**Computational Radon Transform and its Applications**

Cameron J. Nottingham

**ABSTRACT**

The Radon transform and its inverse have broad uses in the field of computational imaging, particularly in the medical field with the invention of the Computed Tomography Scan. Finding and implementing quick and efficient ways to compute the inverse Radon transform of a sinogram is at the forefront of research in computational imaging. Even without advanced methods, there exist simple way to programmatically explore the Radon transform and its inverse.

**INTRODUCTION**

One of the most important breakthroughs in medicine over the last half century is the introduction of the Computed Tomography Scan (CT Scan). Tomography, simply put, is imaging by sectioning. A tomographic image allows one to see inside of the object without cutting it. Tomographic images of objects also eliminate the problem of superposition with flat image, where things in the foreground obstruct things in the background. A CT scan is vitally important in being able to see organs and other objects inside the human body without using an invasive procedure.

A CT Scanning machine does not generate these tomographic images directly. A CT Scan is a series of X-Rays taken at varying angles around the body. Each of these images generated is a projection of the object at the angle the image was taken. These projections are generated using a series of parallel X-rays at a set distance from the object being scanned. These X-Rays go through the patient and are absorbed by a detector, which measures the amount of absorption that each X-Ray had through its part of the object. If this is done at varying angles around the object, a full absorption map for the angles measured can be generated. The question then becomes: How do we construct a tomographic image from this data? Well, one of the greatest breakthroughs in medical imaging technology was the discovery that when all of the radiographic projections of an object are added together, the resulting image is equivalent to the Radon transform of the object [1]. Thus, there is great interest in being able to find the inverse of the Radon transform for use in image reconstruction.

**MATH AND PROCEDURES**

The Radon transform was first introduced in 1917 by the Austrian mathematician Johann Radon. There are multidimensional variants of the Radon transform, and the multidimensional Radon transform is the one used in CT Scans. However, the mathematics involved becomes much more complex, though it remains fundamentally the same. Thus, for simplicity, only the second dimensional transform and its inverse will be discussed.

Simply put, the Radon transform is an integral transform that takes a series of line integrals through a function f(x,y) at different offsets from the origin. It is defined as follows:

Equation 1: Radon Transform

Where δ is the Dirac delta function, s is the distance from the origin, and theta is the angle from the origin [1].

The value of a point in the p(s,θ) distribution is the value of the line integral of f(x,y) taken along the curve of the parametrized straight line defined by s and θ. In other words, each point in p(s,θ) represents the sum of the values of f(x,y) that are intersected by the straight line defined by s and θ. If a distance *s* is fixed and theta is allowed to run from 0 to 180 degrees, then a complete mapping of the function f(x,y) to in p(s,θ) is generated. When an object is scanned using a CT machine it is essentially doing a fixed length radon transform. The object is f(x,y), the distance of the detector from the object is *s,* the angle of the scanner from the center of the object is θ, and the value of p(s,θ) is the absorption of the X-Rays at that point [2].

Thus, if a CT scan generates a sinogram, the more important task is to reconstruct the sinogram into a tomographic image. This is done through the inverse of the Radon transform. This is often called the back-projection formula. It is defined as follows:

Equation 2: Inverse Radon Transform

The formula for back-projection is quite simple if the original radon transform data is well-known. The problem lies in the fact that the reconstructed image will be a blurred version of the original image because of small variations in intensity that occur when the Radon transform is done discreetly (as it must be for any real-world uses). The low frequency data can build up during reconstruction, causing blurring. For this reason, most reconstruction algorithms perform what is called filtered back-projection. In filtered back-projection, the sinogram is first filtered before back-projection begins [1]. These filters are often based on the inverse Fourier transform, which is incredibly useful for eliminating extraneous peaks and valleys in the data.

**PROGRAM ALGORITHMS**

The algorithm for the forward Radon transform of a two dimensional image is quite simple. Rather than a scanner moving around the image, the “scanner” is kept stationary at the tip while the image, centered at the origin, is rotated from 0, 180 degrees relative to the x-axis. At each step, which corresponds to a one-degree shift, the pixels that fall in a straight line vertically are added together and these values in a list are placed in a column that corresponds to the step number. In this way, the sinogram is generated. A rudimentary example of how this works is shown in Figure 1.

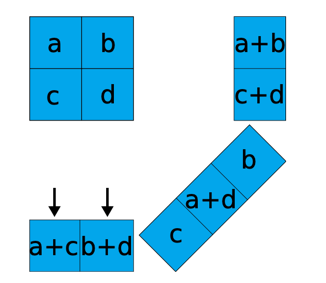
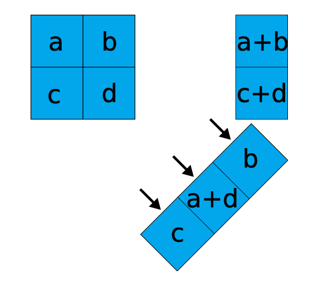
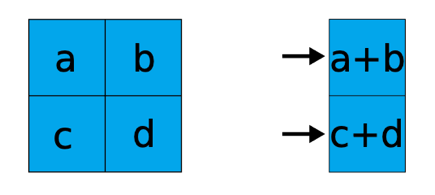


Figure 1: Summation of Pixel Values [2]

