MyAmSatInStem

December 12, 2020

1 AmSat to kindle STEM in India

1.1 Overview

We need a LEO Polar Sun Sync orbit AmSat with FM repeater and few sensors data monitor. Which inturn can be tracked and listened to by students at heart by using either a suitable FM receiver or better still a SDR reciever.

1.2 a NOTE

These are some initial thoughts in general, need to think/check

Below are few simple minded calculations to get a very very rough initial idea of few things. It is more simple geometry and physics based and far removed from actual orbital characteristics. But still …

Also I have nevered looked into satelite orbits and its implications before, on top these are my initial thoughts, that too with simple minded modeling, so take these with a big pinch of salt ;-)

Thus this is a simple stupid look at some aspects of a possible amsat. It uses basic geometry and physics to try and get some rough estimates wrt velocity, orbital period, eclipse time, time to cover equatorial circumference fully, rf path loss, doppler drift …

[1]:
```
import math
import numpy
import matplotlib.pyplot as plt
```

1.3 Earth

[2]:
```
# Gravitational Constant (m^3 / (kg * s^2))
iGravitationalConstant = (6.674 / 10**11)

# Earth Radius (m)
iEarthRadius = 6_400_000

# Earth Mass (kg)
iEarthMass = 5.972 * 10**24

# Earth Circumference
iEarthCircumference = 2 * math.pi * iEarthRadius
```

```
print("EarthCircumference:", iEarthCircumference)
```

```
EarthCircumference: 40212385.96594935
```

1.4 Orbit

The Polar orbit should allow the satellite to cover most parts of the earth.

If a orbit altitude is reached such that the velocity achieved can ensure that N (1 or more) full orbits can be finished in time T, such that T is 24 hours or its multiples, then the satellite will retrace the same path at the same relative time once every T hours i.e T/24 days. NOTE: Not sure if the LEO orbits is sufficiently stable enough over a long period of atleast few weeks, so that it can be used to have a simple tracking wrt the satellite for that period.

If a sun synchronous orbit is achived or maintained, then depending on the angle wrt earth-sun orbital plane,

- if it is 90 degrees, then the satellite could be placed always in day light, so that solar power is always available. However one needs to verify if the temperature of the satellite parts can be maintained well within reasonable operating range, in this case. Is the attitude maintaining rotation good enough or not or conductive heat distribution pipes/paths good enough or ...
- if it is 0 degrees, then the satellite would be always at around roughly mid day or mid night wrt the places on earth, depending on which side of the earth wrt sun, that place is.

The satellite may pass over a given ground location between 0 to 2 times under normal conditions for a polar sun synchronous orbit. Ideally this would be twice if the satellite (and its RF system) covers all parts of the earth fully in a day.

