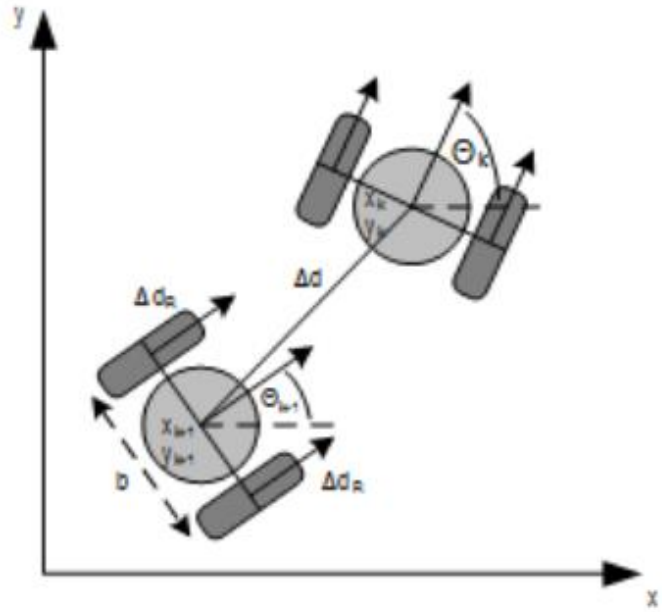
Localization

Odometry

Odometry

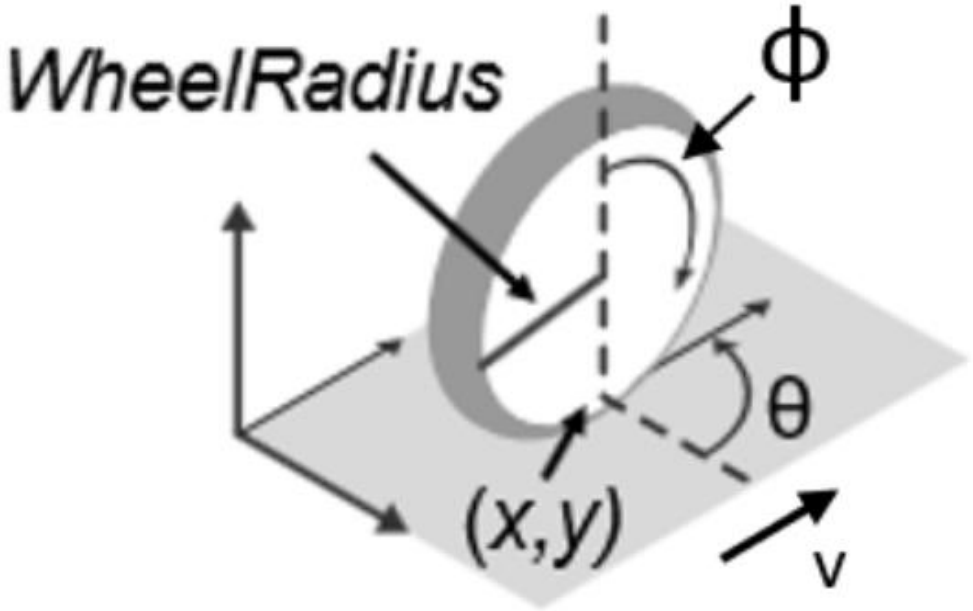
1) Definition

- 로봇·차량의 이동 경로와 거리를 센서 데이터를 바탕으로 추정하는 '주행기록계' 기술로, 보통 절대 위치가 아니라 시작점 기준의 상대적 위치 변화
- 전통적으로 휠 엔코더(encoder)와 IMU를 활용해 이동을 추정하며, 오차는 슬립·스키딩 등으로 누적오차 발생
- Wheel Odometry, IMU Odometry, Visual Odometry, LiDAR Odometry 등 Odom을 추정하기 위한 다양한 기법이 존재하며, sensor fusion을 통한 정확도 향상



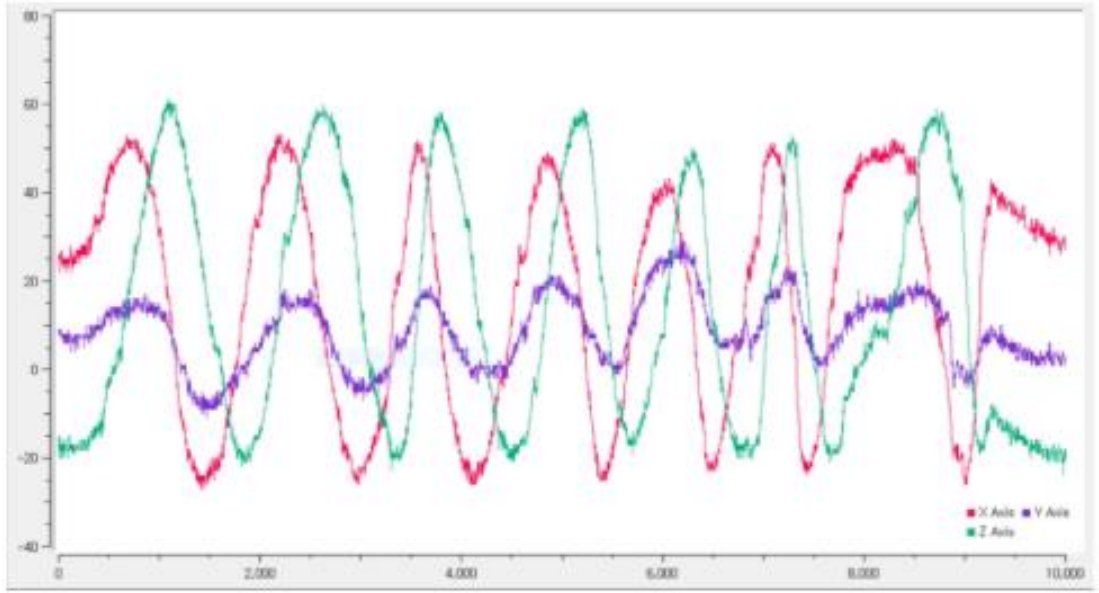
2) Wheel Odometry

- 바퀴 회전 정보를 이용해 로봇의 이동 거리와 자세를 추정하는 가장 기본적인 위치 추정 방법
- 바퀴에 달린 encoder 가 회전량을 측정하면, 이를 통해 로봇이 얼마나 이동했는지 계산



