LPMS-B2 を用いた位置推定プログラム LPMS-self-localization

LPMS-self-localization は LPMS-B2 のデータから装置の位置を推定するプログラムである．

LPMS-B2 は加速度，角速度，磁場それぞれについて三軸のセンサーを持つ装置である． センサーの測定データの時間変化に加え，測定データから計算された装置自身の姿勢（向きと回転）を csv として出力できる．

LPMS-self-localization は LPMS-B2 の出力した加速度と姿勢のデータを入力としてうけとる． 回転操作とフィルター，積分計算を行い，装置の速度と位置を推定して csv として出力する．

```
$ < in.csv | ./lpms_self_localization.py | out.csv
```

LPMS-B2

LPMS-B2 の操作説明

まず， LPMS-B2の取り扱いについて以下に示す． 詳しくは公式サイトを参照すること．

本体の充電

USB ケーブルによって充電を行う．装置の状態と充電状況は LED に表示される．

	LED	色	バッテリー
未接続	点滅	青	> 10%
		赤	< 10%
接続	ゆっくり点滅	青	> 10%
		赤	< 10%
充電中	点灯	緑	> 90%
		青	20% ~ 90%
		赤	< 20%

Windows 設定

設定 > **デバイス** > **Bluetooth とその他のデバイス** の順で選択する． **「Bluetooth またはその他のデバイスを追加する」** から **LPMSB2-xxxxxx** を追加する． （接続時 PIN コードの入力を求められた場合，**1234** を入力する．）

LPMS-Control のインストール

インストーラーをダウンロードし，指示に従う．

LPMS-Control の設定

インストールされた LPMS-Control を起動する．

ツールバーの  を選択する． **Scan devices** から，上で接続したデバイスを選択し，**Add device** を実行する．

ツールバーの  を選択し，接続を行う． ウィンドウの左側 **Connected devices** に，接続済みのデバイス一覧が表示される． これをクリックすると，デバイスごとに計測についての設定ができる．

計測の実行

データの記録をしない計測

ツールバーの  を選択することで計測が開始し，画面のグラフが更新される．

ツールバーの  を選択することでグラフの更新が停止される．

データの記録をする計測

ツールバーの  を選択することで，計測データの保存先を指定する．

ツールバーの  を選択することでで画面のグラフが更新され，記録を開始する．

ツールバーの  を選択することで記録が終了し，データが保存される．

LPMS-self-localization

実行環境

- OS
  - Windows10
- 言語
  - Python 3.10.2

LPMS-self-localization の操作説明

LPMS-self-localization はコマンドラインアプリケーションである．

以下，スクリプト本体である `lpms_self_localization.py` と入力ファイル `in.csv` がカレントディレクトリに存在するとして手順を解説する． 実際の環境に応じて適宜読み替える．

入力ファイル

LPMS-B2 の出力した csv を想定している.

```
SensorId, TimeStamp (s), FrameNumber, AccX (g), AccY (g), AccZ (g), GyroX (deg/s),
GyroY (deg/s), GyroZ (deg/s), MagX (uT), MagY (uT), MagZ (uT), EulerX (deg), EulerY
(deg), EulerZ (deg), QuatW, QuatX, QuatY, QuatZ, LinAccX (g), LinAccY (g), LinAccZ
(g), Pressure (kPa), Altitude (m), Temperature (degC), HeaveMotion (m)
1, -0.0000, 0, -0.00133558, 0.00810515, -1.00156462, -0.02118125, 0.04246204,
-0.04839651, -5.48615694, -19.48448944, -50.30708694, -0.39996749, -0.02038799,
13.98081589, 0.99256057, 0.00344277, 0.00060138, -0.12170178, 0.00098058,
-0.00113269, 0.00158901, 0.00000000, 0.00000000, 0.00000000, 0.00000000
1, 0.0100, 1, -0.00057623, 0.00844607, -1.00184143, 0.06752539, -0.06167802,
-0.01470590, -5.06619167, -20.21798515, -50.47650909, -0.40040997, -0.02113543,
13.98071480, 0.99256063, 0.00344581, 0.00060832, -0.12170088, 0.00021216,
-0.00146058, 0.00186592, 0.00000000, 0.00000000, 0.00000000, 0.00000000
~~~
```

一行目は index で， 二行目以降はサンプリング周期（サンプリング周波数の逆数）ごとの記録である．ただし，抜け値があっても補完される.

以下の項目の列を使用する.

- `TimeStamp (s)`
- `QuatW` , `QuatX` , `QuatY` , `QuatZ`
  - 装置の姿勢を表すクォータニオン.
- `LinAccX (g)` , `LinAccY (g)` , `LinAccZ (g)`
  - 装置のローカル座標（装置とともに回転する座標）における加速度ベクトル.

出力ファイル

出力も csv ファイルである.