1.5 LEO

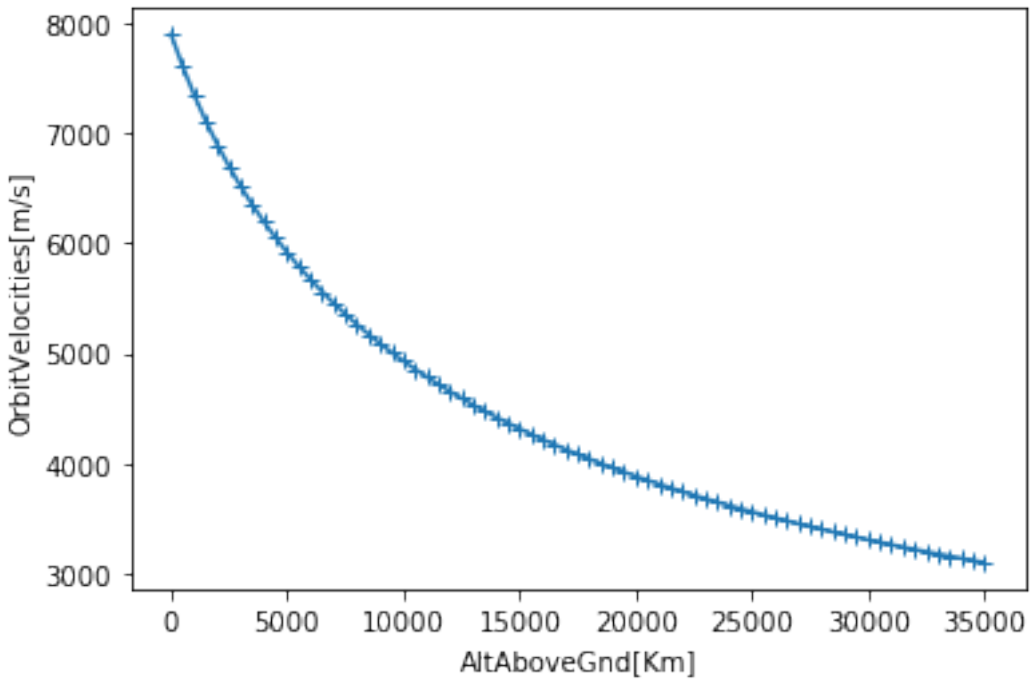
[3]:
```
## LowEarthOrbit

iLeoAltitudeAboveGround = 500_000
iLeoAltitudeFromEarthCenter = iEarthRadius + iLeoAltitudeAboveGround
print("LEO orbit altitude wrt Earth Center:", iLeoAltitudeFromEarthCenter)

# LEO Velocity
#F = (GMm)/(r^2) = ma = m(v^2/r) => GM = r(v^2)
iLeoVelocity = math.sqrt((iGravitationalConstant * iEarthMass)/
 ↪iLeoAltitudeFromEarthCenter)
print("LEO Velocity:", iLeoVelocity)

# Circumference of the orbit
iLeoCircumference = 2 * math.pi * iLeoAltitudeFromEarthCenter
print("LeoCircumference:", iLeoCircumference)

# Time taken for a single orbit
iLeoOrbitTimeTaken = iLeoCircumference/iLeoVelocity
print("LeoOrbitTimeTaken(mins):", iLeoOrbitTimeTaken/60)
```

```
# Num of Siteings in a day
iDayInSecs = 24*60*60
iNumOfSiteingsInADay = iDayInSecs/iLeoOrbitTimeTaken
print("NumOfSiteings in a Day:", iNumOfSiteingsInADay)
iDayInSecs/(iDayInSecs%iLeoOrbitTimeTaken)
#0.146*103
```

```
LEO orbit altitude wrt Earth Center: 6900000
LEO Velocity: 7600.260102337841
LeoCircumference: 43353978.61953914
LeoOrbitTimeTaken(mins): 95.0712607984812
NumOfSiteings in a Day: 15.146533115326104
```

[3]:
```
103.36593937566968
```

[4]:
```
## Orbital Velocities

iAltitudesAboveGround = numpy.linspace(0,10_000_000, 20)
iAltitudesAboveGround = numpy.arange(0,35_500_000, 500_000)
def orbitvelocity_from_altitude(altitudeFromCenter, theMainMass=iEarthMass):
    return numpy.sqrt((iGravitationalConstant*theMainMass)/altitudeFromCenter)

def orbittime_from_altitude(altitudeFromCenter, theMainMass=iEarthMass):
    orbVel = orbitvelocity_from_altitude(altitudeFromCenter, theMainMass)
    orbCirumference = 2 * math.pi * altitudeFromCenter
    return orbCirumference/orbVel

iOrbitVelocities =␣
 ↪orbitvelocity_from_altitude(iAltitudesAboveGround+iEarthRadius)
iOrbitTimes = orbittime_from_altitude(iAltitudesAboveGround+iEarthRadius)
plt.plot(iAltitudesAboveGround/1000, iOrbitVelocities, "+-")
#plt.plot(iAltitudesAboveGround/1000, iOrbitTimes, "+-r")
plt.xlabel("AltAboveGnd[Km]")
plt.ylabel("OrbitVelocities[m/s]")
plt.show()
```



