So* each image gives:

$$I_i = \rho \cdot (\mathbf{N} \cdot \mathbf{L}_i), \quad \text{for } i = 1, \ldots, N$$

Stacking these equations we get:

$$\begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_N \end{bmatrix} = \begin{bmatrix} \mathbf{L}_1^T \\ \mathbf{L}_2^T \\ \vdots \\ \mathbf{L}_N^T \end{bmatrix} \cdot (\rho \mathbf{N}) \quad \Rightarrow \quad \mathbf{I} = L \cdot \mathbf{x}$$

Where: $\mathbf{x}=\rho\mathbf{N} \in R^3$ is the unknown, the 3D vector that encodes both albedo and normal.

So the full equation is: $\mathbf{I}=L \cdot \mathbf{x}$

So, to get ρ and N, x has been solved in the following step.

Step 3: Solving for x=ρN

So, the linear system is solved using least squares (since N≥3):

$$\mathbf{x} = (L^T L)^{-1} L^T \mathbf{I}$$

Step 4: Recovering Albedo and Normal

Since x has been solved in the previous step, so we know:

$$\rho = \|\mathbf{x}\|$$
$$\mathbf{N} = \frac{\mathbf{x}}{\|\mathbf{x}\|}$$

So from these equations normal and albedo have been extracted.

Step 5: Reconstruct Depth via Gradient Integration using Frankot-Chellappa Algorithm [8]

From the surface normal vector N = (Nx, Ny, Nz), the gradients p and q are computed, which are partial derivatives of the depth map Z:

$$p = \frac{\partial Z}{\partial x} = -\frac{N_x}{N_z + \epsilon}, \quad q = \frac{\partial Z}{\partial y} = -\frac{N_y}{N_z + \epsilon}$$

To get Z, we have to solve Poisson's equation:

$$\Delta Z = \frac{\partial^2 Z}{\partial x^2} + \frac{\partial^2 Z}{\partial y^2} = \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y}$$

But rather than solving the Poisson's equation in the spatial domain, we can robustly solve that in the frequency domain using the Frankot-Chellappa algorithm.

So, Fourier transforms of the gradients has been performed:

$$\hat{p} = \mathcal{F}[p], \quad \hat{q} = \mathcal{F}[q]$$

Then frequency grids were defined as:

$$f_x = \frac{2\pi k}{W}, \quad f_y = \frac{2\pi l}{H}, \quad k = 0, \ldots, W-1, \quad l = 0, \ldots, H-1$$

Then, the following Fourier transform of the depth Z was solved:

$$\hat{Z}(f_x, f_y) = -\frac{j f_x \hat{p}(f_x, f_y) + j f_y \hat{q}(f_x, f_y)}{f_x^2 + f_y^2}, \quad \text{with} \quad \hat{Z}(0,0) = 0$$

Finally, the depth map was extracted by inverse Fourier transform:

$$Z(x, y) = \mathcal{F}^{-1}[\hat{Z}(f_x, f_y)]$$

Step 6: Construct 3D Mesh from Depth Map [9]

Using the depth Z(x, y), 3D vertices were constructed as:

$$\mathbf{v}(x,y) = \begin{bmatrix} x \\ y \\ Z(x,y) \end{bmatrix}$$

Then the triangles were formed by connecting adjacent vertices in the image grid. For each pixel (x, y), two triangles were defined as:

$$T_1 = \{\mathbf{v}(x,y), \mathbf{v}(x+1,y), \mathbf{v}(x,y+1)\}$$
$$T_2 = \{\mathbf{v}(x+1,y), \mathbf{v}(x+1,y+1), \mathbf{v}(x,y+1)\}$$

Then these vertices and faces defined the mesh as follows:

$$\text{Mesh} = (\mathcal{V}, \mathcal{F}), \quad \mathcal{V} = \{\mathbf{v}_i\}, \quad \mathcal{F} = \{T_k\}$$

Another alternative for step 6 is:

Step 6 (Alternative): Mesh Reconstruction via Point Cloud + Ball-Pivoting Algorithm (BPA) [10]

A ball radius r was selected based on average nearest-neighbor distances. Then:

• Triplets (pi, pj , pk) were found such that there exists a ball center c ∈ R3 satisfying:

$$\|\mathbf{p}_i - \mathbf{c}\| = \|\mathbf{p}_j - \mathbf{c}\| = \|\mathbf{p}_k - \mathbf{c}\| = r,$$

and for all other points

$$\|\mathbf{p}_l - \mathbf{c}\| \geq r.$$

• A triangle $\Delta p_i p_j p_k$ was formed in the mesh for each such triplet.

• The ball was rolled over edges of existing triangles to find adjacent triangles, repeating until the mesh was complete.

Thus, this method ensured the reconstructed mesh closely followed the shape implied by the point cloud and their normals.

So this was the methodology to get a mesh for the calibrated part. Now the method for

uncalibrated part is discussed below:

Part 2: Uncalibrated

For the uncalibrated part, the entire method has three modes. They are:

- Mode 1: Estimate surface normals using SVD.
- Mode 2: Estimate surface normals using simple deep learning model.
- Mode 3: Estimate light configurations and surface normals using more complex deep learning model.

After getting the normal from the modes the normal to depth integration and 3D mesh generation is the same as in Part 1.

Mode 1: Estimate surface normals using SVD-based method. [11]

We only know the observed intensities I. The goal is to factor the intensity matrix I into two rank-3 matrices. So, the Singular Value Decomposition (SVD) to the intensity matrix:

$$\mathbf{I} = \mathbf{U} \cdot \mathbf{S} \cdot \mathbf{V}^T$$

Then, we keep the top-3 components (rank-3 approximation):

$$\mathbf{I} \approx \mathbf{U}_3 \cdot \mathbf{S}_3 \cdot \mathbf{V}_3^T$$

Now, we define:

$$\mathbf{L}' = \mathbf{U}_3 \cdot \mathbf{S}_3^{1/2} \in \mathbb{R}^{m \times 3}$$
$$\mathbf{B}' = \mathbf{S}_3^{1/2} \cdot \mathbf{V}_3^T \in \mathbb{R}^{3 \times n}$$

So the equation becomes:

$$\mathbf{I} \approx \mathbf{L}' \cdot \mathbf{B}'$$

Problem: Ambiguity, The Generalized Bas-Relief (GBR)

Because this factorization is not unique. Any invertible 3 × 3 matrix G produces the same result:

$$\mathbf{L} = \mathbf{L}'\mathbf{G}^{-1}, \quad \mathbf{B} = \mathbf{G}\mathbf{B}' \quad \Rightarrow \quad \mathbf{I} = \mathbf{L}'\mathbf{G}^{-1} \cdot \mathbf{G}\mathbf{B}' = \mathbf{L}' \cdot \mathbf{B}'$$

This means the normals recovered from B′ are only defined up to a Generalized Bas-Relief (GBR) transformation.

So, for this part the normal has been computing using the SVD by normalizing B′, which has the ambiguity and the results will be found in the Results and Analysis Section.

Mode 2: Estimate surface normals using simple deep learning model.

For this analysis, a deep learning model has been created to estimate normals. Given a set of 96 grayscale images captured under different lighting conditions, the images are stacked to form an input tensor:

$$\mathbf{I}(x, y) = [I_1(x, y), I_2(x, y), \ldots, I_{96}(x, y)] \in \mathbb{R}^{96}$$

The full input tensor is of shape is:

$$\mathbf{I} \in \mathbb{R}^{B\times 96\times H\times W}$$

Here, B is the batch size.

Now the Convolutional Network Structure Model has the following architecture:

$$\mathbf{F}_1 = \mathrm{ReLU}(\mathrm{Conv}_{3\times 3}^{96\to 64}(\mathbf{I}))$$
$$\mathbf{F}_2 = \mathrm{ReLU}(\mathrm{Conv}_{3\times 3}^{64\to 128}(\mathbf{F}_1))$$
$$\mathbf{F}_3 = \mathrm{ReLU}(\mathrm{Conv}_{3\times 3}^{128\to 64}(\mathbf{F}_2))$$
$$\hat{\mathbf{N}} = \tanh(\mathrm{Conv}_{3\times 3}^{64\to 3}(\mathbf{F}_3)) \in \mathbb{R}^{B\times 3\times H\times W}$$

The final output $\hat{N} = [\hat{n}_x, \hat{n}_y, \hat{n}_z]$ is the predicted surface normal vector at each pixel, constrained to the range [−1, 1] by the tanh activation.

Loss: The cosine similarity loss between the predicted normal n̂ and the ground truth normal n has been used, with a binary mask M(x, y) ∈ {0, 1} to ignore invalid pixels.

Next, Both predicted and ground truth normals are normalized:

$$\hat{\mathbf{n}} = \frac{\hat{\mathbf{n}}}{\|\hat{\mathbf{n}}\|_2}, \quad \mathbf{n} = \frac{\mathbf{n}}{\|\mathbf{n}\|_2}$$

So the per-pixel Cosine Loss:

$$\ell(x, y) = 1 - \hat{\mathbf{n}}(x, y) \cdot \mathbf{n}(x, y)$$

And the Masked Mean Loss:

$$\mathcal{L} = \frac{1}{\sum_{x,y} M(x, y)} \sum_{x,y} M(x, y) \cdot \ell(x, y)$$

This loss penalizes angular deviation between the predicted and ground truth normals only in valid regions of the mask.

