Summer Undergraduate Research Opportunities

Emile Okada
University of Cambridge

July, 2016

1 Week 1

1.1 Reading

I started the week reading Chapter 1 section 2 of Charles L. Epstein's "Introduction to the Mathematics of Medical Imaging". It covered how to reconstuct a 2D convex object from the shadows of an object. If $h(\theta)$ is the shadow function as described in the book (essentially the distance of the support line in direction $(-\sin(\theta), \cos(\theta))$ from the origin), then the convex hull can be parameterized by

$$(x(\theta), y(\theta)) = h(\theta) \cdot (\cos(\theta), \sin(\theta)) + h'(\theta) \cdot (-\sin(\theta), \cos(\theta)). \tag{1}$$

We can extend this idea to 3D by considering slices of the object. Fix some vector $\mathbf{v}$ and then consider the collection of planes perpendicular to $\mathbf{v}$. In each of the planes one can use the 2D method to construct a 2D convex hull of the intersection of the object with the plane. Stringing all these 2D slices together then gives a rough reconstruction of the 3D object from its shadows.

I also spent some time reading up on the TV transform and scale spaces, to get a rough idea of which project I'd like to do. I ended up going with the tomography project, but spend roughly 1.5 days doing reading for the other project.

1.1.1 Remaining tasks

The above method only allows for 'slicewise convex' reconstructions. While better than convex, this doesn't ustilize all the data available from the shadows e.g. see the rabit ears below. We know the two ears are two separate 'blobs' and we can tell this from the shadow, but this information is nonetheless lost in the reconstruction. I will try to find ways on how to improve on this (so that e.g. if we scan a human we can see two legs rather than one large blob).

1.2 Coding

On Thursday I started coding. To have some examples to work with I used the Wolfram Mathematica ExampleData function to obtain a 3D model of a rabit.

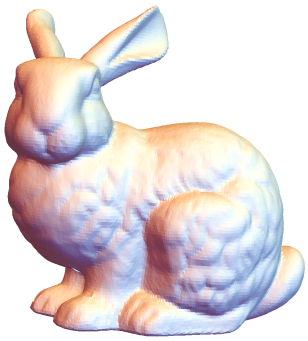

Figure 1: 3D model of a rabit from Mathematica ExampleData.

I then took its orthogonal projection and binarized the image to obtain shadows of the rabit from 10 different angles.

Figure 2: Shadow of rabit with $\theta = 0$.

Figure 3: Shadow of rabit with $\theta = \pi/2$.

I then finished writing the code for reconstructing the object from its shadows in python (see page 7 or my github) on Friday.

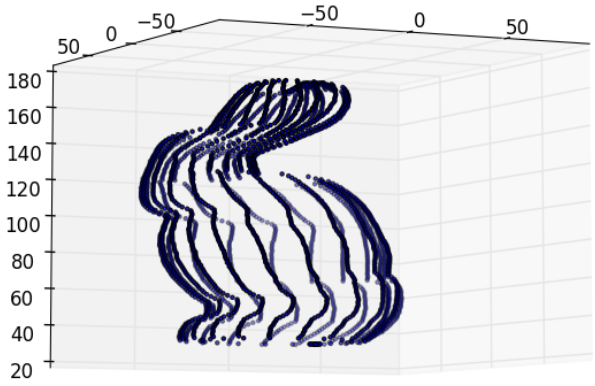

Figure 4: Reconstruction of the rabit using the script on page 7.

1.2.1 Challenges

I had some problems originally with taking the derivative of the shadow function since the input was a bit noisy. To fix this problem I convolved the data with a discrete gaussian (binomial distribution) to smooth out the noise. This worked well with the example data. However, it remains to see whether it will work well enough for the actual real life pictures which will probably be a lot more noisy.

1.2.2 Remaining tasks

There are currently a lot more data points than I need. At the moment I'm calculating the shadow function fow each row of the image matrix. This results in a lot of points which causes the rendering to take some time. I can probably get away with a lot fewer points in the z direction. I also plan on adding a polygonal mesh so it looks a bit nicer. Lastly, I should probably also do some

smoothing in the z direction to fix the sudden cutoffs that tend to occur when the variations in the 3D object are smaller than the 'resolution' in the z-direction.