1.6 Earth coverage

[5]:
```
# Earth Coverage and Altitude

# Sin(Angle) = OppositeSide/Hypotinus
# Cos(Angle) = AdjacentSide/Hypotinus
# Tan(Angle) = Opposite/Adjacent
# Opposite = Tan(Angle) * Adjacent
# AdjacentSide = Altitude from surface
# Angle = FieldOfView
iSatAltitude = numpy.array([200_000, 500_000, 1_000_000, 36_000_000])
iFieldOfView = 106 # Adjusted to sync NumOfSiteings/day with␣
 ↪EarthEquatorialCircumferenceCoverage from 500KM
iFieldOfView = 60
iEarthCoverageRadius = numpy.tan(((iFieldOfView/2)/360)*math.pi*2)*iSatAltitude
print("EarthCoverageRadius[@{}]:{}".format(iFieldOfView, iEarthCoverageRadius))
math.atan(iEarthRadius/(iEarthRadius+36_000_000)) * (360/(math.pi*2))
math.atan(iEarthRadius/(iEarthRadius+500_000)) * (360/(math.pi*2))

# Minimum number of orbits to cover the earths equatorial circumference
iEarthCircumferenceAtEquatorCoveredPerOrbit = iEarthCoverageRadius*2*2
print("EarthCircumferenceAtEquator CoveredPerOrbit:",␣
 ↪iEarthCircumferenceAtEquatorCoveredPerOrbit)
```

```
iMinNumOfOrbits4Equator=iEarthCircumference/
 ↪iEarthCircumferenceAtEquatorCoveredPerOrbit
print("Minimum number of orbits required to cover earths equatorial␣
 ↪circumference fully:", iMinNumOfOrbits4Equator)

# Movement of earth in a given time
iEarthMovementPerSecAtEquatorDueToRotation = iEarthCircumference/iDayInSecs
print("EarthMovementPerSecAtEquatorDueToRotation:",␣
 ↪iEarthMovementPerSecAtEquatorDueToRotation)
iEarthMovementPerOrbitAtEquatorDueToRotation = iLeoOrbitTimeTaken *␣
 ↪iEarthMovementPerSecAtEquatorDueToRotation
print("EarthMovementPerOrbitAtEquatorDueToRotation:",␣
 ↪iEarthMovementPerOrbitAtEquatorDueToRotation)

# Percent of required area covered per orbit
(iEarthCircumferenceAtEquatorCoveredPerOrbit/
 ↪iEarthMovementPerOrbitAtEquatorDueToRotation)*100
```

```
EarthCoverageRadius[@60]:[  115470.05383793   288675.13459481   577350.26918963
20784609.69082653]
EarthCircumferenceAtEquator CoveredPerOrbit: [  461880.2153517
1154700.53837925  2309401.0767585  83138438.76330611]
Minimum number of orbits required to cover earths equatorial circumference
fully: [87.06236948 34.82494779 17.4124739   0.48367983]
EarthMovementPerSecAtEquatorDueToRotation: 465.4211338651545
EarthMovementPerOrbitAtEquatorDueToRotation: 2654890.439929136
```

[5]:
```
array([  17.39733619,   43.49334048,   86.98668096, 3131.52051448])
```

1.7 Return to Same point (roughly)

1.7.1 Earth Equatorial Circumference covered across orbits

[6]:
```
# Earth Equatorial Circumference covered across orbits


bCoverageGraphics = False
bDumpVistedEquatorPos = False


def circular_pos(inPos):
    inPos = (inPos % 360)
    if inPos < 0:
        inPos = 360 + inPos
    return inPos
```

```
def test_circular_pos():
    for i in range(-520,520):
        print(i, circular_pos(i))


def dprint(msg, bDebug=False):
    if bDebug:
        print(msg)


iEarthCircumferencePerDegree = iEarthCircumference/360
print("EarthCircumferencePerDegree:", iEarthCircumferencePerDegree)
iEarthMovementPerOrbitAtEquatorInDegrees =␣
 ↪iEarthMovementPerOrbitAtEquatorDueToRotation/iEarthCircumferencePerDegree
print("EarthMovementPerOrbitAtEquatorInDegrees[%d]:
 ↪%g"%(iLeoAltitudeAboveGround, iEarthMovementPerOrbitAtEquatorInDegrees))


'''
def in_coverageradius(inPos, coverageRadiusInDegs):
    startPos = sane_pos(inPos-coverageRadiusInDegs)
    endPos = sane_pos(inPos+coverageRadiusInDegs)
    ...
'''
def circular_diff(pos1, pos2):
    '''
    pos1 and pos2 need to be +ve and within circular range
    '''
    delta = pos1 - pos2
    if delta > 180:
        delta = delta - 360
    if delta < -180:
        delta = 360 + delta
    return delta


def circular_absdiff(pos1, pos2):
    '''
    pos1 and pos2 need to be +ve and within circular range
    '''
    delta = pos1 - pos2
    if abs(delta) > 180:
        delta = 360-abs(delta)
    return abs(delta)


def test_circular_diffs():
```

