* ***WSG 84 Web Mercator*** is a cylindrical projection system variant of Mercator projection, it is the standard for web mapping system and used by almost all the commercial mapping software, it rose to popularity after Google adopted it in 2005.
* **The Math Magic of Mercator Projection**

R

dy

dx

Q

S

R

dy

dɸ

T

dɸ

Q

S

dx

T

ɸ

ɸ

r

P

O

r

O

P

R

Q

dy

dx

S

T

For sake of simplicity and practicality let us assume dx pixel to be very small and so will be dɸ.

Now we Have:  
 OP OT OS = r  
 OR \* cos(ɸ) = OP = r

We get:  
 dx = OS \* dɸ = r \* dɸ  
 dy = OR \* dɸ = r \* sec(ɸ) \* dɸ  
   
 dy = r \* dɸ \* sec(ɸ) = dx \* sec(ɸ)

Taking dx as 1 pixel => dy = 1 \* sec(ɸ) = sec(ɸ)

dɸ

O

* **The Code Magic of Python NumPy OpenCV**

# import for type hinting and type checking  
from typing import \*  
  
# third party libraries  
import cv2  
import numpy as np  
  
# *todo: optimize*# *todo: implement datum*"""  
for sake of simplicity we will be considering a spherical surface  
:var widthScale: width of image wrt to an unit irrespective of pixel density  
:var heightScale: height of image wrt to an unit irrespective of pixel density  
"""  
  
  
class Image: # custom class for storing image and its properties  
 def \_\_init\_\_(self, array: Union[List, np.ndarray], widthScale: float, heightScale: float):  
 self.image = np.array(array)  
 self.ws = widthScale  
 self.hs = heightScale  
  
 self.px = self.ws / self.image.shape[0] # width per pixel wrt unit  
 self.py = self.hs / self.image.shape[1] # height per pixel wrt unit  
  
 @staticmethod  
 def fromFile(fileName, \*args, \*\*kwargs):  
 return Image(cv2.imread(fileName), \*args, \*\*kwargs)  
  
 def show(self, factor=1):  
 *"""* ***:param*** *factor: resize the image by a factor keeping aspect ratio constant  
 """* cv2.imshow('img', cv2.resize(self.image, (np.array(self.image.shape[:2])[::-1] \* factor).astype(int)))  
 cv2.waitKey(0)  
  
  
# *todo: take variable sized images*class ImageCluster:  
 def \_\_init\_\_(self, imagesArray: List[List[Image]], perWidthScale, perHeightScale, imageShape):  
 self.imArray = np.array(imagesArray)  
 self.ws, self.hs = self.imArray.shape[1] \* perWidthScale, self.imArray.shape[0] \* perHeightScale  
  
 self.avgShape = np.array([0, 0])  
 x = y = None  
 for y, im in enumerate(self.imArray):  
 shapesy = np.array([0, 0])  
 for x, i in enumerate(im):  
 assert [i.ws, i.hs] == [perWidthScale, perHeightScale], "Distance scales mis-match"  
 assert i.image.shape == imageShape, "Image shape mis-match"  
 shapesy += i.image.shape[:2]  
 self.avgShape += shapesy // (x + 1)  
 self.avgShape = self.avgShape // (y + 1)  
  
 def show(self, factor=1):  
 *"""* ***:param*** *factor: resize the image by a factor keeping aspect ratio constant  
 """* npArray = []  
 for y, im in enumerate(self.imArray[::-1]):  
 yArray = []  
 for x, i in enumerate(im[::-1]):  
 yArray.append(i.image)  
 npArray.append(np.concatenate(yArray, axis=1))  
 npArray = np.concatenate(npArray)  
  
 cv2.imshow('imc', cv2.resize(npArray, (factor \* self.avgShape[::-1]).astype(int)))  
 cv2.waitKey(0)

class Projector:  
 def \_\_init\_\_(self, radius):  
 *"""* ***:param*** *radius: radius of the spherical surface(datum) which is being projected from  
 """* self.radius = radius  
  
 def projectToCylinder(self, imageCluster: ImageCluster, yOffsetAngle: float = 0) -> Image:  
 *"""* ***:param*** *imageCluster: instance of the class ImageCluster to be projected* ***:param*** *yOffsetAngle: initial latitude for projection* ***:return****: instance of the class Image after projected to the cylinder  
 """* yOffsetAngle = float(np.radians(yOffsetAngle))  
 theta = np.array(imageCluster.hs / self.radius) # range of the latitude angles for the given imageCluster  
  
 fullImage = []  
 for y, im in enumerate(imageCluster.imArray):  
 newImageX = []  
 for x, i in enumerate(im):  
 newImageX.append(i.image)  
 fullImage.append(np.concatenate(newImageX, axis=1))  
 fullImage = np.concatenate(fullImage) # stitched image of all the part images  
  
 return Image(self.projectImg(fullImage, yOffsetAngle, theta + yOffsetAngle)[::-1].astype(np.uint8), 0, 0)  
  
 @staticmethod  
 def projectImg(img: np.ndarray, thetaStart: float, thetaEnd: float) -> np.ndarray:  
 *"""* ***:param*** *img: numpy array of the image to be project* ***:param*** *thetaStart: latitude initial angle* ***:param*** *thetaEnd: latitude max angle for the image* ***:return****: numpy array of projected image  
 """* thetaStep = (thetaEnd - thetaStart) / img.shape[0] # the latitude range per pixel of the image  
 thetaPixel = np.arange(start=thetaStart, stop=thetaEnd, step=thetaStep) # all the latitude angle for the pixels  
 projectionStretch = 1 / np.cos(thetaPixel) # gives the projection ratio of the pixel on the cylinder  
 pixelWidth = int(max(img.shape[1] \* projectionStretch))  
  
 """  
 algorithm concept:  
 - iterate through all the rows and resize the width with its ratio  
 - add new rows based on the ratio, the fraction of the ratio is carried to the next row wrt its fraction,  
 the carry row is then added with the next row wrt the remaining fraction to be filled  
 """  
 newImg = []  
 carryRow = np.zeros(img.shape[1:]), 0  
 for row, stretch in zip(img[::-1], projectionStretch):  
 fillRow = 1 - carryRow[1]  
 carryStretch = stretch - fillRow  
 newRows = [carryRow[0] + row \* fillRow]  
 if int(carryStretch):  
 newRows.extend(np.tile(row, (int(carryStretch), 1, 1)))  
 newRows = np.array(newRows)  
 carryStretch -= int(carryStretch)  
 carryRow = (row \* carryStretch), carryStretch  
 newRows = cv2.resize(newRows, (int(newRows.shape[1] \* stretch), newRows.shape[0]))  
 extraLen = pixelWidth - newRows.shape[1]  
 # add empty pixels to fill till the max width of the image to make a rect image  
 newRows = np.pad(newRows, ((0, 0), (done := extraLen // 2, extraLen - done), (0, 0)),  
 'constant', constant\_values=0)  
 newImg.extend(newRows)  
  
 return np.array(newImg)

im1 = projection.Image.fromFile('sample\_image.jpg', 4000, 3000)  
imc = projection.ImageCluster([[im1]], im1.ws, im1.hs, im1.image.shape)  
prj = projection.Projector(radius=6400)  
im2 = prj.projectToCylinder(imc, 27)

Sample Image Output and Input

Map

Description automatically generated

Map

Description automatically generated

Sample\_image.jpg

Map

Description automatically generated