```
TimeStamp (s),GlbAccX (m/s^2),GlbAccY (m/s^2),GlbAccZ (m/s^2),GlbVelX (m/s),GlbVelY
(m/s),GlbVelZ (m/s),GlbPosX (m),GlbPosY (m),GlbPosZ (m)
0.0,-0.0014813453779337921,0.00222195072336769,-0.0026821516633868453,0.00105457575891
05,-4.676710606190995e-05
0.01,-0.007872993952671024,-0.0027848474389441406,2.4489575976874636e-
05,0.001050108305391559,-0.0005263783494378711,-0.0004590361127746628,0.00017440723503
05
0.02,-0.007780479256624093,0.007715860968205078,-0.008363194049216783,0.00101418835235
05
~~~
```

一行目は index で， 二行目以降はサンプリング周期（サンプリング周波数の逆数）ごとの記録である.

以下の項目の列が出力される.

- `TimeStamp (s)`
- `GlbAccX (m/s^2)` , `GlbAccY (m/s^2)` , `GlbAccZ (m/s^2)`
  - 装置のグローバル座標（地面に固定された座標）における加速度ベクトル.

- `GlbVelX (m/s)`, `GlbVelY (m/s)`, `GlbVelZ (m/s)`
  - 装置のグローバル座標（地面に固定された座標）における速度ベクトル.
- `GlbPosX (m)`, `GlbPosY (m)`, `GlbPosZ (m)`
  - 装置のグローバル座標（地面に固定された座標）における位置ベクトル.

操作方法と例

```
$ < in.csv | ./lpms_self_localization.py | out.csv
```

または

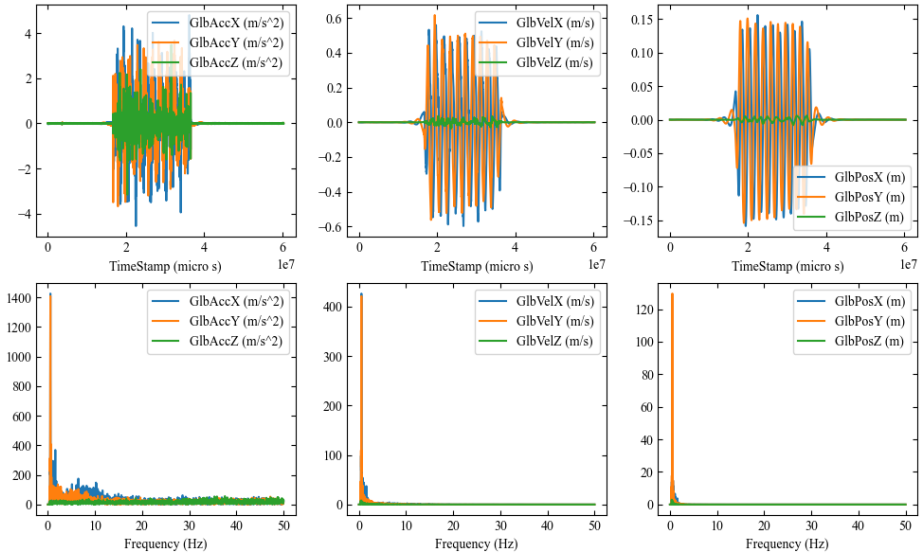
```
$ ./lpms_self_localization.py in.csv -o out.csv
```

で，処理の結果を`out.csv`に出力する.