The training has been performed on DiLigenT dataset while keeping the particular object (pot1 or buddha) for testing. Training epochs have been used as 20 and batch size as 1.

Mode 3: Estimate light configurations surface normals using more complex deep learning model. [5]

For this analysis, to enable a complex deep learning model, SDPS (Self-calibrating Deep Photometric Stereo Networks) has been utilized. SDPS-Net is a two-stage deep learning framework for uncalibrated photometric stereo, which estimates surface normals of an object given multiple images taken under unknown lighting. The stages are:

• Stage 1: Lighting Calibration Network (LCNet)

Estimates lighting direction and intensity for each input image.

• Stage 2: Normal Estimation Network (NENet)

Estimates surface normals using the input images and the estimated lighting from LCNet.

Details of the stages are in the following:

Stage 1: Lighting Calibration Network (LCNet) Overview

It predicts the lighting direction and intensity from a set of input images under unknown lighting.

Lighting Representation: Each lighting condition is represented by:

Direction: Azimuth (ϕ) and elevation (θ), Intensity: Scalar value

Instead of direct regression, LCNet formulates lighting estimation as a classification problem for better learning.

Discretization

Let, Kd and Ke be the number of bins for direction and intensity, respectively.

Then, Azimuth $\phi \in [0^\circ, 180^\circ]$ discretized into Kd bins, Elevation $\theta \in [-90^\circ, 90^\circ]$ discretized into Kd bins, Intensity $\in [0.2, 2.0]$ discretized into Ke bins.

The final direction is reconstructed from the centers of the predicted bins.

Stage 1: Lighting Calibration Network (LCNet) Functionality

Feature Extraction and Local-Global Fusion

a) Extracts a local feature for each input image using a CNN

b) Aggregates all local features via max-pooling to produce a global feature

c) Fuses local and global features and passes through a classifier for azimuth, elevation, and intensity

Loss Function: The total lighting loss is the sum of cross-entropy losses for:

$$L_{\text{Light}} = \lambda_{la} L_{la} + \lambda_{le} L_{le} + \lambda_e L_e$$

Where,

- L_{la}: Loss for azimuth
- L_{le}: Loss for elevation
- L_e: Loss for intensity

These outputs are passed to NENet.

Stage 2: Normal Estimation Network (NENet) Overview

Estimates a dense normal map (one 3D vector per pixel) using the input images and the predicted lighting conditions from LCNet.

• Each image is normalized using its estimated intensity.

• The estimated lighting direction is concatenated with the normalized image

Stage 2: Normal Estimation Network (NENet) Functionality

Feature Processing

- A shared CNN extracts a local feature from each image-light pair
- Max-pooling is applied across local features to obtain a global feature

Normal Prediction

The global feature is passed to a regression head that outputs the surface normal for each pixel.

Loss Function

The loss measures the cosine distance between predicted and ground-truth normals:

$$L_{\text{Normal}} = \frac{1}{hw} \sum_{i=1}^{hw} \left(1 - \mathbf{n}_i^\top \tilde{\mathbf{n}}_i\right)$$

Where,

- $\mathbf{n}_i$: Predicted normal at pixel i
- $\tilde{\mathbf{n}}_i$: Ground-truth normal at pixel i
- h, w: Image height and width

Figure 5 depicts the networks for SDPS. For training Blobby and Sculpture datasets have been utilized. LCNet was trained for 20 epochs with batch size 32 and NENet with 10 epochs and batch size 16.

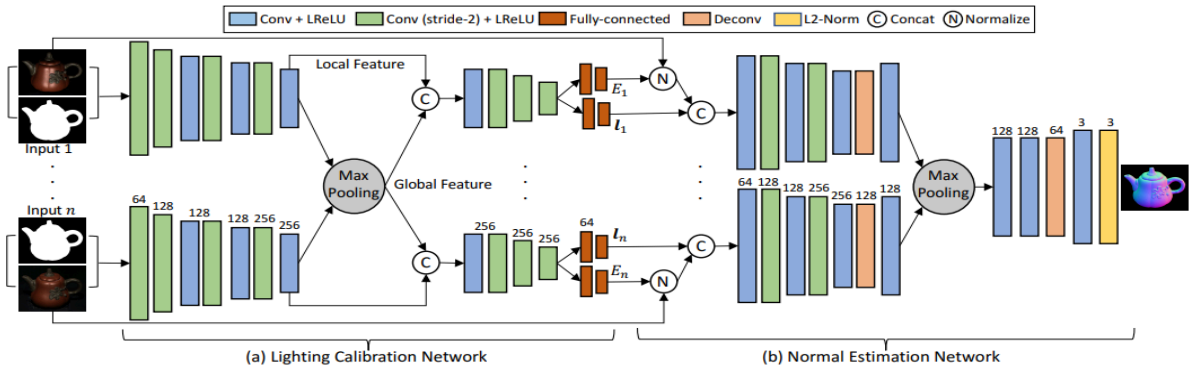

Figure 5. SDPS Network

Better Mesh Visualization

For better mesh visualization all the GIF files of the generated mesh are available at : https://shrestho10.github.io/shagoto-mesh-website/

Results and Analysis

Part 1- Calibrated

Yale Data

Figure 6 depicts the normal, albedo and normal directions (3D) for Yale Data for a subject. From the Figure we can see that the normal in 2D makes sense as the object's left part of the face (right from our point of view) is pinkish as it should be as the direction of surface should be right and outward that supports the more reddish. Similarly, the person's right part of face (left from our viewpoint) is blueish which makes sense as the direction of x is left now. So does the 3D plot make sense as all the arrows are outward and pointing perfectly and so makes sense the albedo.

Figure 7 depicts the generated mesh from the predicted normals of the Yale dataset person from Figure 6. Color mesh is additionally extracted by just adding the colors in the vertices. From the meshes, we can see how good the face has been estimated.

Similarly, Figure 8 depicts the normal, albedo and normal directions (3D) for another Yale Data for a subject. That makes sense exactly as the previous face.

Similarly, Figure 9 depicts the generated mesh from the predicted normals of the Another Yale dataset person from Figure 8. From the meshes, we can see how good the face has been estimated.

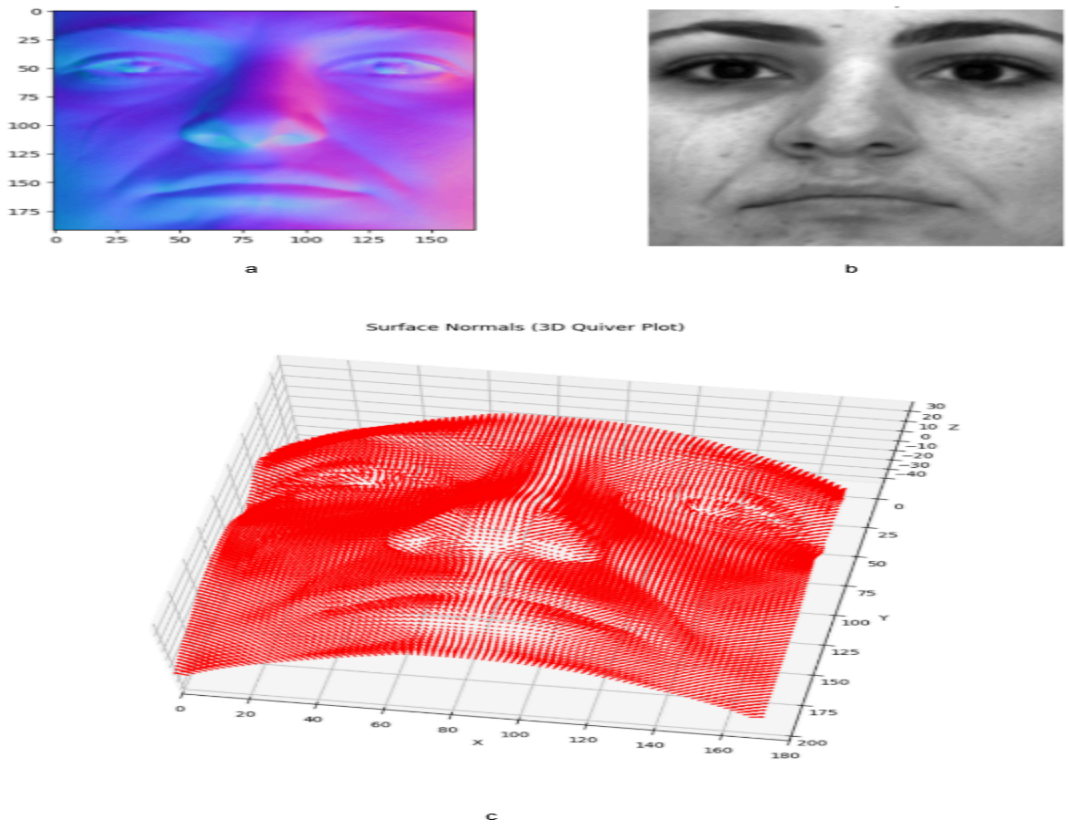

Figure 6. Yale data normal, albedo and normal directions (3D)

Figure 7. Yale data meshes.