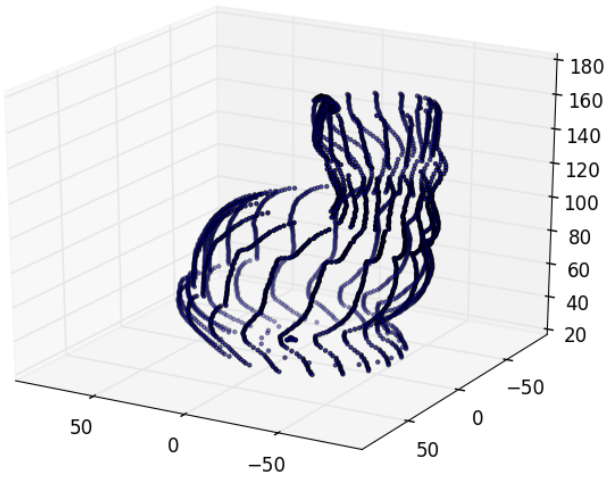

Figure 5: Cutoff of rabits ears. Can probably be fixed by smoothing the data in the z direction.

1.3 Building

I haven't begun building yet, but I now have a rough idea of the the set should look like. At the moment I'm thinking of creating a simple turn table (rotating platform) on which a person stands. I'll then place some bright lights in front for them and a big white sheet of paper behind on which to shine their shadow. A camera is then placed behind the paper to capture the shadows.

1.3.1 Remaining tasks

I'll create a proper design this week and make a basic prototype on a smaller scale hopefully by next week.

2 Week 2

2.1 Reading

This week I had a chat with Martin and Matthias on how to resolve objects like rabit ears (i.e. disjoint unions of convex objects). They suggested a 'back projection' method that is a generalization (or in some sense a generalization of the 'dual') of the method from week 1. The methods can be thought of as follows: imagine standing inside a circle with walls around. Every, say, $\pi/8$ radians there is a light source that creates a shadow diagonally opposite. Now extend each shadow out of the wall to the opposite side, to create 'cylinders' with cross sections with the shape of the shadow. We then intersect these cylinders to get the reconstruction. In some sense this method utilizes all the information available and can distinguish between legs, rabit ears and other disjoint unions

of convex blobs. We also don't need to differentiate anything with this method which is always nice.

2.1.1 Remaining tasks

Read up on TV smoothing for 3D surfaces to make smoother reconstructions.

2.2 Coding

Before I implemented the new method I added a polygonal mesh to the original method.

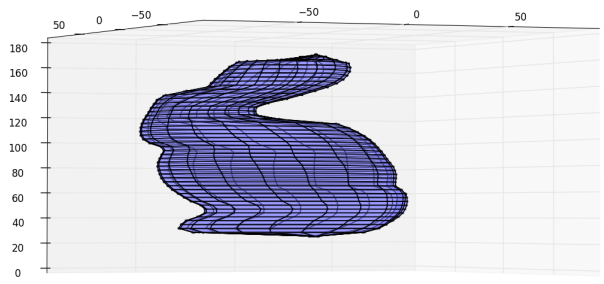

Figure 6: Original reconstruction with polygonal mesh added.

I also implemented the new method. It works by reducing it to the 2D version of the problem and doing the reconstruction by slices like the previous method. It is slightly more computationally expensive (the 'cyclinders' are represented in arrays and the intersection is done by pointwise multiplication). However, there results are a lot better.