簡単なグラフを確認したい場合は

```
$ ./lpms_self_localization.py in.csv -o out.csv -p plot.png
```

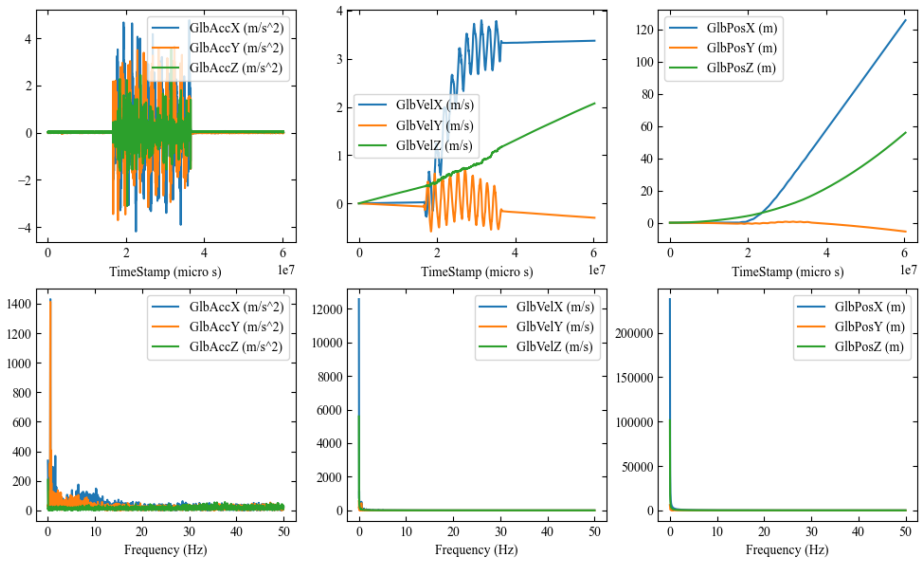
でプロットを`plot.png`に出力する.



さらに，フィルタリングを行わない場合は

```
$ ./lpms_self_localization.py in.csv -o out.csv -p plot.png --no-acc-filter --no-
vel-filter --no-pos-filter
```

のように設定する.