The original image is loaded via python’s PIL library and manipulation of it is done using the built-in image functions within numpy. The core of the Radon transform function is as follows:

for step in range(steps):

rotation = floatImage.rotate(-step\*180/steps)

npRotate = np.array(rotation)

radon[:,step] = sum(npRotate)

return radon

Figure 2: Code Snipett for Radon Transform

The inverse Radon transform is much more involved, due to the need to implement the filtered back-projection. First, the sinogram is loaded into the program. It is then padded with the value 0 around its edges to ensure no data is lost in the reconstruction. Then, the Fourier transforms and filters are applied using scipy’s built in functions. After the filters are applied, the back projection can begin. From the sinogram, each individual projection is regenerated and then interpolated and accumulated with the previous projections to form the tomographic reconstruction of the image. The heart of this process is seen in Figure 3:

for i in range(len(theta)):

angle = yprojection \* np.cos(theta[i]) – xprojection \*np.sin(theta[i])

s = np.arange(radon\_filter.shape[0]) – middle

#linear interpolation

projection = np.interp(angle, s, radon\_filter[:,i],left=0, right=0)

reconstructed\_image += projection

Figure 3: Code Snippet for Inverse Radon Transform

The full code of the project can be found in the Appendix.

**RESULTS**

Overall, the program worked remarkably well at both generating the sinograms for images and then reconstructing them using filtered back-projection. However, the algorithms used did not implement any of the new speed improvements that have been developed in recent years. These improvements are immensely complex and are beyond the scope of a short introduction to tomographic image reconstruction. Figures 4 and 5 show the decomposition and reconstruction of two test images. Figure 4 uses the canonical test image, the Shepp-Logan Phantom, and Figure 5 uses an image of a cat.

Graphical user interface, application

Description automatically generated

Figure 4: Deconstruction and Reconstruction of the Shepp-Logan Phantom

A picture containing text, cat, indoor, mammal

Description automatically generated

Figure 5: Deconstruction and Reconstruction of a Cat

**DISCUSSION**

The algorithms implemented programmatically showed great success in reconstructing the images from their sinograms. Since the sinogram data was generated from clean images, the reconstructions were also much cleaner than they would be in reality. However, the program and its results do show the concept of the algorithms involved and the basics of how image reconstruction works.

**CONCLUSION**

Image manipulation algorithms has become the hallmark of the digital age. Such things were not even possible before the advent of computers. Now most modern programming languages have image libraries built-in and it has become even easier to code image manipulation algorithms. This is important, as some of these algorithms can help save lives. New and faster algorithms and better scanning methods are invented every day, but the core math has remained the same. The Radon transform and its inverse will continue to be relevant into the near future and beyond.

**REFERENCES**

[1]: <https://www.aapm.org/meetings/99AM/pdf/2806-57576.pdf>

[2]: <https://www.desy.de/~garutti/LECTURES/BioMedical/Lecture7_ImageReconstruction.pdf>

**APPENDIX**

import numpy as np

from PIL import ImageMath

from scipy.fftpack import fft, ifft, fftfreq

from scipy.interpolate import interp1d

def radon\_transform(image):

"""Implementation of the radon transform"""

#Need a numpy array to represent the image values

npImage = np.array(image)

#Also need a float copy of the image for logistical reasons

floatImage = ImageMath.eval("float(a)", a = image)

steps = image.size[0]

#Empty array to store the newly created sinogram

radon = np.zeros((steps, len(npImage)), dtype='float64')

#For each step, we need to roate the image and sum along the vertical lines, this is the DRT

for step in range(steps):

rotation = floatImage.rotate(-step\*180/steps)

npRotate = np.array(rotation)

radon[:,step] = sum(npRotate)

return radon

def inverse\_radon\_transform(sinogram, limit):

"""Inverse of the radon transform, reconstructs a sinogram"""

if type(sinogram) != "<class 'numpy.ndarray'>":

sinogram = np.array(sinogram)

size = len(sinogram)

theta = np.linspace(0, limit, len(sinogram), endpoint=False) \* (np.pi/180.0)

#First, we need to pad the image

#the maximum projection size is the nearest power of 2 fromt the sinogram size

#the mimimum must be 64, as this is the standard in calculation

max\_projeciton\_size = max(64, int(2\*\*np.ceil(np.log2(2\*len(sinogram)))))

#Pad the image with 0's

#this makes computation easier and ensures we do not lose anything

pad\_width = ((0,max\_projeciton\_size-len(sinogram)), (0,0))

padded\_sinogram = np.pad(sinogram, pad\_width,mode="constant", constant\_values=0)

#Now we need to deploy the filters, these are based on the Fourier Transforms

#First, get the frequencies

filter = fftfreq(max\_projeciton\_size).reshape(-1,1)

#One of the best filters to apply is the Ram-Lak filter

#Apply the fourier transform

ram\_lak = 2\*np.abs(filter)

fourier\_projection = fft(padded\_sinogram,axis=0) \* ram\_lak

#We only want the real parts of the inverse fourier transform

radon\_filter = np.real(ifft(fourier\_projection, axis=0))

radon\_filter = radon\_filter[:sinogram.shape[0],:]

#Prepare a place for the reconstrcted image to go

reconstructed\_image = np.zeros((size,size))

middle = size // 2

#backprojection begins

[X,Y] = np.mgrid[0:size, 0:size]

xprojection = X - int(size) // 2

yprojection = Y - int(size) // 2

for i in range(len(theta)):

angle = yprojection \* np.cos(theta[i]) - xprojection \* np.sin(theta[i])

s = np.arange(radon\_filter.shape[0]) - middle

#linear interpolation

projection = np.interp(angle, s, radon\_filter[:,i],left=0, right=0)

reconstructed\_image += projection

#Eliminate the pixels that fall ouside the reconstruction cirlce

radius = size // 2

circle = (xprojection \*\* 2 + yprojection \*\* 2) <= radius \*\* 2

reconstructed\_image[~circle] = 0

return reconstructed\_image #\* np.pi / (360)