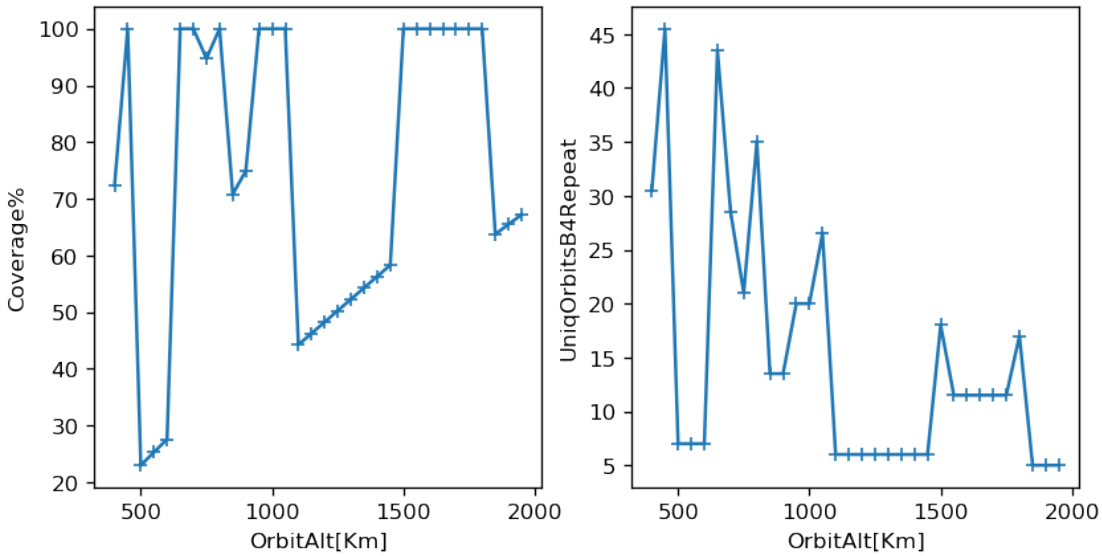
```
    t1=circular_absdiff(270,89)
    t2=circular_absdiff(89,270)
    t3=circular_absdiff(359,0)
    t4=circular_absdiff(0,359)
    print(t1,t2,t3,t4)


def earthcoverage_given_orbittime(orbitTime, coverageRadiusInDegs=0):
    iEarthMovementPerOrbitAtEquator = orbitTime *␣
 ↪iEarthMovementPerSecAtEquatorDueToRotation
    iEarthMovementPerOrbitAtEquatorInDegrees = iEarthMovementPerOrbitAtEquator /
 ↪ iEarthCircumferencePerDegree
    iMaxNumOrbits = 4096
    iStartPos = 0
    iCurPos = iStartPos
    iVisited = set()
    iVisitedL = []
    for i in range(iMaxNumOrbits*2):
        #print(iCurPos)
        iVisited.add(round(iCurPos))
        iVisitedL.append(round(iCurPos))
        # Move to opposite during orbit
        iCurPos = iCurPos + 180 - (iEarthMovementPerOrbitAtEquatorInDegrees/2)
        iCurPos = circular_pos(iCurPos)
        # Doesnt bother if the point being revisited is on sun facing or␣
 ↪eclipsed side
        if circular_absdiff(iStartPos, iCurPos) <= coverageRadiusInDegs:
            iVisited.add(round(iCurPos))
            iVisitedL.append(round(iCurPos))
            dprint("Reached back to starting pos on %d th orbit"%(i/2))
            break
    coverageMultiplier = 2*coverageRadiusInDegs
    if coverageMultiplier == 0:
        coverageMultiplier = 1
    iPercentageCovered = (len(iVisited)/360)*coverageMultiplier*100
    dprint(iVisited, False)
    dprint(iVisitedL, False)
    dprint("%d[/%d] unique orbits, covered %d %% of earth with␣
 ↪FOVCvrageRadiusDegs of %g"%(i/2, iMaxNumOrbits, iPercentageCovered,␣
 ↪coverageRadiusInDegs), False)
    return iPercentageCovered, i/2, iVisitedL


lAlts = numpy.array([])
lCvrd = numpy.array([])
lUniqOrbits = numpy.array([])
iFieldOfView=60
```

```
for alt in range(400_000, 2_000_000, 50_000):
    lAlts = numpy.append(lAlts, alt) # easy to infer
    orbTime = orbittime_from_altitude(alt+iEarthRadius)
    iFOVCoverageRadius = numpy.tan(((iFieldOfView/2)/360)*math.pi*2)*alt
    iFOVCoverageRadiusInDegs = iFOVCoverageRadius / iEarthCircumferencePerDegree
    cvrageP, uniqOrbits, lVisited = earthcoverage_given_orbittime(orbTime,␣
 ↪iFOVCoverageRadiusInDegs)
    lCvrd = numpy.append(lCvrd, cvrageP)
    lUniqOrbits = numpy.append(lUniqOrbits, uniqOrbits)
    print("Orbit: alt [%d] time [%d] FOVCvrgRadiusDegs [%g] cvrage%% [%g] with␣
 ↪[%d] uniqOrbits"%(alt, orbTime, iFOVCoverageRadiusInDegs, cvrageP,␣
 ↪uniqOrbits))
    if bDumpVistedEquatorPos:
        print("DBUG:", lVisited)
    if bCoverageGraphics:
        y = numpy.zeros(len(lVisited))
        plt.plot(lVisited, y, "+")
        plt.show()

savedFigSize = plt.rcParams['figure.figsize']
savedFigDpi = plt.rcParams['figure.dpi']
plt.rcParams['figure.figsize'] = [8,4]
plt.rcParams['figure.dpi'] = 120
#plt.figure(figsize=(8,4), dpi=100)
plt.subplots(1,2)
plt.subplot(121)
tCvrd = numpy.clip(lCvrd, 0, 100)
plt.plot(lAlts/1000, tCvrd, "+-")
plt.ylabel("Coverage%")
plt.xlabel("OrbitAlt[Km]")

plt.subplot(122)
plt.plot(lAlts/1000, lUniqOrbits, "+-")
plt.ylabel("UniqOrbitsB4Repeat")
plt.xlabel("OrbitAlt[Km]")
plt.show()
plt.rcParams['figure.figsize'] = [4,3]
plt.rcParams['figure.dpi'] = 100

bin(2**512-1)
bin(1 << 6)
```

```
EarthCircumferencePerDegree: 111701.07212763708
