Header and Imports

```python
# -*- coding: utf-8 -*-
"""
This module is created for GPS code mesaurement computations
GEO106E - Fundamentals of Programming - Spring Term 2018
Author: MSI
"""
# ------------------------------------------------------------------------------
import numpy as np
# ------------------------------------------------------------------------------
```

Read Rinex Observation File

```python
def readRinex(rinexFile)
	""" This program reads rinex observation file and returns
	to C1 code mesaurements and approximate position.

	Usage:
	AppPos, C1 = readRinex("ista0010.18o")
	"""
	f = open(rinexFile) # open file
	rinexLines = f.readlines() # read lines in rinex file
	# ------------------------------------------------------------------------------
	# Read header lines
	line = 0 # initial line index
	while True:
		if 'APPROX POSITION XYZ' in rinexLines[line]:
			AppPos = rinexLines[line].split() # Split approximate position line
			AppPos = AppPos[0:3] # Remove the text "APPROX POSITION XYZ" from the list
			AppPos = [float(i) for i in AppPos] # Convert string to float
			line +=1 # next line
		elif 'END OF HEADER' in rinexLines[line]:
			line +=1 # next line
			break # we reached to the end of header lines!
		else:
			line +=1 # next line
	del rinexLines[0:line] # Delete all the header lines
	# ------------------------------------------------------------------------------
	# Epoch(time) of observation
	SV = epochLine[7][2:] # ['G05G12G16G18']
	SVList = [SV[i:i+3] for i in range(0,len(SV),3)]
	# ['G05','G12','G16','G18']
	del rinexLines[0] # Delete epoch line
	# ------------------------------------------------------------------------------
	# Take all C1 observations in a list
	C1 = [] # define an empty C1 list
	for line in rinexLines:
		observation = line.split() # split line of observation
		C1.append(float(observation[0])) # while appending to list, convert string to float
	C1 = np.array(C1) # convert C1 list to numpy array
	# ------------------------------------------------------------------------------
	return AppPos, C1 # function outputs: Approximate position (XYZ) and C1 mesaurements
```

Read Orbit File

```
def readOrbit(orbitFile):
        """ This program reads orbit file and returns to the coordinates of positioning satellites  (epheme
        File format:
                1st column - Satellite Vehicle ID (SV)
                2nd column - X coordinates of SVs (X)
                3rd column - Y coordinates of SVs (Y)
                4th column - Z coordinates of SVs (Z)
                5th column - Satellite clock bias (dT)
        Usage:
                orbit = readOrbit('igs19821.sp3')
        """
        orbit = np.genfromtxt(orbitFile, # name of orbitFile
                                        skip_header=5, # skip first 5 lines
                                        names =['SV','X','Y','Z','dT'], # name of columns
                                        dtype=['U10', 'float64', 'float64', 'float64', 'float64'])
        # Convert km to m unit
        orbit['X'] *= 1000
        orbit['Y'] *= 1000
        orbit['Z'] *= 1000
        orbit['dT'] *= 1e-6 # Convert microsecond to second
        return orbit # function output
```