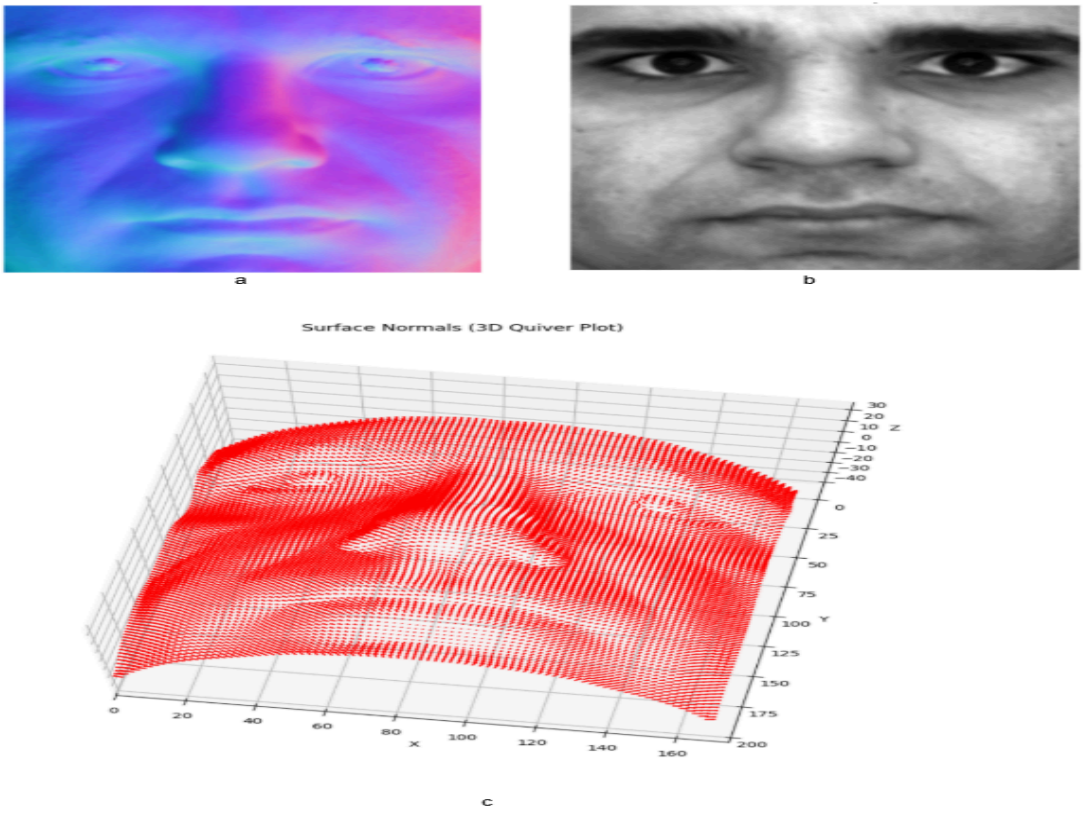

Figure 8. Yale data normal, albedo and normal directions (3D)

Figure 9. Another Yale data meshes.

DiliGenT Data

Figure 10 depicts the normal, albedo and normal directions (3D) for Pot1 of DiliGenT dataset. From the figure we can see the normals make sense as the object's left part is pinkish and the

right part is blueish. Also the directions are perfect, as the albedo.

Figure 11 shows the color albedo that also makes sense. Figure 12 shows the meshes generated from the objects predicted normals and we can see that the meshes are excellently perfect.

Similarly, Figure 13 depicts the normal, albedo and normal directions (3D) for the Buddha of DiliGenT dataset. From the figure we can see the normals make sense as the object's left part is pinkish and the right part is blueish. Also the directions are perfect, as the albedo.

Similarly, Figure 14 shows the color albedo that also makes sense. Figure 15 shows the meshes generated from the objects predicted normals and we can see that the meshes are excellently perfect.

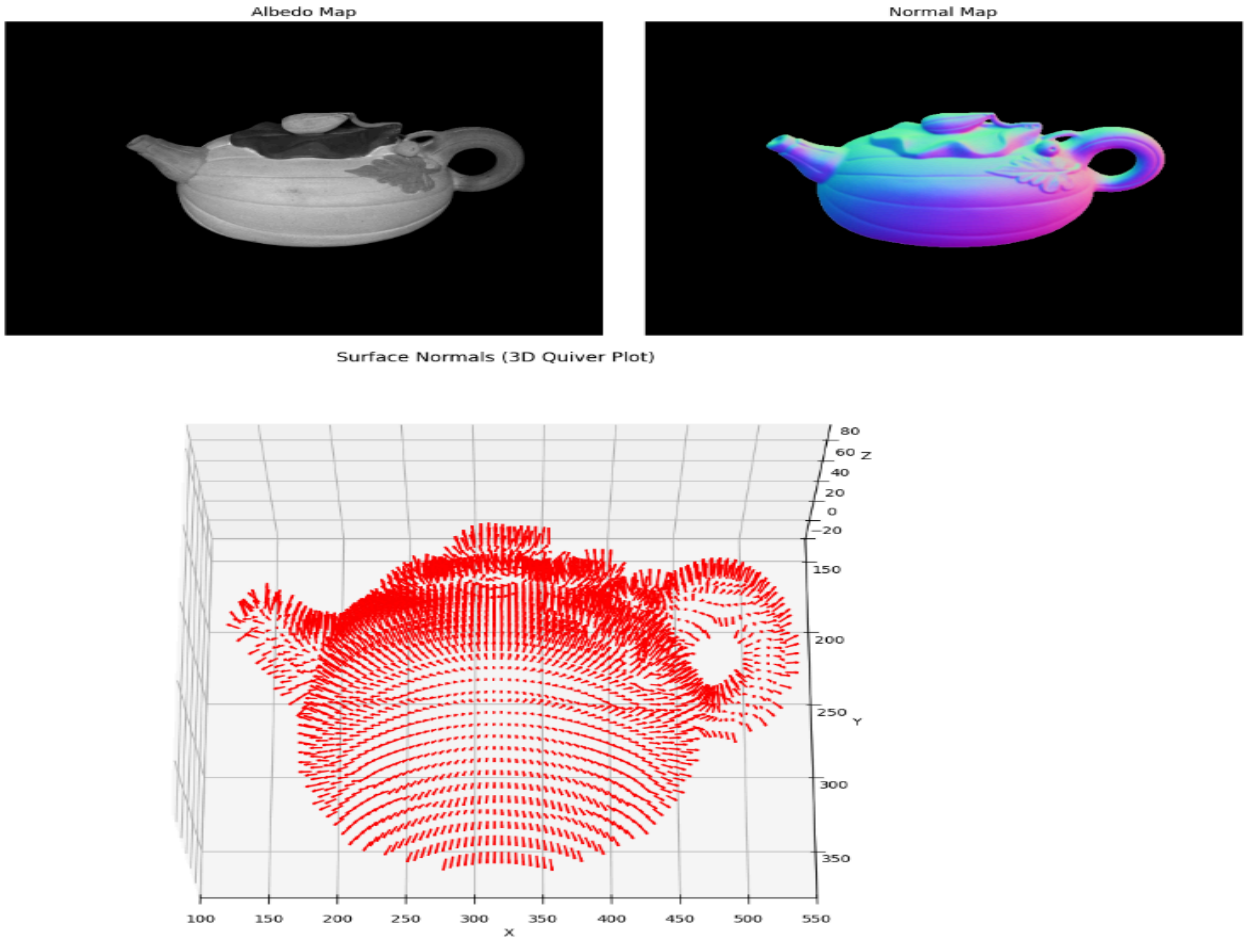

Figure 10. Pot1 normal, albedo and normal directions (3D)