EarthMovementPerOrbitAtEquatorInDegrees[500000]:23.7678
Orbit: alt [400000] time [5580] FOVCvrgRadiusDegs [2.06748] cvrage% [72.3619]
with [30] uniqOrbits
Orbit: alt [450000] time [5642] FOVCvrgRadiusDegs [2.32592] cvrage% [120.172]
```

```
with [45] uniqOrbits
Orbit: alt [500000] time [5704] FOVCvrgRadiusDegs [2.58435] cvrage% [22.972]
with [7] uniqOrbits
Orbit: alt [550000] time [5766] FOVCvrgRadiusDegs [2.84279] cvrage% [25.2692]
with [7] uniqOrbits
Orbit: alt [600000] time [5828] FOVCvrgRadiusDegs [3.10123] cvrage% [27.5664]
with [7] uniqOrbits
Orbit: alt [650000] time [5891] FOVCvrgRadiusDegs [3.35966] cvrage% [166.117]
with [43] uniqOrbits
Orbit: alt [700000] time [5954] FOVCvrgRadiusDegs [3.6181] cvrage% [118.593]
with [28] uniqOrbits
Orbit: alt [750000] time [6017] FOVCvrgRadiusDegs [3.87653] cvrage% [94.7597]
with [21] uniqOrbits
Orbit: alt [800000] time [6080] FOVCvrgRadiusDegs [4.13497] cvrage% [165.399]
with [35] uniqOrbits
Orbit: alt [850000] time [6143] FOVCvrgRadiusDegs [4.3934] cvrage% [70.7826]
with [13] uniqOrbits
Orbit: alt [900000] time [6207] FOVCvrgRadiusDegs [4.65184] cvrage% [74.9463]
with [13] uniqOrbits
Orbit: alt [950000] time [6271] FOVCvrgRadiusDegs [4.91027] cvrage% [114.573]
with [20] uniqOrbits
Orbit: alt [1000000] time [6335] FOVCvrgRadiusDegs [5.16871] cvrage% [120.603]
with [20] uniqOrbits
Orbit: alt [1050000] time [6399] FOVCvrgRadiusDegs [5.42714] cvrage% [162.814]
with [26] uniqOrbits
Orbit: alt [1100000] time [6464] FOVCvrgRadiusDegs [5.68558] cvrage% [44.2212]
with [6] uniqOrbits
Orbit: alt [1150000] time [6529] FOVCvrgRadiusDegs [5.94401] cvrage% [46.2312]
with [6] uniqOrbits
Orbit: alt [1200000] time [6593] FOVCvrgRadiusDegs [6.20245] cvrage% [48.2413]
with [6] uniqOrbits
Orbit: alt [1250000] time [6659] FOVCvrgRadiusDegs [6.46089] cvrage% [50.2513]
with [6] uniqOrbits
Orbit: alt [1300000] time [6724] FOVCvrgRadiusDegs [6.71932] cvrage% [52.2614]
with [6] uniqOrbits
Orbit: alt [1350000] time [6790] FOVCvrgRadiusDegs [6.97776] cvrage% [54.2714]
with [6] uniqOrbits
Orbit: alt [1400000] time [6855] FOVCvrgRadiusDegs [7.23619] cvrage% [56.2815]
with [6] uniqOrbits
Orbit: alt [1450000] time [6921] FOVCvrgRadiusDegs [7.49463] cvrage% [58.2915]
with [6] uniqOrbits
Orbit: alt [1500000] time [6988] FOVCvrgRadiusDegs [7.75306] cvrage% [163.676]
with [18] uniqOrbits
Orbit: alt [1550000] time [7054] FOVCvrgRadiusDegs [8.0115] cvrage% [111.271]
with [11] uniqOrbits
Orbit: alt [1600000] time [7121] FOVCvrgRadiusDegs [8.26993] cvrage% [114.86]
with [11] uniqOrbits
Orbit: alt [1650000] time [7188] FOVCvrgRadiusDegs [8.52837] cvrage% [118.45]
```

```
with [11] uniqOrbits
Orbit: alt [1700000] time [7255] FOVCvrgRadiusDegs [8.7868] cvrage% [122.039]
with [11] uniqOrbits
Orbit: alt [1750000] time [7322] FOVCvrgRadiusDegs [9.04524] cvrage% [125.628]
with [11] uniqOrbits
Orbit: alt [1800000] time [7390] FOVCvrgRadiusDegs [9.30368] cvrage% [186.074]
with [17] uniqOrbits
Orbit: alt [1850000] time [7457] FOVCvrgRadiusDegs [9.56211] cvrage% [63.7474]
with [5] uniqOrbits
Orbit: alt [1900000] time [7525] FOVCvrgRadiusDegs [9.82055] cvrage% [65.4703]
with [5] uniqOrbits
Orbit: alt [1950000] time [7593] FOVCvrgRadiusDegs [10.079] cvrage% [67.1932]
with [5] uniqOrbits
```