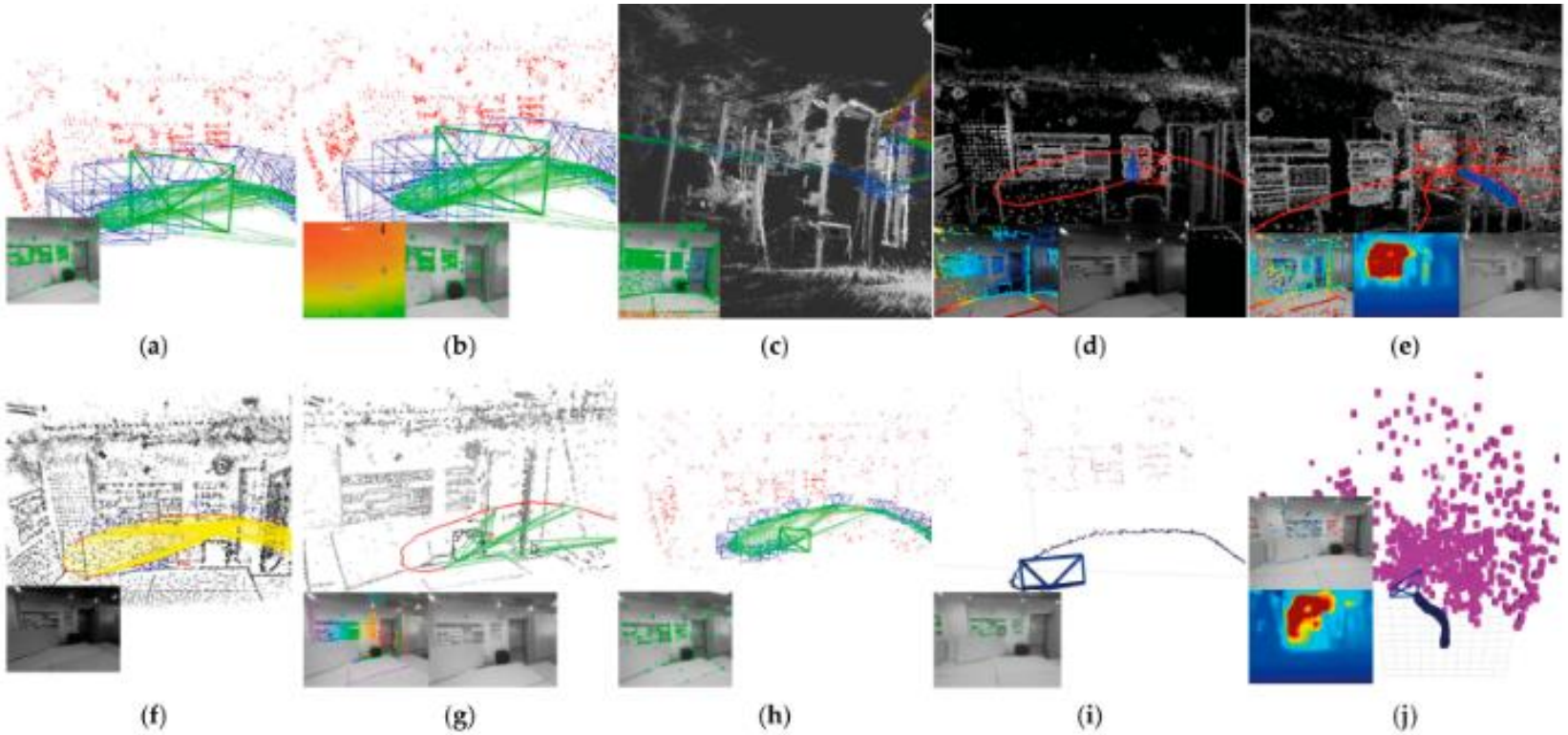
3) IMU Odometry

- 가속도계와 자이로스코프를 사용하여 로봇이나 기체의 회전 및 선형 가속도를 측정, 위치와 자세(Pose)를 추정하는 기술
- Wheel 엔코더 기반 방식보다 더 정확한 방향(Yaw) 측정에 유용하며, 단독 사용 시 누적 오차(Drift)가 발생
- IMU를 통해 가속도, 각속도, Heading을 계측할 수 있으며, 이를 적분하여 Odometry 추정이 가능함