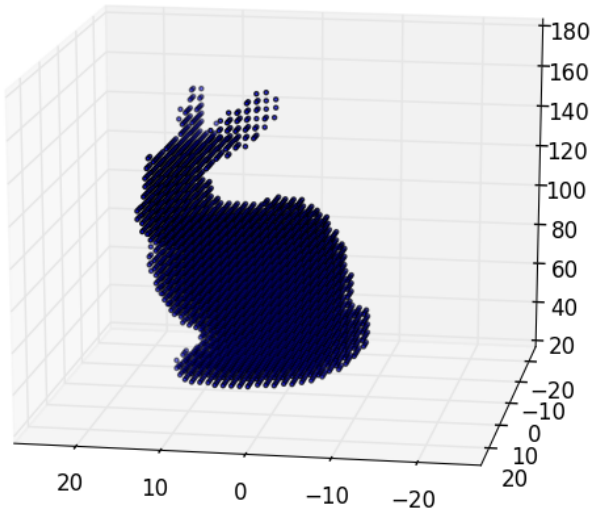

Figure 7: Reconstruciton of rabit with two disjoint ears.

I also implemented a method that only displays the boundary of the reconstructions by taking the morphological perimeter of each slice.

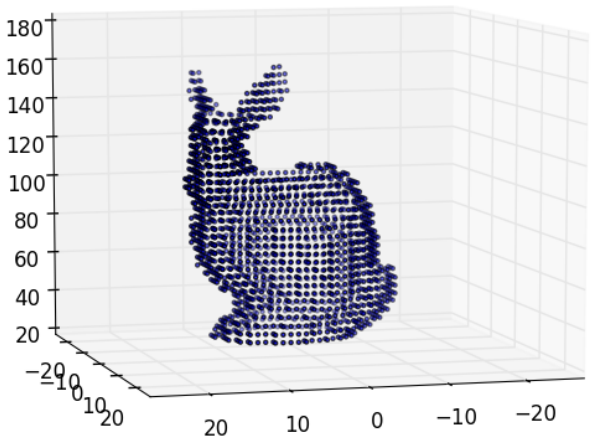

Figure 8: Boundary of rabit reconstruciton.

2.2.1 Remaining tasks

It would be good to add a mesh to this method as well and maybe get some lighting to make it look more realistic. Also need to implement TV method, but will probably wait to do that until I have something that works first.

2.3 Building

I've done some reseach on turntables for humans. I've found quite a few DIY turntables that are cheap and relatively straightforward to make (see this, this and this). I also looked for some ready made ones, but the ones I found were either too expensive (âĆň650) or made for mannequins so can't take the full load of a human. It seems like the best option would be to make my own. I'll contact the engineering department to ask if there is any way I could use some of their tools (possibly under the supervision of an engineering friend) to make the turntable myself. I've also looked at lighting and it seems quite cheap so shouldn't be a problem. Hopefully I'll start ordering the equipment this week.