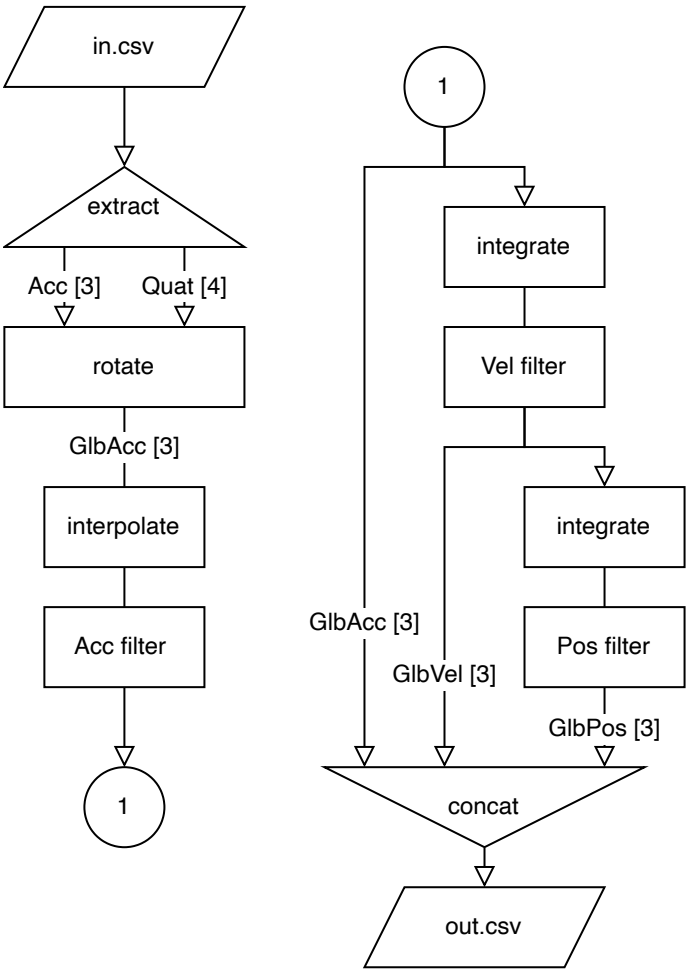
オプションの解説

- `-h, --help`
  - ヘルプを表示する.
- `-o OUTPUT, --output OUTPUT`
  - 推定された加速度，速度，位置のデータを csv として出力する先を指定する． 指定がなければ標準出力に出力する.
- `-p PLOT, --plot PLOT`
  - プロット (png) の出力先を選ぶ． 指定がなければ出力しない.
- `-f FREQ, --freq FREQ`
  - 入力データのサンプリング周波数 (Hz) を指定する． 指定がなければデータから推定する.
- `-i INTERPOLATE, --interpolate INTERPOLATE`
  - 抜け値の補完メソッドを指定する． pandas.DataFrame.interpolate によって補完が行われる． 指定がなければ線形補完とする.
- `--acc-filter ACC_FILTER ACC_FILTER`
  - 加速度に対するハイパスフィルターの阻止域端周波数 [Hz] と通過域端周波数 [Hz] を指定する． 指定がなければ 0.1, 0.3 とする
- `--no-acc-filter`
  - 加速度に対するハイパスフィルターを無効化する.
- `--vel-filter VEL_FILTER VEL_FILTER`
  - 速度に対するハイパスフィルターの阻止域端周波数 [Hz] と通過域端周波数 [Hz] を指定する． 指定がなければ 0.1, 0.3 とする
- `--no-vel-filter`
  - 速度に対するハイパスフィルターを無効化する.
- `--pos-filter POS_FILTER POS_FILTER`

    - 位置に対するハイパスフィルターの阻止域端周波数 [Hz] と通過域端周波数 [Hz] を指定する． 指定がなければ 0.1, 0.3 とする
- `--no-pos-filter`
    - 位置に対するハイパスフィルターを無効化する．

LPMS-self-localization が行う処理の説明

LPMS-self-localization が行う処理をフローチャート図で示す．



1. extract
    - `in.csv` から，ローカル座標（装置とともに回転する座標）における加速度ベクトルである `LinAccX (g)`，`LinAccY (g)`，`LinAccZ (g)` 列（以下，まとめて `Acc[3]` と呼ぶ）と姿勢を表すクォータニオンの `QuatW`，`QuatX`，`QuatY`，`QuatZ` 列（以下，`Quat[4]` と呼ぶ）を取り出す．
2. rotate
    - `Acc[3]` を `Quat[3]` によって回転し，グローバル座標（地面に固定された座標）における加速度ベクトル `GlbAcc[3]` を得る．
3. interpolate
    - `GlbAcc[3]` の抜け値を補完し，時間の刻み幅を一定にする．
4. Acc filter

- `GlbAcc[3]` にバターワースフィルタリングを行う.

5. integrate
   - `GlbAcc[3]` を積分して，速度ベクトル `GlbVel[3]` を得る.
6. Vel filter
   - `GlbVel[3]` にバターワースフィルタリングを行う.
7. integrate
   - `GlbVel[3]` を積分して，位置 `GlbPos[3]` を得る.
8. Pos filter
   - `GlbPos[3]` にバターワースフィルタリングを行う.
9. concat
   - `GlbAcc[3]` , `GlbVel[3]` , `GlbPos[3]` を結合して `out.csv` とする.