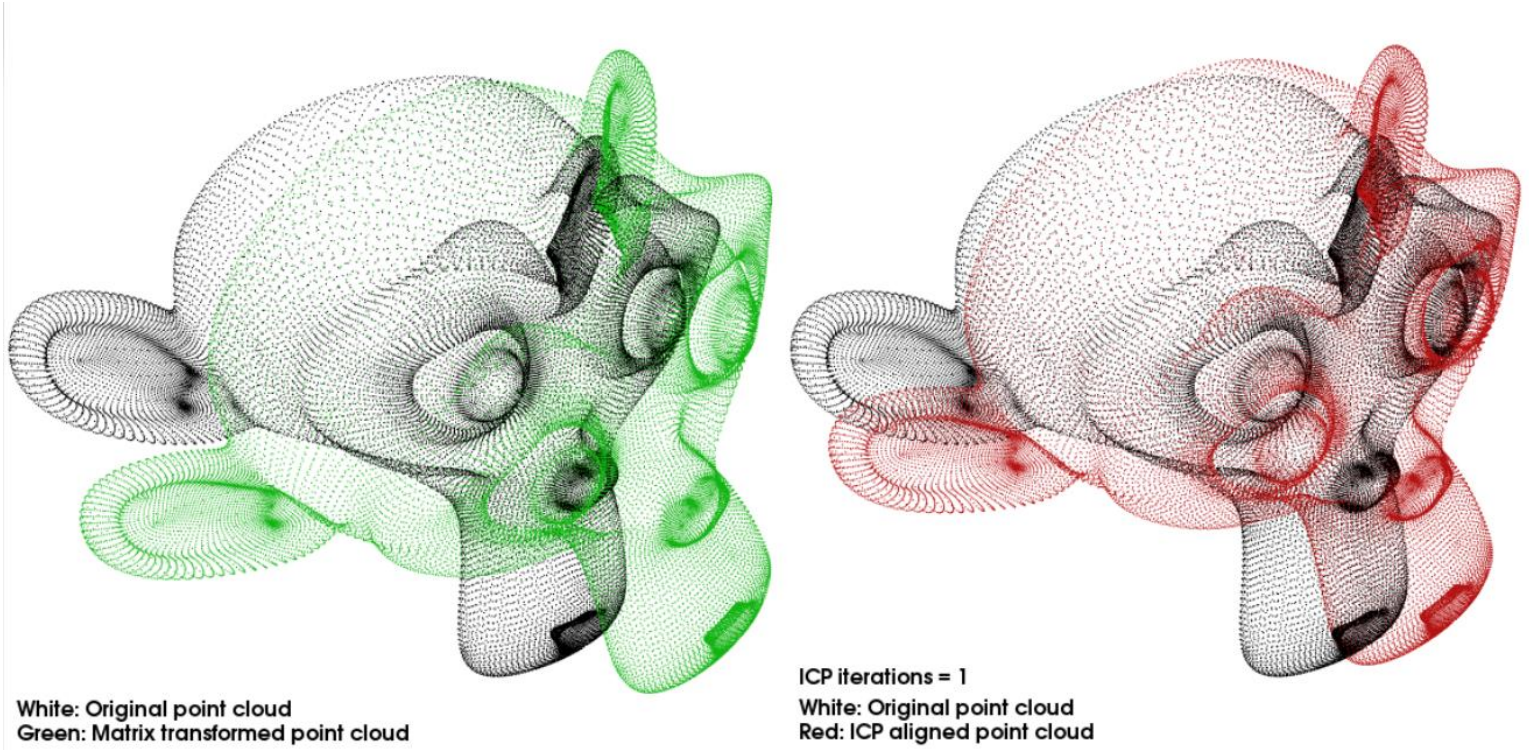
4) Visual Odometry

- 카메라 영상에서 프레임 간 움직임을 추정해서 로봇/카메라의 궤적(odometry)을 구하는 방법
- 특징점을 기반으로 다음 프레임에서 해당 특징점들의 이동 패턴을 기반으로 카메라의 회전과 이동을 추정하여 로봇의 Odometry를 산출하는 방식



5) LiDAR Odometry

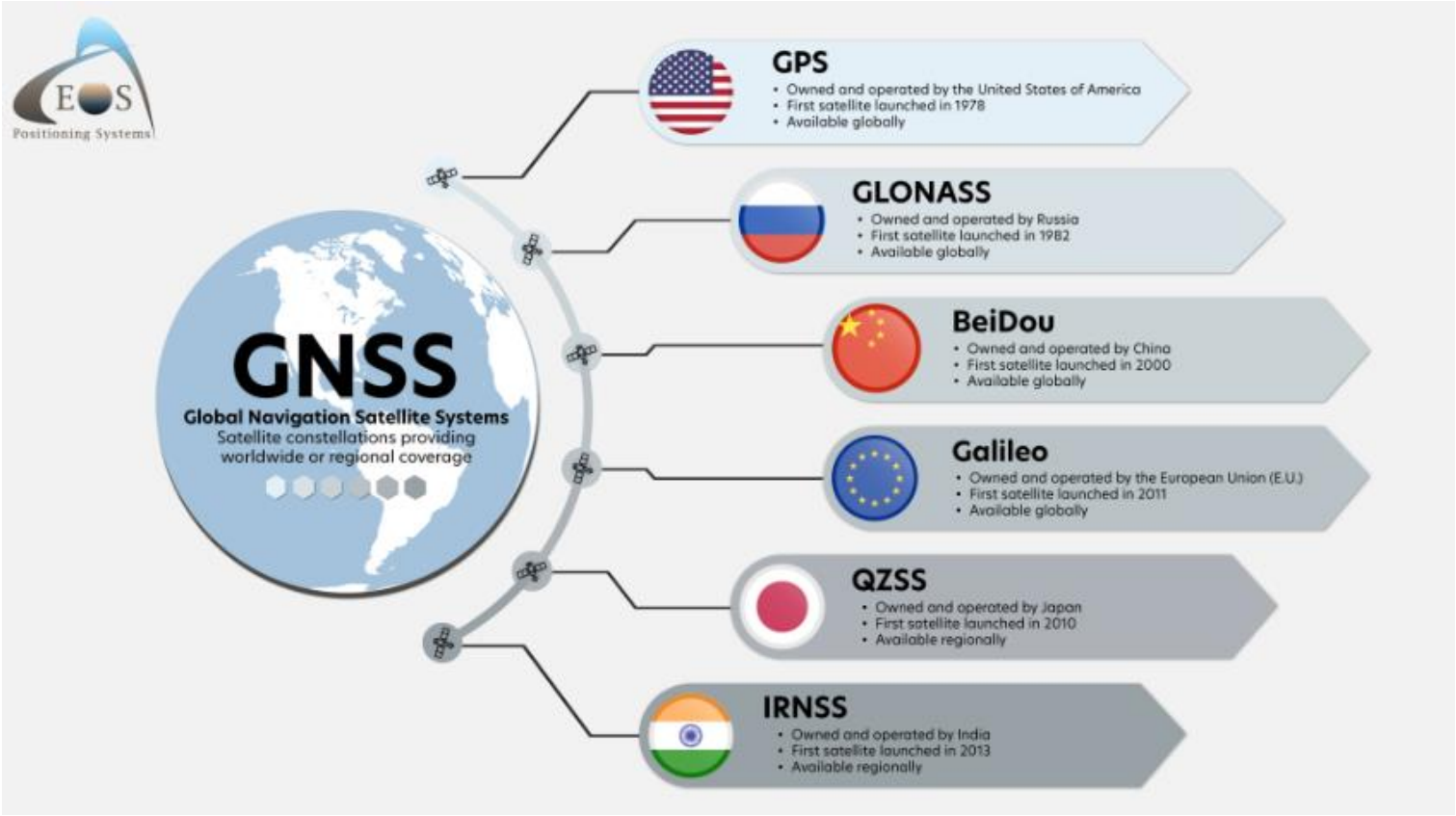
- LiDAR 스캔(포인트클라우드)을 정합(register)해서 센서의 이동과 자세 변화를 추정하는 방법
- 조명 영향 거의 없으며, 야외에 강인하고 정거리에 정확한 효율을 유지할 수 있음
- 계산량이 크며, 구조가 없는 환경에서 약함