3 Code

3.1 3D reconstruction from shadows

```
from __future__ import division
from PIL import Image

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

import math
import re
import os
from misc import *

class image3d:
    def __init__(self,files):
        (self.width, self.height) = Image.open(files[0]).size
        self.mid = self.width//2

        self.resolution = len(files)

        self.vertical_pixel_spacing = self.height//50
        self.xy_scaling = 5

        self.scaled_width = self.width//self.xy_scaling
        self.scaled_mid = self.scaled_width//2

        self.xy = self.scaled_mid*0.75
        self.aspect = 1

        self.shadow_bands = [[] for i in range(self.height)]
        self.x, self.y, self.z = [], [], []

        #Determine the shadow function
        for f in files:
            img = Image.open(f)
            img = self.preprocess(img)
            img_data = self.img_to_array(img)
            for i, row in enumerate(img_data):
                if i % self.vertical_pixel_spacing == 0:
                    self.shadow_bands[i].append(self.read_bands(row))

        for i in range(self.height):
                if i % self.vertical_pixel_spacing == 0:
                    #Add points
                    #coords = self.get_coords(self.intersected_area(i),i)
                    coords = self.get_coords(self.morphological_perimeter(self.
```

```
                    self.x.append(coords[0])
                    self.y.append(coords[1])
                    self.z.append(coords[2])

        self.sparse_x = self.x
        self.sparse_y = self.y
        self.sparse_z = self.z

        #Create figure
        fig = plt.figure()
        ax = fig.add_subplot(111,projection='3d')
        ax.scatter(self.flat_x,self.flat_y,self.flat_z,marker='.')
        plt.xlim([-self.xy,self.xy])
        plt.ylim([-self.xy,self.xy])
        plt.show()

    @property
    def flat_x(self):
        return [x_coord for row in self.sparse_x for x_coord in row]

    @property
    def flat_y(self):
        return [y_coord for row in self.sparse_y for y_coord in row]

    @property
    def flat_z(self):
        return [z_coord for row in self.sparse_z for z_coord in row]

    def preprocess(self,img):
        """Convert image to greyscale and binarize image"""

        img = img.convert('L')
        out = img.point(lambda i: 0 if i>255/2 else 1)
        return out

    def img_to_array(self,img):
        """Convert Image object to matrix"""

        l = list(img.getdata())
        (w,h) = img.size
        return [l[w*i:w*(i+1)] for i in range(h)]

    def read_bands(self,img_row):
        differences = [t-s for s,t in zip(img_row,img_row[1:])]
        left_edges = [i+1-self.mid for i,el in enumerate(differences) if el ==
        right_edges = [i-self.mid for i,el in enumerate(differences) if el == -

        if len(left_edges)>len(right_edges):
            right_edges.append(len(img_row)-1)
        elif len(left_edges)<len(right_edges):
```

```
            left_edges.append(0)

        if len(left_edges) == len(right_edges):
            return zip(left_edges, right_edges)
        return []

    def in_band(self, theta, m, n, bands):
        d = -(m-self.scaled_mid)*math.sin(theta)+(n-self.scaled_mid)*math.cos(t
        for lower, upper in bands:
            if lower <= self.xy_scaling*d <= upper:
                return True
        return False

    def banded_matrix(self,theta,bands):
        return np.array([[1 if self.in_band(theta,i,j,bands) else 0 for i in ra

    def intersected_area(self,z_coord):
        print str(100*z_coord/self.height) + "%"
        intersection = np.array([[1 for i in range(self.scaled_width)] for j in
        for i, bands in enumerate(self.shadow_bands[z_coord]):
            intersection = intersection * self.banded_matrix(math.pi*i/self.res
        return intersection

    def get_coords(self,matrix,z_coord):
        xy = map(list,np.array(np.nonzero(matrix))-self.scaled_mid)
        return xy + [[self.height-z_coord for i in range(len(xy[0]))]]

    def morphological_perimeter(self,matrix):
        perimeter_matrix = [[0 for i in range(self.scaled_width)] for j in rang
        adj = [(1,0),(-1,0),(0,1),(0,-1),(1,1),(1,-1),(-1,1),(-1,-1)]
        for i in range(self.scaled_width):
            for j in range(self.scaled_width):
                if matrix[i][j] == 1:
                    for x,y in adj:
                        if 0 <= i-x < self.scaled_width and 0 <= j-y < self.sca
                            if matrix[i-x][j-y] == 0:
                                perimeter_matrix[i][j] = 1

        return perimeter_matrix

files = ['./pictures/'+f for f in os.listdir('./pictures/')]
files = sorted(files, key=lambda x: int(re.search(r'(\d+)\.jpg',x).group(1)))

space_shuttle = image3d(files)
```

3.2 Miscellaneous functions

```
from __future__ import division
import math
```

```
def pad_left(l,k):
    return l[-k:] + l

def pad_right(l,k):
    return l + l[:k]

def transpose(l):
    return zip(*l)

def binomial(n,k):
    return math.factorial(n)/(math.factorial(k)*math.factorial(n-k))

def binomial_density(n):
    return [binomial(n,k)/(2**n) for k in range(n+1)]

def dot_product(lista,listb):
    return sum(a*b for a,b in zip(lista,listb))

def differentiate(y, step_size):
    y_temp = y[-1:] + y + y[:1]
    return [(t - s)/(2*step_size) for s, t in zip(y_temp, y_temp[2:])]

def list_convolve(l,kernel):
    left = (len(kernel)-1)//2
    right = len(kernel)//2
    l_temp = l
    if left > 0:
        l_temp = l[-left:] + l + l[:right]
    elif right > 0:
        l_temp = l + l[:right]
    if kernel != []:
        return [dot_product(l_temp[i:i+len(kernel)],kernel) for i in range(len(
    return l
```