[6]:
```
'0b1000000'
```

1.7.2 Find Orbit Radius given a Orbit Time

iOrbitTime = iOrbitCircumference/iOrbitVelocity

iOrbitCircumference = iOrbitTime * iOrbitVelocity

2 * Pi * iOrbitRadius = iOrbitTime * sqrt((iGravitationalConstant * iEarthMass)/iOrbitRadius)

4 * Pi^2 * iOrbitRadius^2 = iOrbitTime^2 * (iGravitationalConstant * iEarthMass) / iOrbitRadius

iOrbitRadius^3 = (iOrbitTime^2 * iGravitationalConst * iEarthMass) / (4 * Pi^2)

[7]:
```
def orbitradius_giventime(orbitTime, theMainMass=iEarthMass,␣
↪gravitationalConstant=iGravitationalConstant):
```

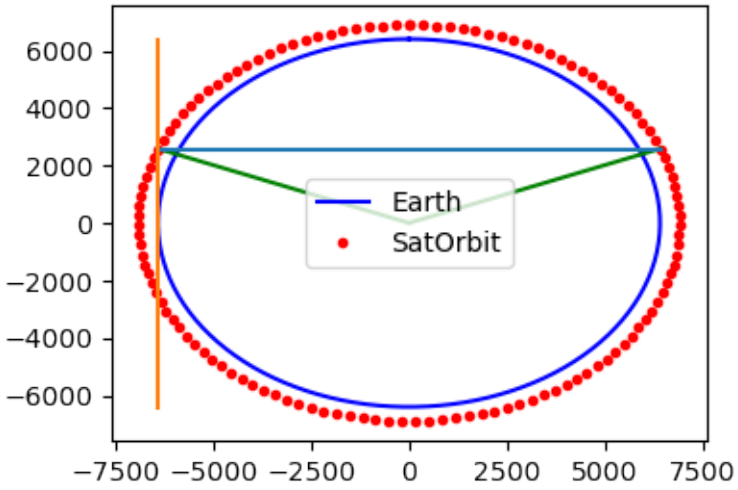
```
    return numpy.cbrt((((orbitTime)**2)*gravitationalConstant*theMainMass)/
 ↪(4*(numpy.pi**2)))

orbitradius_giventime(iLeoOrbitTimeTaken)-iEarthRadius
orbitradius_giventime(2*60*60)-iEarthRadius
#orbitradius_giventime(1.41545*60*60)-iEarthRadius
#1.41545*60
```