GNSS

1) Definition

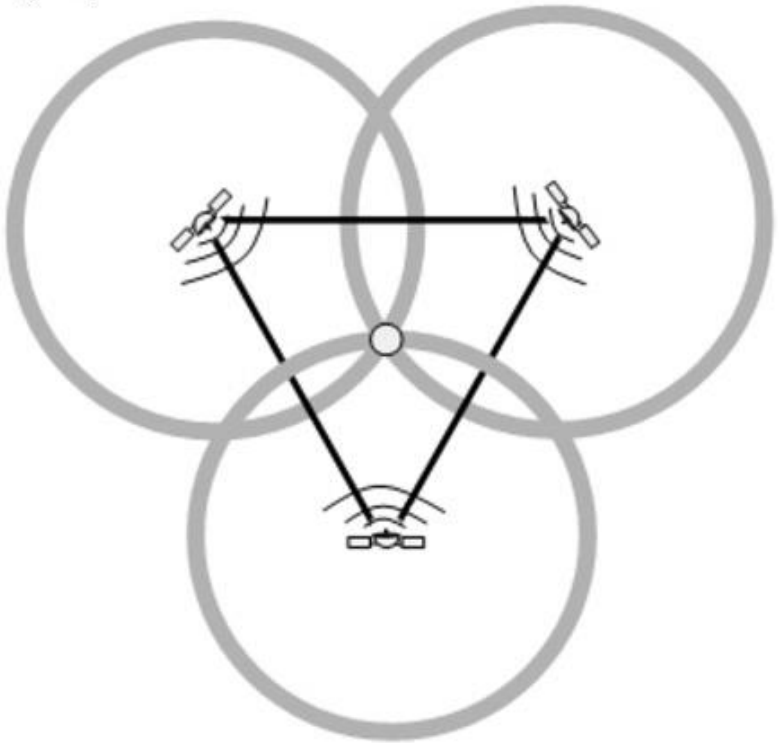
- GNSS(Global Navigation Satellite System): 위성으로 위치/시간을 제공하는 시스템 총칭
- 대표 시스템
  - GPS(미국), GLONASS(러시아), Galileo(EU), BeiDou(중국)



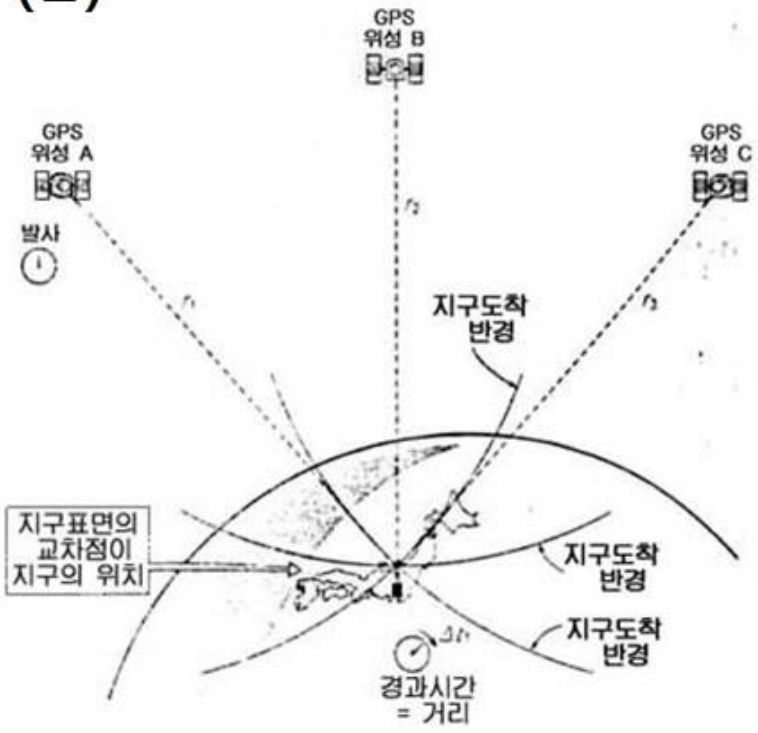
1) GNSS Localization

- GNSS를 이용한 위치 측정은 위성에서 신호를 보내고, 수신기가 받은 시간 차로 거리를 추정함
- 수신기의 위치는 삼각 측량법을 이용하여 측정하게 됨

(ㄱ)