Figure 11. Color Albedo

Figure 12. Meshes for Pot1

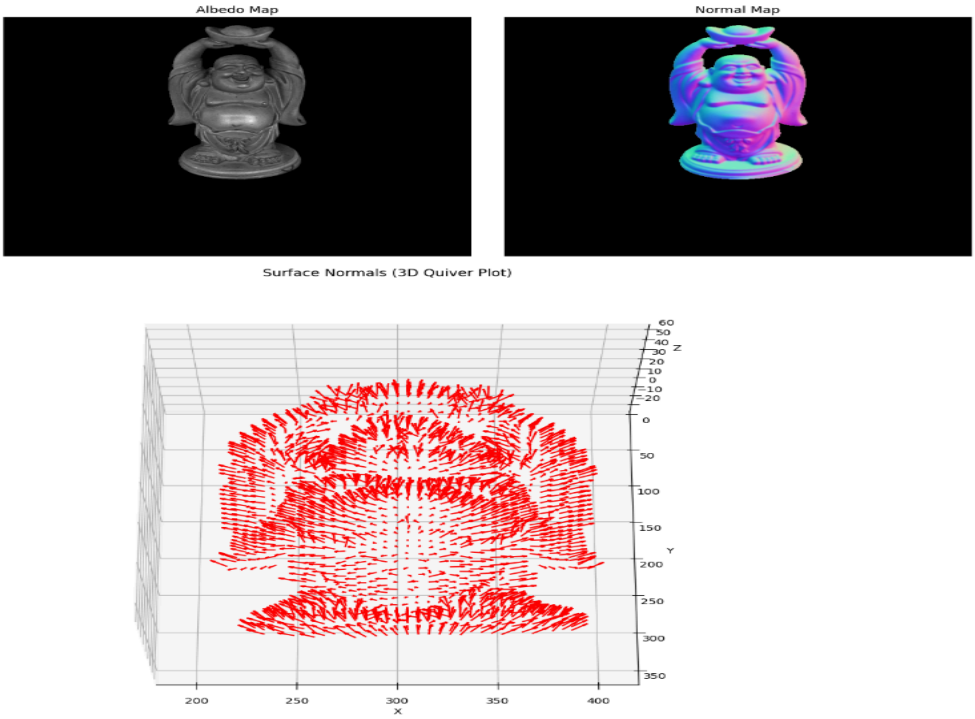

Figure 13. Buddha normal, albedo and normal directions (3D)

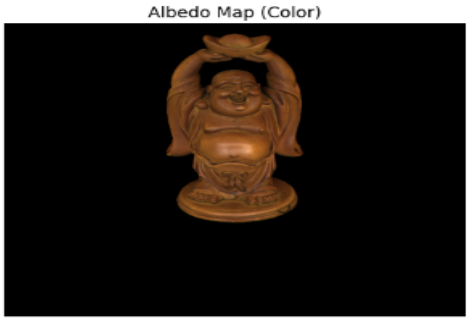

Figure 14. Color Albedo

Figure 15. Meshes for Buddha

Part 2 Uncalibrated

Mode 1 SVD

Ground Truth Normals

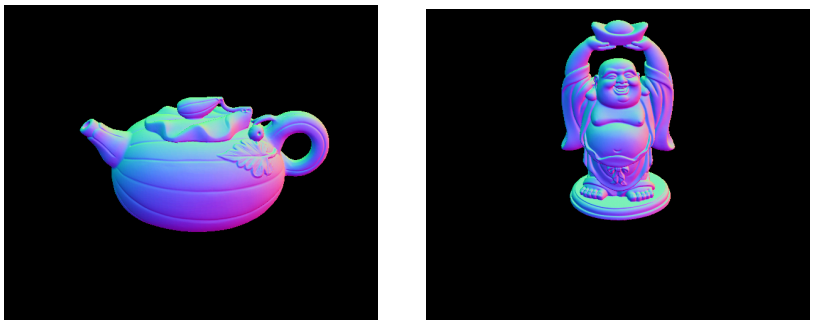

Figure 16. Ground Truth Normals for Pot1 and Buddha

Figure 17 shows the SVD based Pot1 normal and normal directions (3D). We can see the poor approximations because of the ambiguity here.

Figure 18 shows the meshes for Pot1 generated from the SVD, which also shows the poor approximation of the meshes because of the ambiguity.

Similarly, Figure 19 shows the SVD based Buddha normal and normal directions (3D). We can see the poor approximations because of the ambiguity here.

Figure 20 shows the meshes for Buddha generated from the SVD, which also shows the poor approximation of the meshes because of the ambiguity.

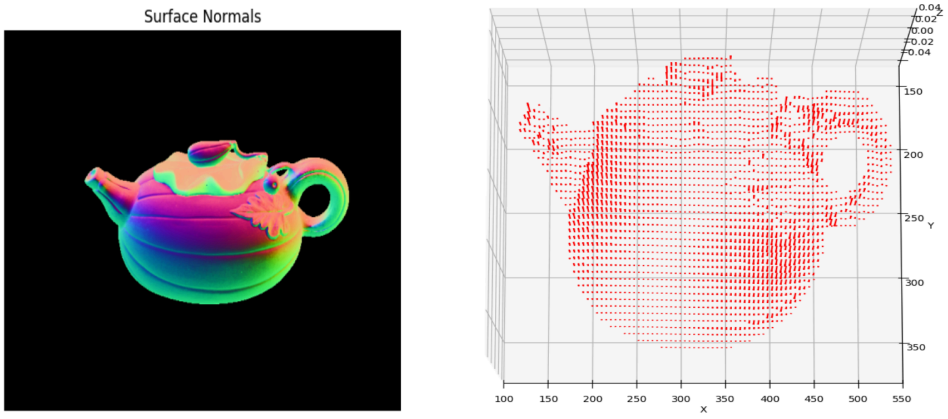

Figure 17. SVD based Pot 1 normal and normal directions (3D)

Figure 18. Meshes from SVD Pot1

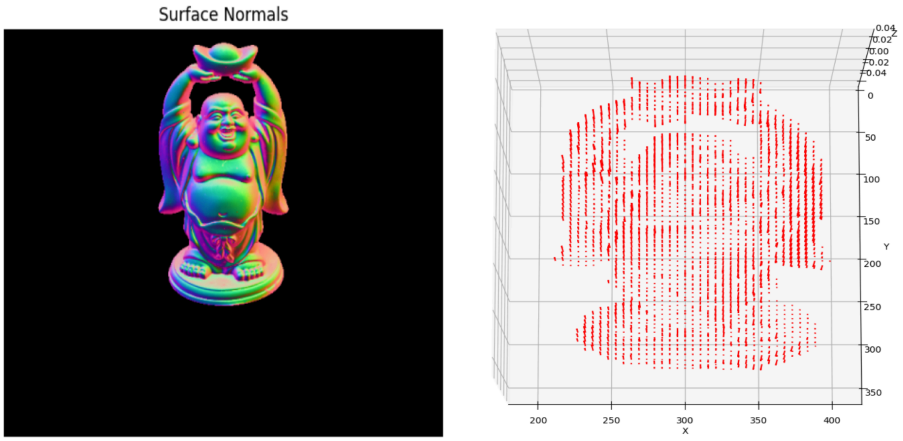

Figure 19. SVD based Buddha normal, and normal directions (3D)

Figure 20. Meshes from Buddha SVD

Mode 2 Simple Deep Learning (SDL)

Figure 21 depicts SDL based Pot1 normal and normal directions (3D). We can see even with a shallow network the normal approximation is very good and so confirms the direction of the 3D plot.

Figure 22 depicts the meshes for Pot1 with SDL, which further confirms that the normal approximation was very good as the meshes are excellent.

Similarly, Figure 23 depicts SDL based Buddha normal and normal directions (3D). We can see even with a shallow network the normal approximation is very good and so confirms the direction of the 3D plot.

Figure 24 depicts the meshes for Buddha with SDL, which further confirms that the normal approximation was very good as the meshes are excellent.

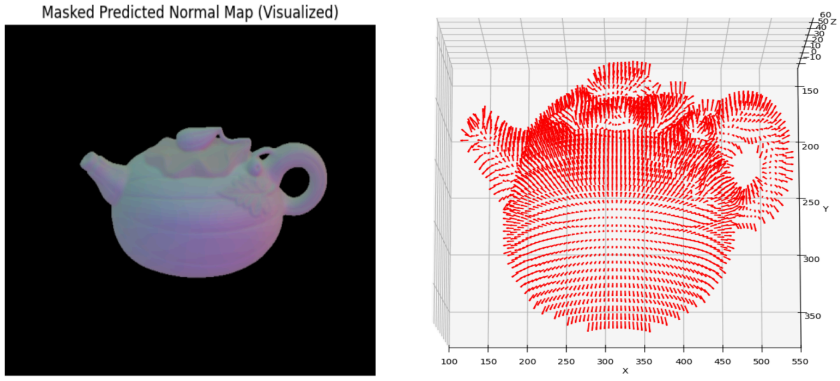

Figure 21. SDL based Pot1 normal and normal directions (3D)