[7]: 1658800.7683976134

1.8 Eclipse due to earth

[8]:
```
## Show Earth and Satellite Orbit
rads=numpy.linspace(0,numpy.pi*2,128)
xE = numpy.sin(rads)*iEarthRadius/1000
yE = numpy.cos(rads)*iEarthRadius/1000
plt.plot(xE,yE,"b", label="Earth")
xS = numpy.sin(rads)*iLeoAltitudeFromEarthCenter/1000
yS = numpy.cos(rads)*iLeoAltitudeFromEarthCenter/1000
plt.plot(xS,yS,"r.", label="SatOrbit")
plt.plot([0,xS[24]],[0,yS[24]],"g")
plt.plot([0,xS[-1-24]],[0,yS[-1-24]],"g")
plt.plot([xS[24],xS[-1-24]], [yS[24],yS[-1-24]])
#plt.plot([xS[-1-24],xS[-1-39]], [yS[-1-24],yS[-1-39]])
plt.plot([-iEarthRadius/1000, -iEarthRadius/1000],[-iEarthRadius/
 ↪1000,iEarthRadius/1000])
plt.legend()
plt.show()

'''

## EvenMoreCrudeMath: rough amount of time for which satellite will be eclipsed␣
 ↪by earth in a orbit
iHalfOfEarthToLeoCircumference = (0.5*iEarthCircumference)/iLeoCircumference
iMaxEclipseTime = iHalfOfEarthToLeoCircumference * iLeoOrbitTimeTaken
print("Satellite could be Eclipsed by Earth for a max of {} mins per Orbit".
 ↪format(iMaxEclipseTime/60))
# Actual Eclipse time will be smaller than this, as light bends over (among␣
 ↪others...)

'''

## Blind, based on simple direct EarthDia
iMinEclipseTime = (((iEarthRadius*2)/iLeoCircumference)*iLeoOrbitTimeTaken)
#print("EarthDia:", iEarthRadius*2)
#print("LeoCircumference:", iLeoCircumference)
```

```
print("Satellite could be Eclipsed by Earth for a minimum of around {} mins per␣
 ↪Orbit".format(iMinEclipseTime/60))

## Bit better based on circle segment
halfAngle = math.asin(iEarthRadius/iLeoAltitudeFromEarthCenter)
#iEclipsedCircumference = (2*halfAngle) * iLeoAltitudeFromEarthCenter
iEclipsedCircumference = ((2*halfAngle)/(2*numpy.pi)) * iLeoCircumference
iEclipseTime = ((iEclipsedCircumference/iLeoCircumference)*iLeoOrbitTimeTaken)
print("Satellite could be Eclipsed by Earth for around {} mins per Orbit".
 ↪format(iEclipseTime/60))
```




```
Satellite could be Eclipsed by Earth for a minimum of around 28.06921479801882
mins per Orbit
Satellite could be Eclipsed by Earth for around 35.94430921257815 mins per Orbit
```

1.9 Util functions

[9]:
```
# dB wrt milliwatt normally
def to_dBm(inWatts):
    return 10 * math.log10(inWatts/0.001)

# dB wrt isotropic antenna
def to_dBi(inPower):
    return 10 * math.log10(inPower/1)

iLightSpeed = 300_000_000 # meters / sec

def wavelen2freq(waveLen):
    return iLightSpeed/waveLen
```

```
def freq2wavelen(freq):
    return iLightSpeed/freq
```

1.10 RF Comm

1.10.1 RF Frequencies

AmSats normally use the following bands

- In V band = 145.8 - 146 MHz
- In U band = 435 - 438 MHz

The V band will be more crowded compared to U band.

At same time V band has

- less doppler effect (and so the related freq drift)
- lesser rf path losses

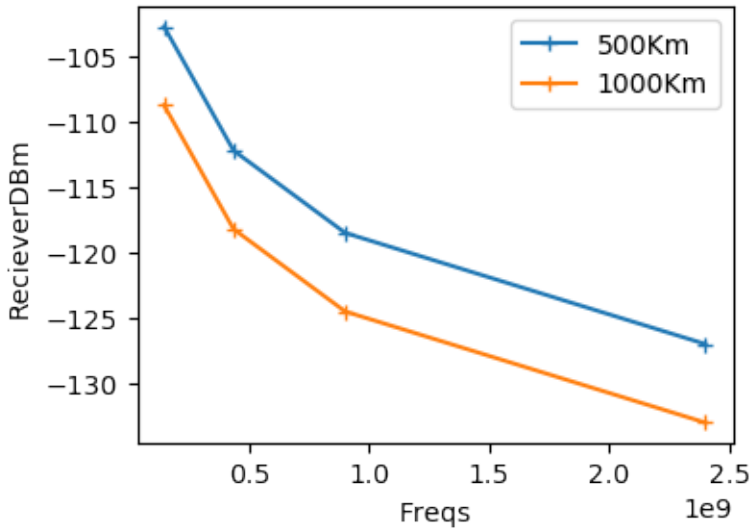
compared to the higher frequency U band.