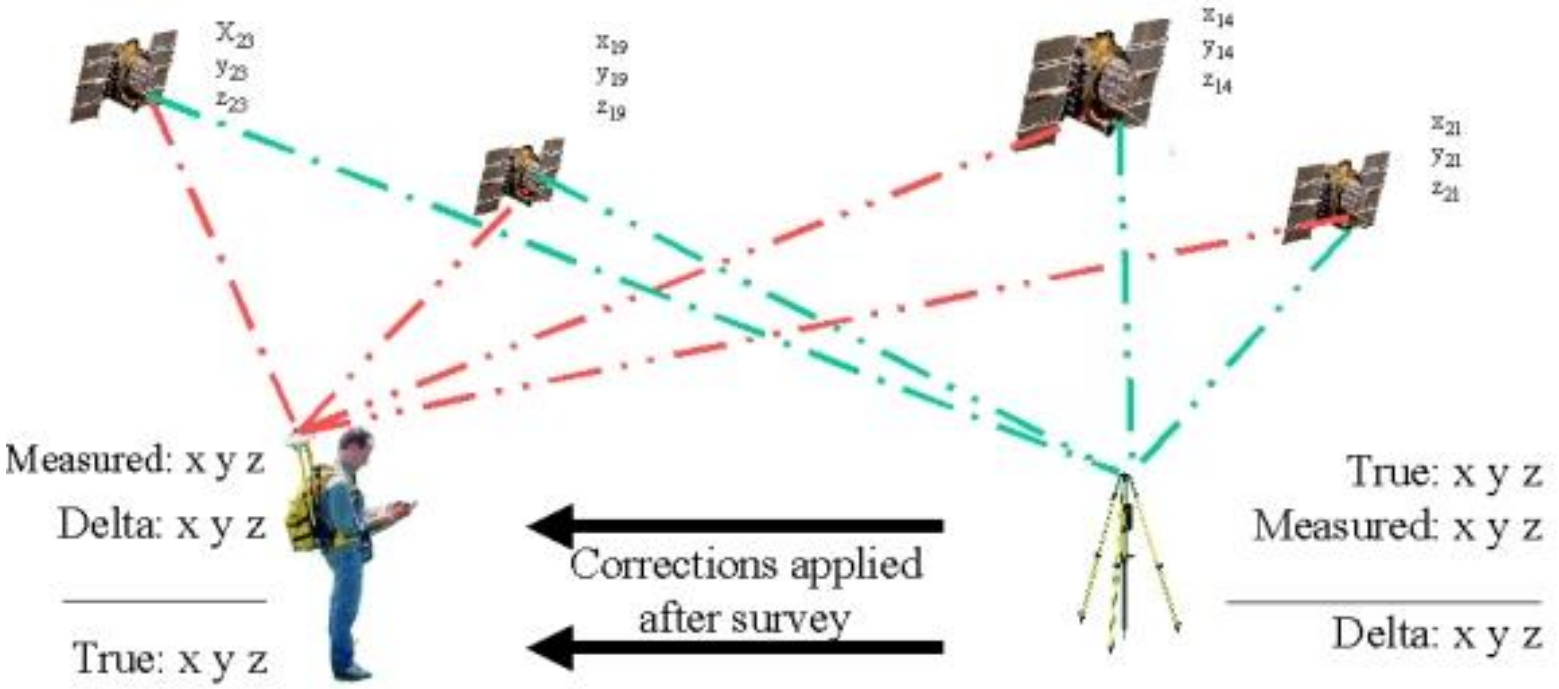


(ㄴ)



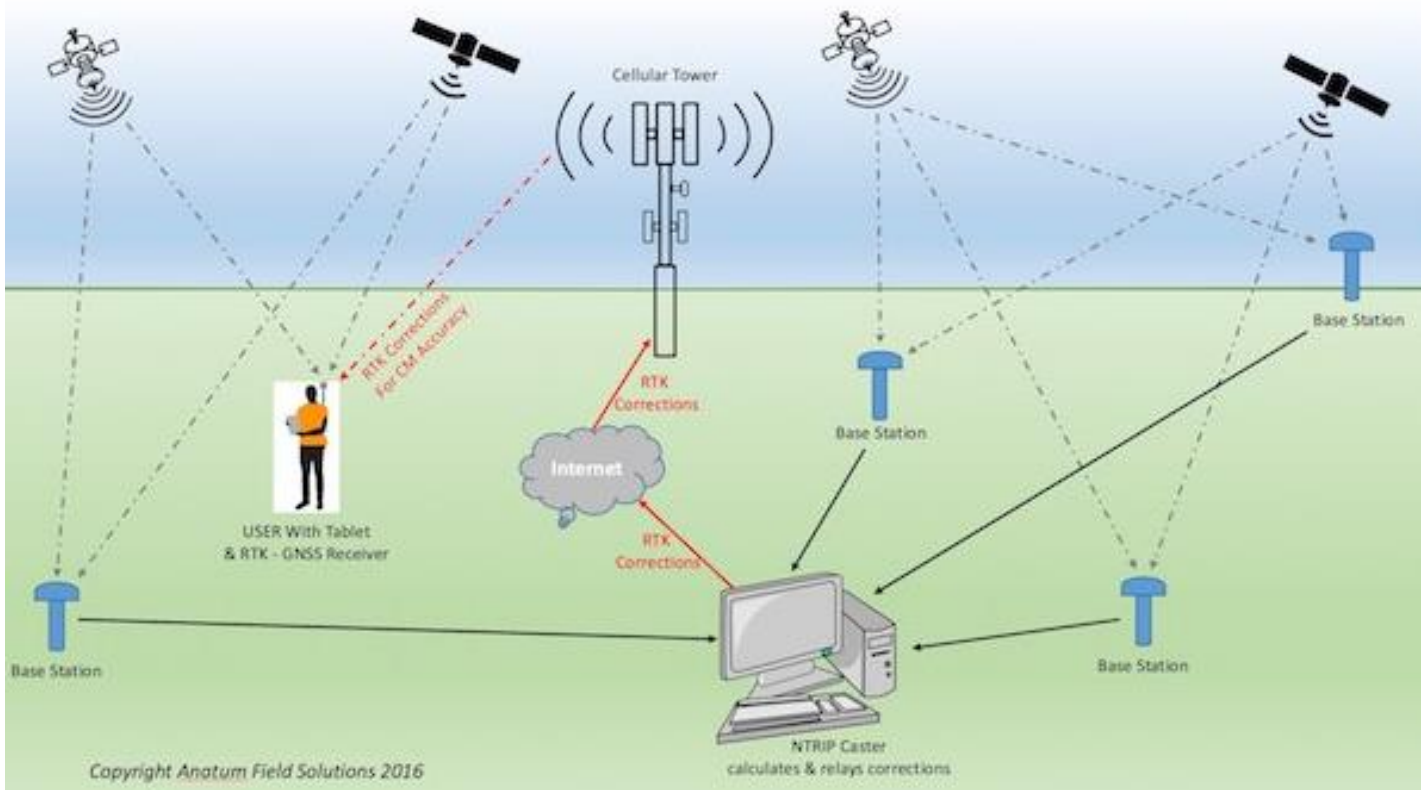
3) RTK(Differential GPS)

- 일반적인 GPS 오류를 수정하는 데 도움이 되는 기술
- 이동국과 기지국 사이의 거리를 고려해 이동국의 위치에서 보정값을 산술적으로 계산
- 코드 오차를 빼는 방식으로 수m 급 오차를 가짐



2) RTK(Real-Time Kinematic GPS)

- 일반적인 GPS 오류를 수정하는 데 도움이 되는 기술
- 이동국과 기지국 사이의 거리를 고려해 이동국의 위치에서 보정값을 산술적으로 계산
- 파장 단위를 분석하여 cm 급 오차를 가짐



Utilization of GNSS Data

Utilization of GNSS Data

1) Fake Data Package

- GPS Data와 IMU Data를 가상으로 Publish 하는 패키지 설치 및 빌드

```
$ mkdir –p ~/gps_ws/src

$ cd ~/gps_ws/src

$ git clone https://github.com/RASLab-sjbyun/fake_data.git

$ cd ~/gps_ws

$ colcon build

$ source install/setup.bash

$ ros2 launch fake_data fake_data.launch.py
```