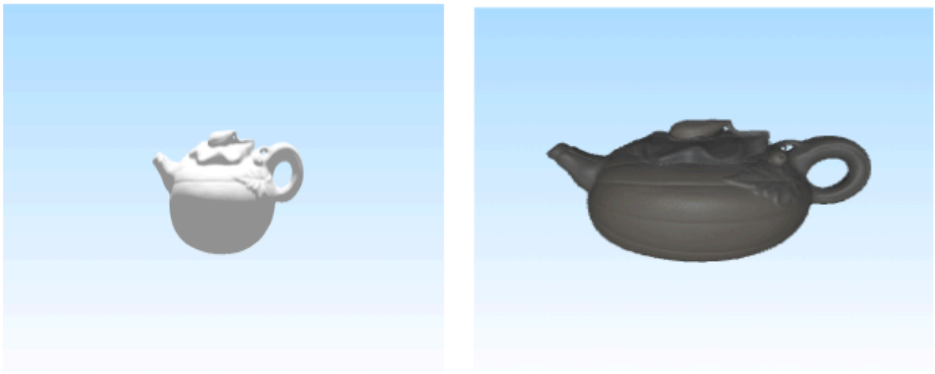

Figure 22. Meshes for Pot1 SDL

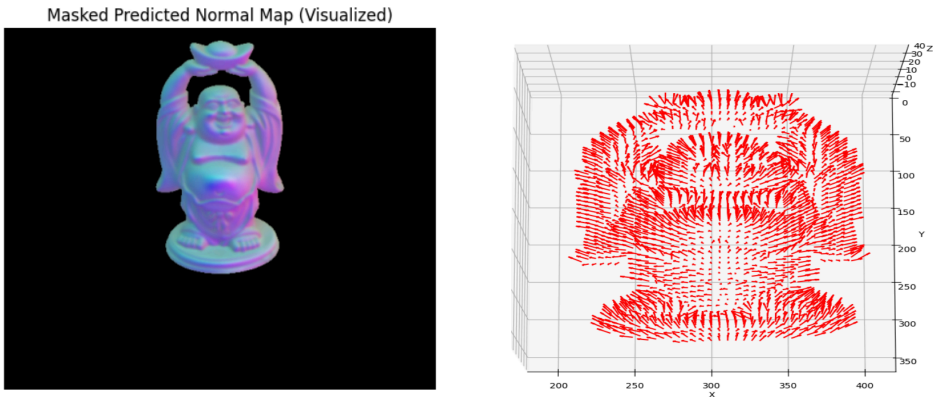

Figure 23. SDL based Buddha normal and normal directions (3D)

Figure 24. Meshes for Buddha SDL

Mode 3 SDPS

Figure 25 depicts SDPS based Pot1 normal and normal directions (3D). We can see the normal approximation is the best and so confirms the direction of the 3D plot.

Figure 26 depicts the meshes for Pot1 with SDPS, which further confirms that the normal approximation was the best as the meshes are excellent.

Similarly, Figure 27 depicts SDPS based Buddha normal and normal directions (3D). We can see the normal approximation is the best and so confirms the direction of the 3D plot.

Figure 28 depicts the meshes for Buddha with SDPS, which further confirms that the normal approximation was the best as the meshes are excellent..

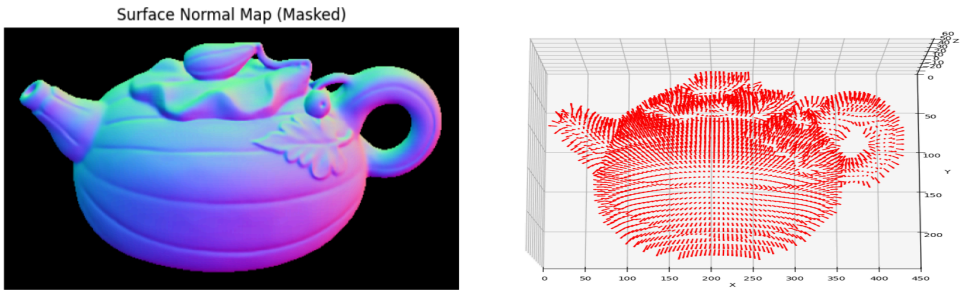

Figure 25. SDPS based Pot1 normal and normal directions (3D)

Figure 26. SDPS based meshes for Pot1.

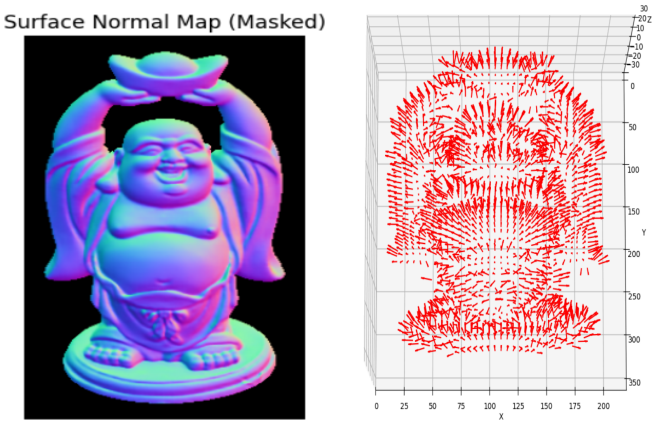

Figure 27. SDPS based Buddha normal and normal directions (3D)

Figure 28. SDPS based Buddha meshes

Finally, Figure 29 and Figure 30 depict the mean angular errors in terms of normal for different

methods for Pot1 and Buddha respectively. Where the mean angular error is:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}\cos^{-1}\left(\frac{\mathbf{n}_i^\top \hat{\mathbf{n}}_i}{\|\mathbf{n}_i\|\,\|\hat{\mathbf{n}}_i\|}\right)$$

Where,

- N is the total number of samples (pixels).
- n_i is the ground-truth normal vector at the i-th sample.
- $\hat{n}_i$ is the predicted normal vector at the i-th sample.
- $\| \cdot \|$ denotes the vector norm (length).
- $\cos^{-1}$ computes the angle (in radians) between the two vectors.

For Pot1 we can see that the worst error was with SVD. Best performance was with SDPS. SDL being shallow got excellent results and was the second best performer. For, Buddha, we can see that again SVD performed the worst. SDPS again here got the best results. Calibrated got second position.

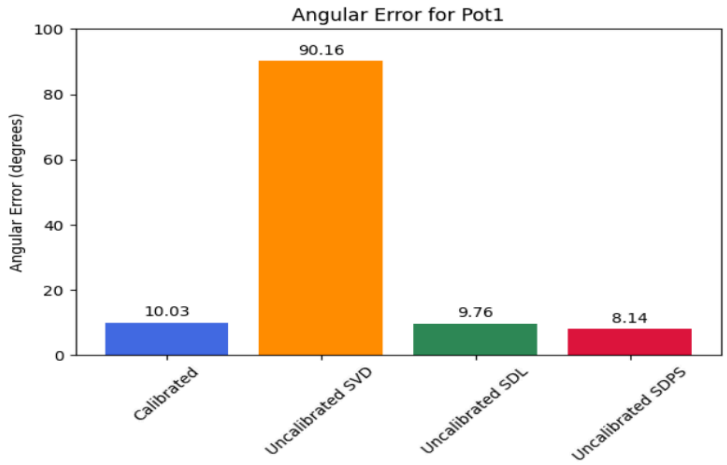

Figure 29. Angular errors for different method for Pot1

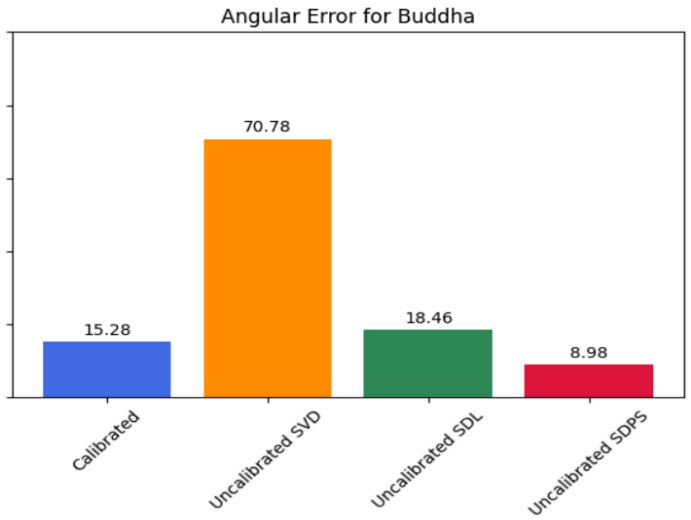

Figure 30. Angular errors for different method for Buddha

Discussion

So from this project, I learnt various aspects of photometric stereo, for example, calibrated and uncalibrated. For calibrated the least square approximation was enabled to get the normals and Frankot-Challappa was enabled to get the depth. For uncalibrated SVD based, simple deep learning based and Complex deep learning based SDPS methodology were enabled. In both cases SDPS performed even better than calibrated. The reasons behind it can be credited to the phenomenon that deep learning models are more robust learners when noise in images, noise in lights, shadow handling etc. But from these experiments we can see that even in uncalibrated stereo settings, deep learning models can have very good comparable performance and even better results than calibrated settings. For future work, we should explore more in deep neural networks and natural light settings. Moreover, depth estimation from videos should also be very promising.

References

[1] Ackermann, J., & Goesele, M. (2013). A survey of photometric stereo techniques. Foundations and Trends in Computer Graphics and Vision, 9(3–4), 149–254. https://doi.org/10.1561/0600000065

[2] [Woodham, R. J. (1980). Photometric method for determining surface orientation from multiple images. *Optical Engineering*, 19(1), 139–144.

[3] Basri, R., Jacobs, D. W., & Kemelmacher, I. (2007). Photometric stereo with general, unknown lighting. *International Journal of Computer Vision*, 72(3), 239–257.

[4] Chen, W., Han, Z., Xu, H., & Dong, J. (2019). PS-FCN: A flexible learning framework for photometric stereo. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 3–18).

[5] Chen, G., Han, K., Shi, B., Matsushita, Y., & Wong, K.-Y. K. (2019). SDPS-Net: Self-calibrating Deep Photometric Stereo Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[6] A. S. Georghiades, P. N. Belhumeur and D. J. Kriegman, "From few to many: illumination cone models for face recognition under variable lighting and pose," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 23, no. 6, pp. 643-660, June 2001, doi: 10.1109/34.927464.