Pseudorange Computation

```
def pseudorange(rinexFile, orbitFile):
        """ This program solves pseudorange equations to find
        the position of a receiver using code measurements.
        Usage:
                finalPosition = pseudorange(rinexFile, orbitFile)
        Output:
                X xoordinate of receiver ---> finalPosition[0]
                Y xoordinate of receiver ---> finalPosition[1]
                Z xoordinate of receiver ---> finalPosition[2]
                Satellite clock error of receiver ---> finalPosition[3]
        """
        AppPos, SVList, C1 = readRinex(rinexFile) # read observation file
        orbit = readOrbit(orbitFile) # read orbit file
        # ----------------------------------------------------------------------------
        # Remove the satellites that are not observed by the receiver
        for SV in orbit['SV']:
                if SV not in SVList:
                        SVIndex = np.where(orbit['SV']==SV)
                        orbit = np.delete(orbit, SVIndex)
        # ----------------------------------------------------------------------------
        # Pseudorange Equations (neglecting some terms for ease of understanding)
        # $R_i(t) = \rho_i(t) + c \times (\delta_{SV} - \delta_{rec})$
        # where t is time of measurement
        #               $R_i(t)$ is pseudorange measurement
        #               $\rho_i(t)$ is geometric distance between satellite-receiver
        #               c is speed of light (299792458.0 m/s)
        #               $\delta_{SV}$ is satellite clock bias
        #               $\delta_{rec}$ is receiver clock bias (which is unknown here)
        c = 299792458.0 # speed of light
        # First, computation of geometric distances between satellite and receiver
        # $\rho_i(t) = \sqrt{(X_{SV} - X_R)^2 + (Y_{SV} - Y_R)^2 + (Z_{SV} - Z_R)^2}$
        distance = np.sqrt((orbit['X']-AppPos[0])**2 +(orbit['Y']-AppPos[1])**2+(orbit['Z']-AppPos[2])**2)
```

```python
        # --------------------------------------------------------------------------
        # Solution of resection based on adjustment computation
        # $X = (A^T PA)^{-1}(A^T PL)$
        # where X is matrix of unknowns,
        #                  A is matrix of coefficients (design matrix),
        #                  P is matrix of weights,
        #                  L is matrix of observations (observtion vector)
        # Design matrix A
        # | - $\frac{X_{SV}^1 - X_R}{\rho_1}$ - $\frac{Y_{SV}^1 - Y_R}{\rho_1}$ - $\frac{Z_{SV}^1 - Z_R}{\rho_1}$ - c |
        # | - $\frac{X_{SV}^2 - X_R}{\rho_2}$ - $\frac{Y_{SV}^2 - Y_R}{\rho_2}$ - $\frac{Z_{SV}^2 - Z_R}{\rho_2}$ - c |
        # |        $\vdots$        $\vdots$        $\vdots$        $\vdots$|
        # | - $\frac{X_{SV}^i - X_R}{\rho_i}$ - $\frac{Y_{SV}^i - Y_R}{\rho_i}$ - $\frac{Z_{SV}^i - Z_R}{\rho_i}$ - c |
        coeffMatrix = np.zeros([len(C1),4]) # len(C1): number of Eq. | 4 unknowns-> size: [len(C1),4]
        coeffMatrix[:,0] = ( AppPos[0] - orbit['X']) / distance # - $\frac{X_{SV}^i - X_R}{\rho_i}$
        coeffMatrix[:,1] = ( AppPos[1] - orbit['Y']) / distance # - $\frac{Y_{SV}^i - Y_R}{\rho_i}$
        coeffMatrix[:,2] = ( AppPos[2] - orbit['Z']) / distance # - $\frac{Z_{SV}^i - Z_R}{\rho_i}$
        coeffMatrix[:,3] = c
        # --------------------------------------------------------------------------
        # L matrix
        LMatrix = np.zeros([len(C1),1])
        LMatrix = C1 - distance + c * orbit['dT']
        # --------------------------------------------------------------------------
        # Adjustment Computation
        # $N^{-1} = (A^T PA)$
        NMatrix = np.linalg.inv(np.dot(np.transpose(coeffMatrix), coeffMatrix)) # inverse of N
        # $n = (A^T PL)$
        nMatrix = np.matmul(np.transpose(coeffMatrix), LMatrix)
        # $X = (A^T PA)^{-1}(A^T PL)$
        XMatrix = np.dot(NMatrix, nMatrix) # Matrix of Unknowns
        # --------------------------------------------------------------------------
        # $X_{Final} = X_{approximate} + x_{unknown}$
        # $Y_{Final} = Y_{approximate} + y_{unknown}$
        # $Z_{Final} = Z_{approximate} + z_{unknown}$
        posFinal = [ AppPos[0] + XMatrix[0], AppPos[1] + XMatrix[1], AppPos[2] + XMatrix[2], XMatrix[3]]
        return posFinal # function output
```