Utilization of GNSS Data

2) ROS2 Data Type

- ROS2에서 사용되는 GNSS Data는 sensor_msgs/msg/NavSatFix Type를 사용함

```
lab@lab:~$ ros2 interface show sensor_msgs/msg/NavSatFix
# Navigation Satellite fix for any Global Navigation Satellite System
#
# Specified using the WGS 84 reference ellipsoid

# header.stamp specifies the ROS time for this measurement (the
#        corresponding satellite time may be reported using the
#        sensor_msgs/TimeReference message).
#
# header.frame_id is the frame of reference reported by the satellite
#        receiver, usually the location of the antenna.  This is a
#        Euclidean frame relative to the vehicle, not a reference
#        ellipsoid.
std_msgs/Header header
        builtin_interfaces/Time stamp
                int32 sec
                uint32 nanosec
        string frame_id

# Satellite fix status information.
NavSatStatus status
        #
        int8 STATUS_NO_FIX =  -1        #
        int8 STATUS_FIX =      0        #
        int8 STATUS_SBAS_FIX = 1        #
        int8 STATUS_GBAS_FIX = 2        #
        int8 status
        uint16 SERVICE_GPS =     1
        uint16 SERVICE_GLONASS = 2
        uint16 SERVICE_COMPASS = 4      #
        uint16 SERVICE_GALILEO = 8
        uint16 service

# Latitude [degrees]. Positive is north of equator; negative is south.
float64 latitude

# Longitude [degrees]. Positive is east of prime meridian; negative is west.
float64 longitude

# Altitude [m]. Positive is above the WGS 84 ellipsoid
# (quiet NaN if no altitude is available).
float64 altitude

# Position covariance [m^2] defined relative to a tangential plane
# through the reported position. The components are East, North, and
# Up (ENU), in row-major order.
#
# Beware: this coordinate system exhibits singularities at the poles.
float64[9] position_covariance

# If the covariance of the fix is known, fill it in completely. If the
# GPS receiver provides the variance of each measurement, put them
# along the diagonal. If only Dilution of Precision is available,
# estimate an approximate covariance from that.

uint8 COVARIANCE_TYPE_UNKNOWN = 0
uint8 COVARIANCE_TYPE_APPROXIMATED = 1
uint8 COVARIANCE_TYPE_DIAGONAL_KNOWN = 2
uint8 COVARIANCE_TYPE_KNOWN = 3

uint8 position_covariance_type
```

- std_msgs/Header header
  - stamp(측정 시각)
  - frame_id(프레임 이름)

- NavSatStatus status
  - status (gps fix 품질)
  - Service (사용 GNSS 시스템)

- latitude (위도)
- Longitude(경도)
- altitude(고도)

- position_covariance (공분산 행렬)

- position_covariance_type

3) Gps기반 Localization Package 생성

- gps 데이터 기반의 Odom 및 TF 발행을 위한 패키지 생성

1. 패키지 생성

   - cd ~/gps_ws/src/
   - ros2 pkg create --build-type ament_python gps_tutorial

4) URDF 정의 및 Static TF 발행

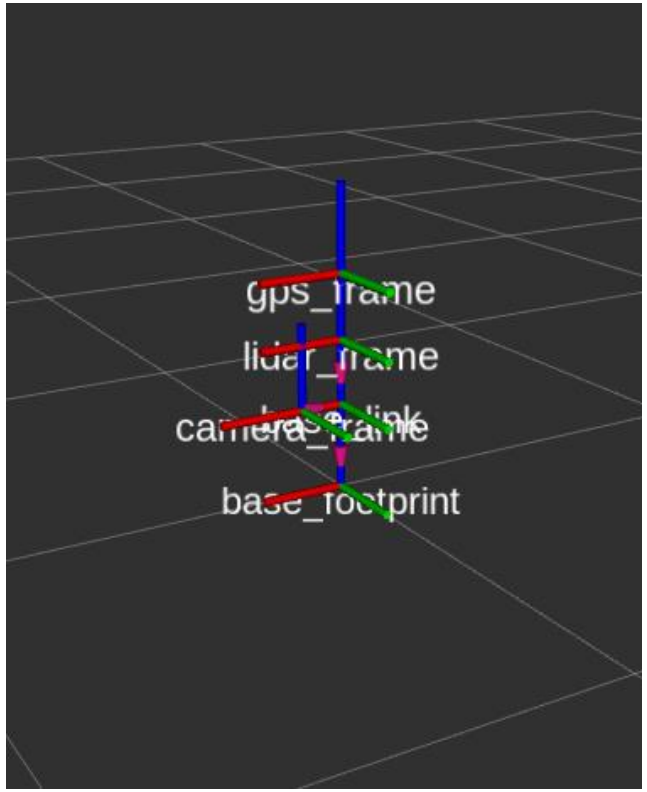
- URDF(Unified Robot Description Format)는 XML 기반의 파일 형식으로, ROS(Robot Operating System) 및 시뮬레이션 환경(Gazebo)에서 로봇의 물리적 구조(Link), 관절(Joint), 기하학적 형태, 관성, 센서 정보를 정의하는 표준

1. urdf 폴더 및 파일 생성
   - cd ~/gps_ws/src/gps_tutorial && mkdir urdf
   - cd urdf && gedit robot.urdf.xacro
2. Launch 폴더 및 파일 생성
   - cd ~/gps_ws/src/gps_tutorial && mkdir launch
   - cd launch && gedit robot_tf.launch.py
3. setup.py 수정
   - cd ~/gps_ws/src/gps_tutorial && gedit setup.py
4. 빌드 및 수행
   - cd ~/gps_ws && colcon build
   - source install/setup.bash
   - ros2 launch gps_tutorial robot_tf.launch.py
5. 시각화 수행
   - rviz2

```
import os
from glob import glob
```