[7] Boxin Shi, Zhipeng Mo, Zhe Wu, Dinglong Duan, Sai-Kit Yeung, and Ping Tan, "A Benchmark Dataset and Evaluation for Non-Lambertian and Uncalibrated Photometric Stereo", In IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), Volume 41, Issue 2, Pages 271-284,2019.

[8] R. T. Frankot and R. Chellappa, "A method for enforcing integrability in shape from shading algorithms," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 10, no. 4, pp. 439-451, July 1988, doi: 10.1109/34.3909.

[9] Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1996). *Computer Graphics: Principles and Practice* (2nd ed.). Addison-Wesley Professional.

[10] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva and G. Taubin, "The ball-pivoting algorithm for surface reconstruction," in IEEE Transactions on Visualization and Computer Graphics, vol. 5, no. 4, pp. 349-359, Oct.-Dec. 1999, doi: 10.1109/2945.817351.

[11] Hideki Hayakawa, "Photometric stereo under a light source with arbitrary motion," J. Opt. Soc. Am. A 11, 3079-3089 (1994)

Appendix

Codes done by me

All the codes are done by me except the last part in uncalibrated (mode 3) normal prediction. For this I have enabled the implementation of SDPS paper. Their git repo is: https://github.com/guanyingc/SDPS-Net.git

Now let's explain all of my implementation and codings:

Since the project has two parts
So:
Part 1 is in → Shagoto_CS217_Final_Project/calibrated
Part 2 is in → Shagoto_CS217_Final_Project/uncalibrated

First Let's Talk about the Calibrated Part.

Part1:

In Shagoto_CS217_Final_Project/calibrated
There are two folders

First → Yale

Yale Folder is: Shagoto_CS217_Final_Project/calibrated/Yale
This folder contains code for calibrated photometric stereo for Yale dataset

Main code is in :
Shagoto_CS217_Final_Project/calibrated/Yale/Final_Yale_Calibrated_Mesh_Generation.ipynb

Entire code has been written by me.

In the File here is the code that I applied:

For getting the light directions:

```
import numpy as np, imageio, re, glob

data_dir = "./yaleB_data/CroppedYalePNG/yaleB15_P00A"
```

```
file_list = sorted(glob.glob(data_dir + "*.png"))
images = []
light_dirs = []

for filename in file_list:

    name = filename.split("/")[-1]
    if "Ambient" in name:
        continue

    # Parsing azimuth (A) and elevation (E) from filename (e.g. A+035E+65)
    m = re.search(r"A([+-]\d+)E([+-]\d+)", name)

    if not m:
        continue
    az, el = int(m.group(1)), int(m.group(2))

    # Converting degrees to radians
    az_rad = np.deg2rad(az)
    el_rad = np.deg2rad(el)



    # Computing light direction (camera-centered coordinate frame)
    Lx =  np.cos(el_rad) * np.sin(az_rad)
    Ly =  np.sin(el_rad)
    Lz =  np.cos(el_rad) * np.cos(az_rad)

    light_dirs.append((Lx, Ly, Lz))

    # Loading image and normalize intensity
    img = imageio.imread(filename).astype(np.float64) / 255.0
    images.append(img)

images = np.stack(images, axis=0)
light_dirs = np.array(light_dirs)
```

Getting the normals and albedo:

```
# Reshaping image stack for linear solve: (N_images, H*W)
N_img, H, W = images.shape
I_stack = images.reshape(N_img, -1)  # each column is intensities for one pixel
L = light_dirs  # shape (N_images, 3)

# Solving for (rho * N) at each pixel via least-squares (pseudo-inverse)
# x = (L^T L)^{-1} L^T I  => shape (3, H*W)
L_pinv = np.linalg.pinv(L)
X = L_pinv.dot(I_stack)    # shape (3, H*W) = [ρN_x; ρN_y; ρN_z]

# Compute albedo and normals
rho = np.linalg.norm(X, axis=0)              # scalar albedo per pixel
nonzero = rho > 1e-8
N_stack = np.zeros_like(X)
N_stack[:, nonzero] = X[:, nonzero] / rho[nonzero]  # normalize to unit normals

# Reshape normal components to image grids
Nx = N_stack[0,:].reshape(H, W)
Ny = N_stack[1,:].reshape(H, W)
Nz = N_stack[2,:].reshape(H, W)
albedo = rho.reshape(H, W)
```

For Plotting:

```
import matplotlib.pyplot as plt

normals_vis = np.stack([-Nx, -Ny, Nz], axis=-1)
normals_vis = (normals_vis + 1) / 2
plt.imshow(normals_vis)
plt.title("Normals visualization with Nx as red")
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(6, 6))
plt.title("Albedo Map")
```

```
plt.imshow(albedo, cmap='gray')
plt.axis('off')
plt.show()
```

Then to get the depth from Frankot Chellappa:

```
# Compute gradients p, q
p = -Nx / (Nz + 1e-8)
q = -Ny / (Nz + 1e-8)

# Fourier integration (Frankot-Chellappa)
# Setup frequency grids
fx = np.fft.fftfreq(W) * 2*np.pi
fy = np.fft.fftfreq(H) * 2*np.pi
FX, FY = np.meshgrid(fx, fy)

P = np.fft.fft2(p)
Q = np.fft.fft2(q)

# Avoid zero frequency division
denom = (FX**2 + FY**2)
denom[0,0] = 1.0


# Compute Fourier-domain height
Z_freq = -(1j*FX * P + 1j*FY * Q) / denom
Z_freq[0,0] = 0  # set the DC component (offset) to zero
# Inverse FFT to get depth
Z = np.real(np.fft.ifft2(Z_freq))

# Create grid of (X,Y) pixel coordinates
X_coords = np.arange(W)
Y_coords = np.arange(H)
X, Y = np.meshgrid(X_coords, Y_coords)
# Stack into (H*W, 3) vertex array
vertices = np.stack([X, Y, Z], axis=-1).reshape(-1, 3)
```

Visualizing Normals:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Create grid of (X,Y) pixel coordinates
X_coords = np.arange(W)
Y_coords = np.arange(H)
X, Y = np.meshgrid(X_coords, Y_coords)
# --- Downsample for clarity ---
step = 2
X_plot = X[::step, ::step]
Y_plot = Y[::step, ::step]
Z_plot = Z[::step, ::step]
Nx_plot = Nx[::step, ::step]
Ny_plot = Ny[::step, ::step]
Nz_plot = Nz[::step, ::step]

Z_plot = -Z_plot

Nz_plot = Nz_plot


# --- Create 3D quiver plot ---
fig = plt.figure(figsize=(12, 9))


ax = fig.add_subplot(111, projection='3d')
ax.quiver(X_plot, Y_plot, Z_plot, -Nx_plot, -Ny_plot, Nz_plot, length=5.0, normalize=True, color='r')
ax.view_init(elev=70, azim=100)
ax.set_xlim(180,0)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Surface Normals (3D Quiver Plot)')
plt.tight_layout()
plt.show()
```

Lastly adding points in the triangles and generating the mesh:

```
faces = []
for i in range(H-1):
    for j in range(W-1):
        idx = i*W + j
        # vertices of the quad: [idx, idx+1, idx+W, idx+W+1]
        faces.append([idx,     idx+1,   idx+W    ])  # first triangle
        faces.append([idx+1,   idx+W+1, idx+W    ])  # second triangle
faces = np.array(faces, dtype=np.int32)


import open3d as o3d


mesh = o3d.geometry.TriangleMesh()
mesh.vertices = o3d.utility.Vector3dVector(vertices)
mesh.triangles = o3d.utility.Vector3iVector(faces)
mesh.compute_vertex_normals()  # for nicer shading
o3d.visualization.draw_geometries([mesh])

o3d.io.write_triangle_mesh("./generated_mesh/B15_MESH.ply", mesh)
```

Also just added the color to generate color mesh:

```
albedo_flat= albedo.reshape(-1)
albedo_norm = albedo_flat / albedo_flat.max()

# any value darker than thresh becomes true black
thresh = 0.29
albedo_norm[albedo_norm < thresh] = 0.0

colors = np.stack([albedo_norm]*3, axis=1)
mesh.vertex_colors = o3d.utility.Vector3dVector(colors)

o3d.io.write_triangle_mesh("./generated_mesh/B15_MESH_clr.ply", mesh)
```

Second → DiliGenT

DiliGenT Folder is: Shagoto_CS217_Final_Project/calibrated/DiliGenT
This folder contains code for calibrated photomoetric stereo for DiliGenT dataset
Main code is in : **Shagoto_CS217_Final_Project/calibrated/DiliGenT/from lights.ipynb**

Entire code has been written by me.

Here is the entire code to first get the normals, then depth and then generating the mesh:

```
import cv2
import numpy as np
import os
import open3d as o3d
from numpy.fft import fft2, ifft2
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def frankot_chellappa(p, q):
    h, w = p.shape
    wx = np.tile(np.fft.fftfreq(w), (h, 1))
    wy = np.tile(np.fft.fftfreq(h), (w, 1)).T