1.10.2 RF Losses

[10]:
```
# Using Friis transmission formula
# Pr = Pt * Dt * Dr * (Lambda/4*Pi*D)^2
# Pr_dBm = Pt_dBm + Dt_dBi + Dr_dBi + 10*log10((Lambda/4*Pi*D)^2)
# Pr_dBm = Pt_dBm + Dt_dBi + Dr_dBi + 20*log10(Lambda/4*Pi*D)
# Pr_dBm = Pt_dBm + Dt_dBi + Dr_dBi + 20*log10(Lambda) - 20*log10(D) -␣
 ↪20*log10(1/4*Pi)
#
powerTransmitter = 0.5 # Watts; this is what is normal for LEO AmSats
powerTransmitterDBm = to_dBm(powerTransmitter)
print("TransmitPower(dBm):", powerTransmitterDBm)
directivityTransmitter = 1
directivityTransmitterDBi = to_dBi(directivityTransmitter)
directivityReciever = 1 # Change based on antenne system used
directivityRecieverDBi = to_dBi(directivityReciever)
frequencies = numpy.array([145_800_000, 435_000_000, 900_000_000, 2_400_000_000␣
 ↪])
print("Frequencies:", frequencies)
distances = numpy.array([[500_000],[1_000_000]])


# This doesnt consider any losses through_the/due_to medium of propagation, nor␣
 ↪multipath, nor ...
# However as the path loss is the major factor given the distances involved,␣
 ↪its used to get a rough feel
def freespace_pathloss(Pt_dBm, Dt_dBi, Dr_dBi, frequencies, distances):
    Pr_dBm = Pt_dBm + Dt_dBi + Dr_dBi + 20*numpy.log10(iLightSpeed/(4*numpy.
 ↪pi*distances*frequencies))
```

```
    return Pr_dBm


powerRecieverDBm = freespace_pathloss(powerTransmitterDBm,␣
 ↪directivityTransmitterDBi, directivityRecieverDBi,
                                      frequencies, distances)
print("PowerAtReciever_Rough(dBm):\n",powerRecieverDBm)
plt.plot(frequencies, numpy.atleast_2d(powerRecieverDBm).T,"+-")
plt.legend(["500Km", "1000Km"])
plt.xlabel("Freqs")
plt.ylabel("RecieverDBm")
plt.show()
```

```
TransmitPower(dBm): 26.989700043360187
Frequencies: [ 145800000  435000000  900000000 2400000000]
PowerAtReciever_Rough(dBm):
 [[-102.70662271 -112.20125737 -118.51632242 -127.03569706]
 [-108.72722262 -118.22185728 -124.53692233 -133.05629698]]
```

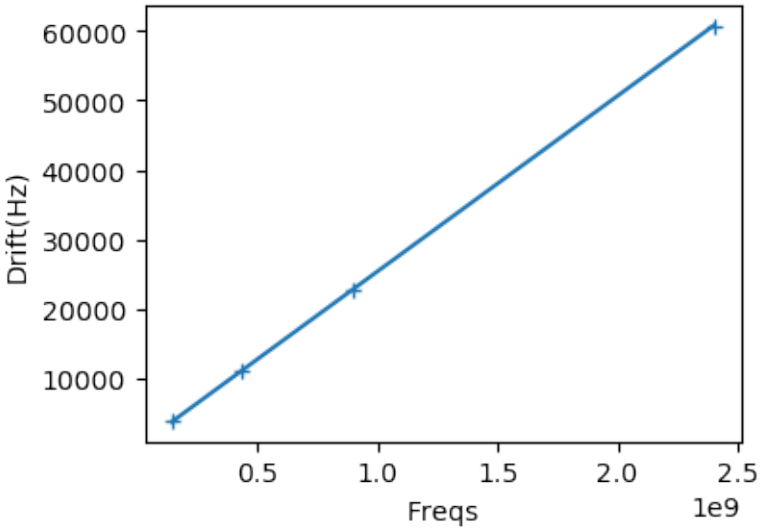


1.10.3 Doppler effect

[11]:
```
commFreqs = numpy.array([145_800_000, 436_000_000])
dopShifts = commFreqs*(iLeoVelocity/iLightSpeed)
print("DopplerShifts[in Hz] for {} is {}".format(commFreqs, dopShifts))
dopShifts = frequencies*(iLeoVelocity/iLightSpeed)
plt.plot(frequencies, dopShifts, "+-")
plt.xlabel("Freqs")
plt.ylabel("Drift(Hz)")
```

```
plt.show()
```

```
DopplerShifts[in Hz] for [145800000 436000000] is [ 3693.72640974
11045.71134873]
```



1.10.4 Data comm

In turn use a simple mechanism to transmit the sensor data like

a) Simple Analog frequency modulation: Different frequencies for different values

b) Analog frequency modulated tones for different decimal digits from 0 to 9, maybe each 200 Hz appart.

c) Binary Digital Keying: Presence or absence of Frequency or Toggle btw 2 different sets of frequencies indicate between 0 and 1.

1.11 Misc

FootNotes Will reside somewhere at https://github.com/hanishkvc