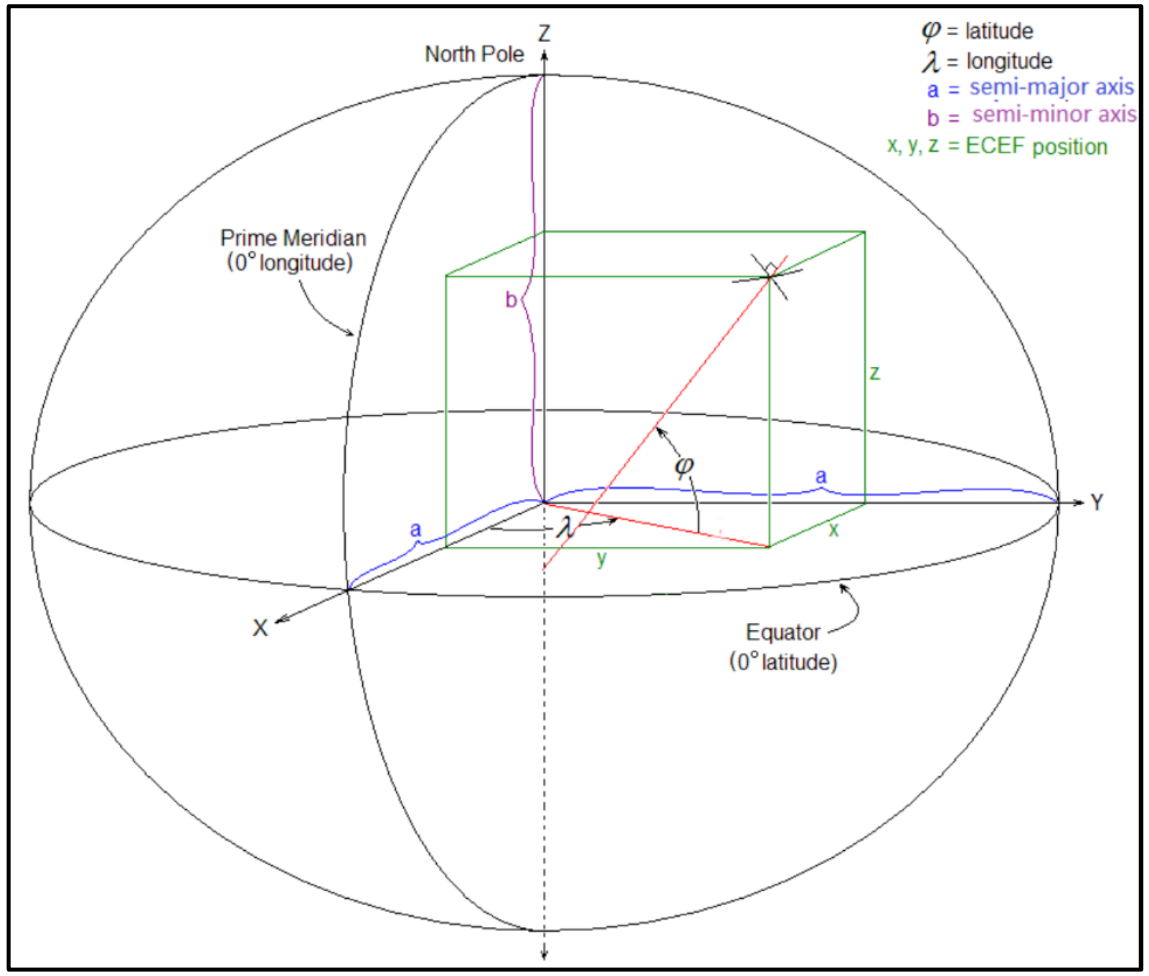
```
# launch 파일 설치
(os.path.join('share', package_name, 'launch'),
 glob('launch/*.launch.py')),

# urdf/xacro 파일 설치
(os.path.join('share', package_name, 'urdf'),
 glob('urdf/*')),
```

```
<?xml version="1.0"?>
<robot name="my_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <!-- ========================
       Links
       ======================== -->
  <link name="base_footprint"/>
  <link name="base_link"/>
  <link name="camera_frame"/>
  <link name="lidar_frame"/>
  <link name="gps_frame"/>

  <!-- ========================
       Fixed joints (TF tree)
       ======================== -->

  <!-- base_footprint -> base_link (z +0.20 m) -->
  <joint name="base_footprint_to_base_link" type="fixed">
    <parent link="base_footprint"/>
    <child  link="base_link"/>
    <origin xyz="0 0 0.20" rpy="0 0 0"/>
  </joint>

  <!-- base_link -> camera_frame (x +0.10 m) -->
  <joint name="base_link_to_camera" type="fixed">
    <parent link="base_link"/>
    <child  link="camera_frame"/>
    <origin xyz="0.10 0 0" rpy="0 0 0"/>
  </joint>

  <!-- base_link -> lidar_frame (z +0.15 m) -->
  <joint name="base_link_to_lidar" type="fixed">
    <parent link="base_link"/>
    <child  link="lidar_frame"/>
    <origin xyz="0 0 0.15" rpy="0 0 0"/>
  </joint>

  <!-- base_link -> gps_frame (z +0.30 m) -->
  <joint name="base_link_to_gps" type="fixed">
    <parent link="base_link"/>
    <child  link="gps_frame"/>
    <origin xyz="0 0 0.30" rpy="0 0 0"/>
  </joint>

</robot>
```



4) Odom Publish

- GPS Data를 활용하여, Odom을 발행하기 위해서는 위도, 경도, 고도를 x, y, z 축으로 변환하는 과정이 필수적임
- GPS Data를 x, y, z로 변환하기 위해서는 위도, 경도, 고도 Data를 기반으로 ECEF 좌표계로 변환하는 과정이 필요함



