Coordinate Systems

October 2, 2020

fnc.py - Coordinate Systems block

The goal of this file is explaining each step that was taken towards implementing all the transformation matrices.

[10]:
```
#%% Script information
# Name: fnc.py
# Authors: Trajectory Team (Matias Pellegrini, Pablo Lobo)
# Owner: LIA Aerospace
#
#%% Script description
#
# The aim of this module is defining functions to be used in the simulation.
#
#%% Packages
import numpy as np
import c as c

#%% Coordinate Systems

# This block implements the different functions required to transform
# the different tensors from one coordinate system to another.
# Source: Zipfel.
```

Function: Tge

The aim of this function is to calculate the transformation matrix between the geographical and Earth coordinate systems.

[11]:
```
def Tge(long,lat):
    # Zipfel (3.13)
    # === INPUTS ===
    # long [rad]              longitude
    # lat [rad]               Latitude
    # === OUTPUTS ===
    # tge [3x3 mat]           T_GE
    # Create the basic values
    slon = np.sin(long)
    clon = np.cos(long)
```

```
    slat = np.sin(lat)
    clat = np.cos(lat)
    # Create 9 positions
    ind11 = -slat*clon;
    ind12 = -slat*slon
    ind13 = clat
    ind21 = -slon
    ind22 = clat
    ind23 = 0
    ind31 = -clat*clon
    ind32 = -clat*slon
    ind33 = -slon
    # Create the matrix itself
    tge = np.array([[ind11, ind12, ind13],[ind21, ind22, ind23],[ind31, ind32,␣
↪ind33]])
    return tge
```

Function: Tei

The aim of this function is to calculate the transformation matrix between the Earth and inertial coordinate systems.

[12]:
```
def Tei(hangle):
    # Zipfel (3.12)
    # === INPUTS ===
    # hangle [rad]            Hour angle
    # === OUTPUTS ===
    # tei [3x3 mat]           T_EI
    # Create the basic values
    sha = np.sin(hangle)
    cha = np.cos(hangle)
    # Create 9 positions
    ind11 = cha
    ind12 = sha
    ind13 = 0
    ind21 = -sha
    ind22 = cha
    ind23 = 0
    ind31 = 0
    ind32 = 0
    ind33 = 1
    # Create the matrix itself
    tei = np.array([[ind11, ind12, ind13],[ind21, ind22, ind23],[ind31, ind32,␣
↪ind33]])
    return tei
```

Function: Tmv

The aim of this function is to calculate the transformation matrix between the load factor and velocity coordinate systems.

[13]:
```
def Tmv(bang):
    # Zipfel (8.22)
    # === INPUTS ===
    # bang [rad]              Bank Angle
    # === OUTPUTS ===
    # tmv [3x3 mat]           T_MV
    # Create the basic values
    sba = np.sin(bang)
    cba = np.cos(bang)
    # Create 9 positions
    ind11 = 1
    ind12 = 0
    ind13 = 0
    ind21 = 0
    ind22 = -cba
    ind23 = sba
    ind31 = 0
    ind32 = -sba
    ind33 = cba
    # Create the matrix itself
    tmv = np.array([[ind11, ind12, ind13],[ind21, ind22, ind23],[ind31, ind32,␣
 ↪ind33]])
    return tmv
```

Function: Tvg

The aim of this function is to calculate the transformation matrix between the flight path and geographic coordinate systems.

[14]:
```
def Tvg(gamma,chi):
    # Zipfel (3.25)
    # === INPUTS ===
    # gamma [rad]             Heading Angle
    # chi [rad]               Flight Path Angle
    # === OUTPUTS ===
    # tvg [3x3 mat]           T_VG
    # Create the basic values
    schi = np.sin(chi)
    cchi = np.cos(chi)
    sgamma = np.sin(gamma)
    cgamma = np.cos(gamma)
    # Create 9 positions
    ind11 = cchi*cgamma
    ind12 = cgamma*schi
    ind13 = -sgamma
```

```
    ind21 = -schi
    ind22 = -cchi
    ind23 = 0
    ind31 = sgamma*cchi
    ind32 = sgamma*schi
    ind33 = cchi
    # Create the matrix itself
    tvg = np.array([[ind11, ind12, ind13],[ind21, ind22, ind23],[ind31, ind32,␣
↪ind33]])
    return tvg
```

Function: JD

The aim of this function is to calculate the Julian Date. Source: Vallado, Algorithm #14.