    Px = fft2(p)
    Qy = fft2(q)

    denom = (2 * np.pi * wx)**2 + (2 * np.pi * wy)**2
    denom[0, 0] = 1  # avoid division by zero

    Z = (-1j * 2 * np.pi * wx * Px - 1j * 2 * np.pi * wy * Qy) / denom
    z = np.real(ifft2(Z))
    return z

def depth_from_normals(normals, mask):
    nz = normals[:, :, 2]
    nz[nz == 0] = 1e-6

    p = -normals[:, :, 0] / nz
    q = -normals[:, :, 1] / nz

    p[~mask] = 0
```

```
    q[~mask] = 0

    depth = frankot_chellappa(p, q)
    return depth

def save_mesh(depth, normals, mask, filename):
    h, w = depth.shape
    xx, yy = np.meshgrid(np.arange(w), np.arange(h))
    points = np.stack((xx, yy, depth), axis=2).reshape(-1, 3)
    normals = normals.reshape(-1, 3)
    mask = mask.flatten()

    points = points[mask]
    normals = normals[mask]

    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(points)
    pcd.normals = o3d.utility.Vector3dVector(normals)

    pcd.estimate_normals()
    pcd.orient_normals_consistent_tangent_plane(10)
    distances = pcd.compute_nearest_neighbor_distance()
    avg_dist = np.mean(distances)
    radius = 1.5 * avg_dist

    mesh = o3d.geometry.TriangleMesh.create_from_point_cloud_ball_pivoting(
        pcd, o3d.utility.DoubleVector([radius, radius * 2])
    )
    o3d.io.write_triangle_mesh(filename, mesh)

def estimate_normals(intensity_stack, light_dirs, mask):
    h, w, n = intensity_stack.shape
    I = intensity_stack.reshape(-1, n).T  # shape: (n, H*W)
    mask_flat = mask.flatten()

    # Only solve for pixels inside the mask
    I_masked = I[:, mask_flat]  # shape: (n, num_mask_pixels)
    L = light_dirs  # shape: (n, 3)
```

```
    L_pinv = np.linalg.pinv(L)
    G = L_pinv @ I_masked  # shape: (3, num_mask_pixels)

    norm = np.linalg.norm(G, axis=0) + 1e-6
    normals = G / norm  # shape: (3, num_mask_pixels)

    normal_map = np.zeros((h * w, 3), dtype=np.float32)
    normal_map[mask_flat] = normals.T
    normal_map = normal_map.reshape(h, w, 3)

    albedo = norm
    albedo_map = np.zeros((h * w,), dtype=np.float32)
    albedo_map[mask_flat] = albedo
    albedo_map = albedo_map.reshape(h, w)

    return normal_map, albedo_map

# ==== MAIN ====
image_folder = "./DiLiGenT/pmsData/pot1PNG"
n_images = 96

# Load mask
mask = cv2.imread(os.path.join(image_folder, "mask.png"), 0)
mask = (mask > 200)

# Load light directions and intensities
light_dirs = np.loadtxt(os.path.join(image_folder, "light_directions.txt"))  # (96, 3)
light_intensities = np.loadtxt(os.path.join(image_folder, "light_intensities.txt"))  # (96, 3)

# Load grayscale intensity images from RGB and normalize
intensity_images = []
for i in range(1, n_images + 1):
    img_path = os.path.join(image_folder, f"{i:03d}.png")
    rgb = cv2.imread(img_path).astype(np.float32) / 255.0  # shape: (H, W, 3)

    # Normalize each channel by corresponding light intensity
    light_intensity = light_intensities[i - 1]  # shape: (3,)
    norm_rgb = rgb / (light_intensity + 1e-6)  # avoid divide by zero
```

```
    # Convert to grayscale after normalization
    gray = 0.2989 * norm_rgb[:, :, 2] + 0.5870 * norm_rgb[:, :, 1] + 0.1140 * norm_rgb[:, :, 0]
    intensity_images.append(gray)

intensity_stack = np.stack(intensity_images, axis=-1)

# Estimate normals and albedo
normal_map, albedo_map = estimate_normals(intensity_stack, light_dirs, mask)
normal_mapk=normal_map
# smoothing
normal_map = cv2.GaussianBlur(normal_map, (5, 5), 0)

# Visualization
normal_vis = (normal_map + 1) / 2.0
normal_vis[~mask] = 0

albedo_vis = albedo_map / (albedo_map.max() + 1e-6)
albedo_vis[~mask] = 0

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.title("Albedo Map")
plt.imshow(albedo_vis, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title("Normal Map")
plt.imshow(normal_vis)
plt.axis('off')
plt.tight_layout()
plt.show()

# Flip for correct orientation
normal_map[:, :, 1] *= -1
normal_map[:, :, 2] *= -1

# Estimate depth
depth = depth_from_normals(normal_map, mask)
H, W = depth.shape
```

```
# Quiver plot of normals
X_coords, Y_coords = np.arange(W), np.arange(H)
X, Y = np.meshgrid(X_coords, Y_coords)

step = 5
X_plot = X[::step, ::step]
Y_plot = Y[::step, ::step]
Z_plot = -depth[::step, ::step]
Nx_plot = normal_map[::step, ::step, 0]
Ny_plot = normal_map[::step, ::step, 1]
Nz_plot = -normal_map[::step, ::step, 2]
mask_small = mask[::step, ::step]

fig = plt.figure(figsize=(12, 9))
ax = fig.add_subplot(111, projection='3d')
ax.quiver(X_plot[mask_small], Y_plot[mask_small], Z_plot[mask_small],
          Nx_plot[mask_small], Ny_plot[mask_small], Nz_plot[mask_small],
          length=10.0, normalize=True, color='r')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Surface Normals (3D Quiver Plot)')
ax.view_init(elev=70, azim=90)
ax.set_xlim(550, 100)
plt.tight_layout()
plt.show()

# Save mesh
os.makedirs("./generated_mesh_from_light", exist_ok=True)
save_mesh(depth, normal_map, mask, "./generated_mesh_from_light/pot_mesh.ply")
```

To get angular error I added the following code:

```
# Load ground truth normals
normal_gt_path = os.path.join(image_folder, "normal.txt")
normal_gt = np.loadtxt(normal_gt_path)  # shape: (H*W, 3)
normal_gt = normal_gt.reshape(mask.shape[0], mask.shape[1], 3)  # reshape to (H, W, 3)
```

```
# Normalize predicted and ground-truth normals
norm_pred = normal_mapk / (np.linalg.norm(normal_mapk, axis=2, keepdims=True) + 1e-6)
norm_gt = normal_gt / (np.linalg.norm(normal_gt, axis=2, keepdims=True) + 1e-6)

# Dot product per pixel
dot_product = np.sum(norm_pred * norm_gt, axis=2)
dot_product = np.clip(dot_product, -1.0, 1.0)  # Clamp for safety

# Angular error in degrees
angular_error = np.arccos(dot_product) * (180.0 / np.pi)

# Mask out background
angular_error[~mask] = np.nan

mean_angular_error = np.nanmean(angular_error)
print(f"Mean Angular Error: {mean_angular_error:.2f} degrees")
```

Also just to add color in the mesh I add:

```
def save_mesh(depth, normals, mask, filename, colors=None):
    h, w = depth.shape
    xx, yy = np.meshgrid(np.arange(w), np.arange(h))
    points = np.stack((xx, yy, depth), axis=2).reshape(-1, 3)
    normals = normals.reshape(-1, 3)
    mask = mask.flatten()

    points = points[mask]
    normals = normals[mask]

    # Create point cloud
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(points)
    pcd.normals = o3d.utility.Vector3dVector(normals)

    if colors is not None:
        pcd.colors = o3d.utility.Vector3dVector(colors)
```

```
    # Ball pivoting for mesh
    pcd.estimate_normals()
    pcd.orient_normals_consistent_tangent_plane(10)
    distances = pcd.compute_nearest_neighbor_distance()
    avg_dist = np.mean(distances)
    radius = 2 * avg_dist

    mesh = o3d.geometry.TriangleMesh.create_from_point_cloud_ball_pivoting(
        pcd, o3d.utility.DoubleVector([radius, radius * 2])
    )

    o3d.io.write_triangle_mesh(filename, mesh)
```

```
save_mesh(depth, normal_map, mask, "./generated_mesh_from_light/pot_colored_mesh.ply",
colors=colors)
```

Part2:

In Shagoto_CS217_Final_Project/uncalibrated
There is one folder because uncalibrated is done for DiliGenT only as it has ground truth normals
Folder: Shagoto_CS217_Final_Project/uncalibrated/diligent

Now this folder → Shagoto_CS217_Final_Project/uncalibrated/diligent has 3 modes

First SVD:

main code: **Shagoto_CS217_Final_Project/uncalibrated/diligent/SVD_based-final.ipynb**

Entire code has been done by me.

So here is the code I wrote to get normals by SVD:

```
import os
import numpy as np
import cv2
import glob
```

```
# Set folder path
image_folder = "./DiLiGenT/pmsData/pot1PNG/"
mask_path = os.path.join(image_folder, "mask.png")