ソースコード

```
#! /usr/bin/env python
import sys
import argparse
import pandas as pd
import numpy as np
import quaternion
from scipy import fftpack as fft
from scipy import integrate
from scipy import signal
from matplotlib import pyplot as plt

def main():
    # 重力加速度
    g : float = 9.80665
    # 列名
    sTimeStamp       = 'TimeStamp (s)'
    sTimeStampMicroS = 'TimeStamp (micro s)'
    sQuats           = ['QuatW', 'QuatX', 'QuatY', 'QuatZ']
    sLinAccs         = ['LinAccX (g)', 'LinAccY (g)', 'LinAccZ (g)']
    sGlbAccs         = ['GlbAccX (m/s^2)', 'GlbAccY (m/s^2)', 'GlbAccZ (m/s^2)']
    sGlbVels         = ['GlbVelX (m/s)', 'GlbVelY (m/s)', 'GlbVelZ (m/s)']
    sGlbPoss         = ['GlbPosX (m)', 'GlbPosY (m)', 'GlbPosZ (m)']

    # 引数の処理
    parser = argparse.ArgumentParser( description='LPMS の出力した csv から位置を推定し，
csv として出力します．' )
    parser.add_argument( 'input', type=argparse.FileType('r'), nargs='?',
default=sys.stdin,
            help='入力ファイルを指定します．　指定がなければ標準入力を受け取ります．' )
    parser.add_argument( '-o', '--output', type=argparse.FileType('w'),
default=sys.stdout,
            help='推定された加速度，速度，位置のデータを csv として出力する先を指定します．　指定
がなければ標準出力に出力されます．' )
    parser.add_argument( '-p', '--plot',
```

```
            help='プロット (png) の出力先を選びます．指定がなければ出力しません．' )
    parser.add_argument( '-f', '--freq', type=int,
            help='入力データのサンプリング周波数 (Hz) を指定します．指定がなければデータから推定します．' )
    parser.add_argument( '-i', '--interpolate',
            help='抜け値の補完メソッドを指定します． pandas.DataFrame.interpolate によって補完が行われます．指定がなければ線形補完です．', default='linear' )
    parser.add_argument( '--acc-filter', nargs=2, default=[0.1, 0.3],
            help='加速度に対するハイパスフィルターの阻止域端周波数 [Hz] と通過域端周波数 [Hz] を指定します．指定がなければ 0.1, 0.3 とします．')
    parser.add_argument( '--no-acc-filter', action='store_true',
            help='加速度に対するハイパスフィルターを無効化します．')
    parser.add_argument( '--vel-filter', nargs=2, default=[0.1, 0.3],
            help='速度に対するハイパスフィルターの阻止域端周波数 [Hz] と通過域端周波数 [Hz] を指定します．指定がなければ 0.1, 0.3 とします．')
    parser.add_argument( '--no-vel-filter', action='store_true',
            help='速度に対するハイパスフィルターを無効化します．')
    parser.add_argument( '--pos-filter', nargs=2, default=[0.1, 0.3],
            help='位置に対するハイパスフィルターの阻止域端周波数 [Hz] と通過域端周波数 [Hz] を指定します．指定がなければ 0.1, 0.3 とします．')
    parser.add_argument( '--no-pos-filter', action='store_true',
            help='位置に対するハイパスフィルターを無効化します．')
    args = parser.parse_args()

    # 標準入力の csv から、必要な列を DataFrame に
    dfInput = pd.read_csv( args.input, engine='python', sep=',\s+' )
    # TimeStamp 列を float (s) から int (micro s) に変換し、 index に指定
    timeStampMicroS : pd.Index = round( dfInput[ sTimeStamp ] * 1000000 ) . rename( sTimeStampMicroS ) . astype(int)
    dfInput.index = timeStampMicroS
    if args.freq == None:
        # データのサンプリング周期 (micro s) として TimeStamp (micro s) の差分の最頻値を取る
        samplingCycle : int = int( timeStampMicroS. diff() . mode() [0] )
        # サンプリング周波数 (Hz)
        samplingFreq  : int = int( 1000000 / samplingCycle )
    else:
        samplingFreq  : int = args.freq
        samplingCycle : int = int( 1000000 / samplingFreq )
    samplingTime : int = timeStampMicroS. iloc[-1]

    # クォータニオンによる回転を行い，グローバル座標での加速度を得る
    glbAcc = g * dfQuatRotation( dfInput, sLinAccs, sQuats, sGlbAccs )
    # データの抜けを args に従い補完
    glbAcc = glbAcc . reindex( range( 0, samplingTime + 1, samplingCycle ) ) . interpolate( method=args.interpolate , axis='index' )
    # フィルタリングと時間積分を繰り返して速度と位置を得る
    if not args.no_acc_filter:
        glbAcc = dfFilter( glbAcc, samplingFreq, args.acc_filter[1], args.acc_filter[0], 'high' )
    glbVel = dfIntegrate( glbAcc, sGlbVels )
    if not args.no_vel_filter:
        glbVel = dfFilter( glbVel, samplingFreq, args.vel_filter[1],
```

```
args.vel_filter[0], 'high' )
    glbPos = dfIntegrate( glbVel, sGlbPoss )
    if not args.no_pos_filter:
        glbPos = dfFilter( glbPos, samplingFreq, args.pos_filter[1],
args.pos_filter[0], 'high' )

    # csv を出力
    dfOutput = pd.concat( [ glbAcc, glbVel, glbPos ], axis=1 )
    dfOutput.index = ( dfOutput.index / 1000000 ) . rename( sTimeStamp )
    dfOutput . to_csv( args.output )

    # plot 用
    if args.plot != None:
        fftAcc = abs( dfFFT( glbAcc, samplingFreq ) )
        fftVel = abs( dfFFT( glbVel, samplingFreq ) )
        fftPos = abs( dfFFT( glbPos, samplingFreq ) )
        fig, ax = plot6( [glbAcc, glbVel, glbPos, fftAcc, fftVel, fftPos ] )
        fig.savefig( args.plot )

def dfQuatRotation( df : pd.DataFrame, vectCols : list, quatCols : list, newColNames
: list ):
    # DataFrame の vect ( 3列 ) を quat ( 4列 ) で回転する
    index = df.index
    vect = df[vectCols].to_numpy()
    quat = quaternion.as_quat_array( df[quatCols].to_numpy() )
    rotatedVect = quaternion.as_vector_part( quat.conjugate() *
quaternion.from_vector_part( vect ) * quat )
    # 回転したベクトルに新しい列名をつけて DataFrame として返す
    return pd.DataFrame( data=rotatedVect, index=index, columns=newColNames )

def dfIntegrate( df : pd.DataFrame, newColNames : list ):
    # DataFrame を index/1000000 で積分する
    # index の単位を micro s とすることで時間積分となる
    index = df.index
    arr = df.to_numpy()
    arrInt = integrate.cumtrapz( arr, x=index.to_numpy() / 1000000, axis=0,
initial=0 )
    return pd.DataFrame( data=arrInt, index=index, columns=newColNames )

def dfFFT( df : pd.DataFrame, fSample ):
    # index を float (Hz) に
    index = pd.Index( np.linspace( 0, fSample, len(df) ), name='Frequency (Hz)' )
    columns = df.columns
    arr = df.to_numpy()
    arrFFT = fft.fft( arr, axis=0 )
    # ナイキスト周波数以降は切り捨て
    return pd.DataFrame( data=arrFFT, index=index, columns=columns ) . query('index
<= ' + str( fSample/2 ))

def dfFilter( df : pd.DataFrame, fSample, fPass, fStop, bType ):
    # バターワースフィルタリングを行う
    index = df.index
```

```
    columns = df.columns
    arr = df.to_numpy()
    fNiquist = fSample / 2
    wPass = fPass / fNiquist
    wStop = fStop / fNiquist
    gPass = 3
    gStop = 40
    N, Wn = signal.buttord( wPass, wStop, gPass, gStop )
    b, a = signal.butter( N, Wn, bType)
    arrFilt = signal.filtfilt(b, a, arr, axis=0)
    return pd.DataFrame( data=arrFilt, index=index, columns=columns )

def plotInit():
    # plot 初期化
    # フォント指定
    plt.rcParams['font.size'] = 10
    plt.rcParams['font.family'] = 'Times New Roman'
    # 目盛を内側に
    plt.rcParams['xtick.direction'] = 'in'
    plt.rcParams['ytick.direction'] = 'in'

def plot1(df):
    # 1つのプロットを出力する
    plotInit()
    fig, ax= plt.subplots()
    df.plot(ax=ax)
    ax.xaxis.set_ticks_position('both')
    ax.yaxis.set_ticks_position('both')
    return fig, ax

def plot6(dfs):
    # 6つのプロットを並べる
    plotInit()
    # Figure インスタンスを作成
    fig, axes = plt.subplots(2, 3, figsize=(12,7))
    i = 0
    for axarr in axes:
        for ax in axarr:
            dfs[i].plot(ax=ax)
            i += 1
            ax.xaxis.set_ticks_position('both')
            ax.yaxis.set_ticks_position('both')
    return fig, axes

if __name__ == '__main__':
    main()
```