Unless specified, JD usually implies a time based on UT1. Equation valid from years 1900 to 2100.

[15]:
```
def JD(yr,mo,d,h,min,s):
    # === INPUTS ===
    # yr [adim]                 Year of interest
    # mo [adim]                 Month of interest (1 to 12)
    # d [adim]                  Day of interest (1 to 31)
    # h [adim]                  Hour of interest (0 to 23)
    # min [adim]                Min of interest (0 to 59)
    # s [adim]                  Seconds of interest (0 to 59)
    # === OUTPUTS ===
    # jdate [adim]              Julian Date
    # Function
    jdate = 367*yr - int((7*(yr+int((mo+9)/12)))/4) + int((275*mo)/9) + d +␣
↪1721013.5 + ((((s/60)+min)/60) + h)/24
    return jdate
```

Function: jd2tjd

The aim of this function is to calculate the number of Julian centuries elapsed from the epoch J2000.0. Source: Vallado, Eq. (3.42)

Equation valid for epoch J2000.0, see p.183 for other epochs.

[16]:
```
def jd2tjd(jdate):
    # === INPUTS ===
    # jdate [julian date]      Julian date, as provided by function JD
    # === OUTPUTS ===
    # Function
    # tjdate [centuries]       Julian centuries elapsed since J2000.0 epoch
    tjdate = (jdate - 2451545)/36525
    return tjdate
```

Function: tjd2gmst

The aim of this function is to calculate the Greenwich Mean Sidereal Time given the number of Julian centuries elapsed from the epoch J2000.0. Source: Vallado, Eq. (3.47)

[17]:
```python
def tjd2gmst(tjdate):
    # === INPUTS ===
    # tjdate [centuries]      Julian centuries elapsed since J2000.0 epoch
    # === OUTPUTS ===
    # gmst_s [s]              GMST in seconds
    # gmst_d [°]              GMST in degrees
    # Function
    gmst_s = 67310.54841 + (876600*3600 + 8640184.812866)*tjdate + 0.
 ↪093104*tjdate**2 - 6.2*10**-6* (tjdate**3)
    # Reduce this quantity to a result within the range of 86400s
    secs_day = 86400
    gmst_aux = gmst_s % secs_day
    # Convert to degrees
    gmst_d = gmst_aux / 240
    # Convert to an angle in the 0-360 range
    if gmst_d < 0:
        gmst_d += 360
    return gmst_s, gmst_d
```

Function: date_now

The aim of this function is to return the date at time of invoking it.

[18]:
```python
def date_now():
    # === INPUTS ===
    #
    # === OUTPUTS ===
    # date_out [datetime]        Date at time of function invoking
    # Function
    from datetime import datetime
    date = datetime.now()
    return date
```

Function: date_parts

The aim of this function is to return the different values stored in the input datetime value.

[19]:
```python
def date_parts(date_in):
    # === INPUTS ===
    # date_in [datetime]       Input date
    # === OUTPUTS ===
    # yr [int]                 Year on input date
    # m [int]                  Month on input date
    # d [int]                  Day on input date
    # h [int]                  Hour on input date
    # m [int]                  Minute on input date
```

```python
    # s [int]                  Second on input date
    yr = date_in.year
    mo = date_in.month
    d = date_in.day
    h = date_in.hour
    m = date_in.minute
    s = date_in.second
    return yr, mo, d, h, m, s
```

Function: add_timestep

The aim of this function is to add a given timestep to a given date.

[20]:
```python
def add_timestep(date,timestep):
    # === INPUTS ===
    # date [datetime]       Initial time
    # timestep [timedelta]  Timestep to be added
    # === OUTPUTS ===
    # date_out [datetime]   Final time
    # Function
    date_out = date + timestep
    return date_out
```