# Load binary mask
mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
assert mask is not None, "Mask file not found."
mask = mask > 10
H, W = mask.shape
mask_flat = mask.flatten()

# Get list of image files (excluding mask and normal_gt)
image_files = sorted(glob.glob(os.path.join(image_folder, "*.png")))
image_files = [f for f in image_files if 'normal' not in f and 'mask' not in f]
num_images = len(image_files)

# Load images and apply mask
pixels_in_mask = np.sum(mask)
intensity_matrix = np.zeros((num_images, pixels_in_mask), dtype=np.float32)  # shape (num_images,
N_pixels)

for i, img_file in enumerate(image_files):
    img = cv2.imread(img_file, cv2.IMREAD_GRAYSCALE).astype(np.float32) / 255.0
    assert img is not None, f"Image not found: {img_file}"
    intensity_matrix[i, :] = img.flatten()[mask_flat]

# Helps reduce lighting variations
intensity_matrix -= intensity_matrix.mean(axis=1, keepdims=True)

# SVD
U, S, Vt = np.linalg.svd(intensity_matrix, full_matrices=False)

# Take top-3 components
U3 = U[:, :3]
S3 = np.diag(S[:3])
V3 = Vt[:3, :]

# Estimated normals (unnormalized)
```

```
B_hat = S3 @ V3  # Shape (3, N_pixels)

# Normalize estimated normals
normals_est = B_hat / (np.linalg.norm(B_hat, axis=0, keepdims=True) + 1e-8)

# Reconstruct full normal map
normal_map = np.zeros((H * W, 3), dtype=np.float32)
normal_map[mask_flat] = normals_est.T  # shape (H*W, 3)
normal_map = normal_map.reshape((H, W, 3))

normal_mapk=normal_map
# Normalize to [0, 1] for visualization
normal_map_vis = (normal_map + 1) / 2.0
mask_3c = np.stack([mask] * 3, axis=-1)
normal_map_vis[~mask_3c] = 0.0

import matplotlib.pyplot as plt


plt.figure(figsize=(6, 6))
plt.imshow(normal_map_vis)
plt.title("Surface Normals")
plt.axis("off")
plt.show()
```

Then depth from normals using frankot Chellappa, mesh generation, color mesh generation, angular loss calculation, albedo plotting, and normal visualizations codes are the same as previous.

Second: Simple Deep learning

Folder: Shagoto_CS217_Final_Project/uncalibrated/diligent/simple deep learning
main code: **Shagoto_CS217_Final_Project/uncalibrated/diligent/simple deep learning/simple depth learning.ipynb**

Entire code has been done by me.

Here is my code to train a simple model:

```python
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, ConcatDataset
from torchvision import transforms as T
from PIL import Image
import numpy as np
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
from scipy.io import loadmat

class PhotometricStereoDataset(Dataset):
    def __init__(self, folder):
        self.folder = folder
        self.image_paths = [os.path.join(folder, f"{i:03}.png") for i in range(1, 97)]
        self.mask_path = os.path.join(folder, "mask.png")
        self.normal_gt_mat_path = os.path.join(folder, "Normal_gt.mat")
        self.to_tensor = T.ToTensor()

    def __len__(self):
        return 1

    def __getitem__(self, idx):
        imgs = [self.to_tensor(Image.open(p).convert("L")) for p in self.image_paths]
        imgs = torch.cat(imgs, dim=0)

        mask = self.to_tensor(Image.open(self.mask_path))[0]

        mat = loadmat(self.normal_gt_mat_path)
        normal_np = mat['Normal_gt']
        normal = torch.from_numpy(normal_np).permute(2, 0, 1).float()

        if normal.max() > 1.0 or normal.min() < -1.0:
            normal = normal * 2 - 1

        return imgs, normal, mask
```

```
class SimpleNormalEstimator(nn.Module):
    def __init__(self, in_channels=96, out_channels=3):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(in_channels, 64, 3, padding=1), nn.ReLU(),
            nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(),
            nn.Conv2d(128, 64, 3, padding=1), nn.ReLU(),
            nn.Conv2d(64, out_channels, 3, padding=1),
            nn.Tanh()
        )

    def forward(self, x):
        return self.net(x)

def normal_loss(pred, target, mask):
    pred = F.normalize(pred, dim=1)
    target = F.normalize(target, dim=1)


    mask = mask.unsqueeze(1)

    loss = 1 - (pred * target).sum(dim=1, keepdim=True)  # [B, 1, H, W]

    return (loss * mask).mean()


train_folders = [
    "../DiLiGenT/pmsData/catPNG/",
    "../DiLiGenT/pmsData/cowPNG/",
    "../DiLiGenT/pmsData/buddhaPNG/",
    "../DiLiGenT/pmsData/ballPNG/",
    "../DiLiGenT/pmsData/bearPNG/",
    "../DiLiGenT/pmsData/gobletPNG/",
    "../DiLiGenT/pmsData/readingPNG/",
     "../DiLiGenT/pmsData/pot2PNG/"
]

train_datasets = [PhotometricStereoDataset(folder) for folder in train_folders]
train_loader = DataLoader(ConcatDataset(train_datasets), batch_size=1, shuffle=True)
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SimpleNormalEstimator().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)


epochs = 20
for epoch in range(epochs):
    model.train()
    total_loss = 0
    for imgs, normals, mask in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
        imgs, normals, mask = imgs.to(device), normals.to(device), mask.to(device)
        optimizer.zero_grad()
        pred = model(imgs)
        loss = normal_loss(pred, normals, mask)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1} Loss: {total_loss / len(train_loader):.4f}")


torch.save(model.state_dict(), "normal_estimator_pot1.pth")
set_seed(42)
model = SimpleNormalEstimator().to(device)
model.load_state_dict(torch.load("normal_estimator_pot1.pth"))



test_folder = "../DiLiGenT/pmsData/pot1PNG/"
test_dataset = PhotometricStereoDataset(test_folder)
imgs, _, _ = test_dataset[0]
imgs = imgs.unsqueeze(0).to(device)

model.eval()
with torch.no_grad():
    pred = model(imgs)[0].cpu()
    predk=pred
    pred = (pred + 1) / 2
```

After the normal prediction all the next codes such as depth from frankot-chellappa, angular error, and visualizations are the same as previous.

Third SDPS:

Folder → Shagoto_CS217_Final_Project/uncalibrated/diligent/Complex_Deep_Learning

To generate a complex model this code has been taken from https://github.com/guanyingc/SDPS-Net.git repository.

Then the only change made was to get the predicted normal, so the file were changed in the repository is:

test_stage2.py and the function I changed was the following:

```
def prepareSave(args, data, pred_c, pred,i):
    results = [data['img'].data, data['m'].data, (data['n'].data+1) / 2]
    if args.s2_est_n:
        masked_pred2 = pred['n'] * data['m'].data.expand_as(pred['n'].data)
        save_raw_normal(masked_pred2, f"./shagoto_predicted_normal/normal_shagoto{i}.npy")
        pred_n = (pred['n'].data + 1) / 2
        masked_pred = pred_n * data['m'].data.expand_as(pred['n'].data)
        res_n = [masked_pred, data['error_map']]
        results += res_n
        #save_raw_normal(masked_pred, f"./shagoto_predicted_normal/normal_shagoto{i}.npy")

    nrow = data['img'].shape[0]
    return results, nrow
```

Then extra file was created and coded by me:
Shagoto_SDPS PREDICTED NORMAL_and_ MESH.ipynb

So, After getting the normal then all the codes to get the depth, errors, visualizations are the same as my previous codes.

Lastly to plot the comparison I added:

```
import matplotlib.pyplot as plt
import numpy as np

methods = ['Calibrated', 'Uncalibrated SVD', 'Uncalibrated SDL',
'Uncalibrated SDPS']
```

```
pot1_errors = [10.03, 90.16, 9.76, 8.14]
buddha_errors = [15.28, 70.78, 18.46, 8.98]

# Define different colors for each method
method_colors = ['royalblue', 'darkorange', 'seagreen', 'crimson']

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(12, 5), sharey=True)

# Plot for Pot1
bars1 = axs[0].bar(methods, pot1_errors, color=method_colors)
axs[0].set_title('Angular Error for Pot1')
axs[0].set_ylabel('Angular Error (degrees)')
axs[0].tick_params(axis='x', rotation=45)
for bar in bars1:
    yval = bar.get_height()
    axs[0].text(bar.get_x() + bar.get_width() / 2, yval + 1,
f'{yval:.2f}', ha='center', va='bottom')

# Plot for Buddha
bars2 = axs[1].bar(methods, buddha_errors, color=method_colors)
axs[1].set_title('Angular Error for Buddha')
axs[1].tick_params(axis='x', rotation=45)
axs[0].set_ylim(0, 90)
axs[1].set_ylim(0, 100)

for bar in bars2:
    yval = bar.get_height()
    axs[1].text(bar.get_x() + bar.get_width() / 2, yval + 1,
f'{yval:.2f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()
```

So, all the codes in the project are done by me except the mode 3 in uncalibrated, which is to generate normals from a complex deep learning work.