◆ 지구는 구가 아닌 타원형의 형태를 지님

◆ 이에 따라, 타원 방정식과 법선 방정식을 모두 만족하는 조건을 통해 위도, 경도, 고도 Data를 기반으로 ECEF 좌표계로 변환할 수 있음

$$x = (N + h)cos\varphi cos\lambda$$

$$y = (N + h)cos\varphi sin\lambda$$

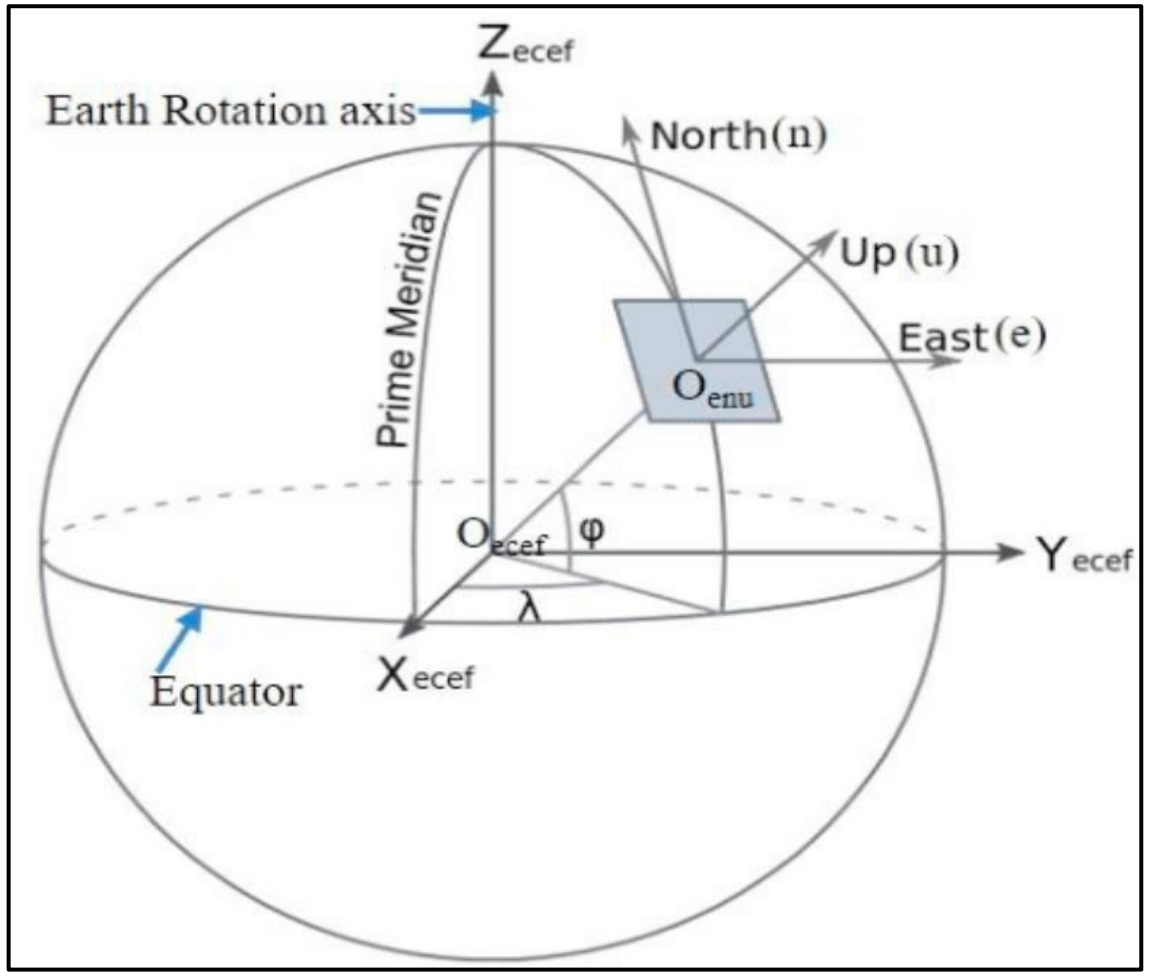
$$z = (N(1 - e^2) + h)sin\varphi$$

$$N = \frac{a}{\sqrt{1 - e^2 \sin^2 \varphi}}$$

```python
WGS84_A = 6378137.0
WGS84_E2 = 6.69437999014e-3

def lla_to_ecef(lat_rad, lon_rad, alt):
    sin_lat = math.sin(lat_rad)
    cos_lat = math.cos(lat_rad)
    sin_lon = math.sin(lon_rad)
    cos_lon = math.cos(lon_rad)

    N = WGS84_A / math.sqrt(1.0 - WGS84_E2 * sin_lat * sin_lat)

    x = (N + alt) * cos_lat * cos_lon
    y = (N + alt) * cos_lat * sin_lon
    z = (N * (1.0 - WGS84_E2) + alt) * sin_lat
    return x, y, z
```

4) Odom Publish

- GPS Data를 활용하여, Odom을 발행하기 위해서는 위도, 경도, 고도를 x, y, z 축으로 변환하는 과정이 필수적임
- ECEF 좌표계로 최종 ENU 좌표계로 시킨 후 Odom을 추정 지구는 타원형이기 때문에 평면좌표로 변환하여야 함



◆ ECEF와 ENU 좌표계는 좌표 축이 회전되어 있는 형태이므로, 회전 행렬을 통해 좌표 변환이 가능함

$$\begin{bmatrix} e \\ n \\ u \end{bmatrix} = R \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \begin{bmatrix} de \\ dn \\ du \end{bmatrix} = R \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix}$$

$$\hat{e} = \begin{bmatrix} -\sin\lambda \\ \cos\lambda \\ 0 \end{bmatrix} \quad \hat{n} = \begin{bmatrix} -\sin\varphi\cos\lambda \\ -\sin\varphi\sin\lambda \\ \cos\varphi \end{bmatrix} \quad \hat{u} = \begin{bmatrix} \cos\varphi\cos\lambda \\ \cos\varphi\sin\lambda \\ \sin\varphi \end{bmatrix}$$

$$R = \begin{bmatrix} -\sin\lambda & \cos\lambda & 0 \\ -\sin\varphi\cos\lambda & -\sin\varphi\sin\lambda & \cos\varphi \\ \cos\varphi\cos\lambda & \cos\varphi\sin\lambda & \sin\varphi \end{bmatrix}$$

```
def ecef_to_enu(x, y, z, ref_x, ref_y, ref_z, ref_lat, ref_lon):
    dx = x - ref_x
    dy = y - ref_y
    dz = z - ref_z

    sin_lat = math.sin(ref_lat)
    cos_lat = math.cos(ref_lat)
    sin_lon = math.sin(ref_lon)
    cos_lon = math.cos(ref_lon)

    e = -sin_lon * dx + cos_lon * dy
    n = -sin_lat * cos_lon * dx - sin_lat * sin_lon * dy + cos_lat * dz
    u =  cos_lat * cos_lon * dx + cos_lat * sin_lon * dy + sin_lat * dz
    return e, n, u
```

3) Odom && TF Publish

- GPS Data 기반 Odom 및 TF 발행
  1. gps_odom_tf 생성
     - cd ~/gps_ws/src/gps_tutorial/gps_tutorial
     - gedit gps_odom_tf .py
  2. setup.py 수정
     - cd ~/gps_ws/src/gps_tutorial
     - gedit setup.py
  3. Build 및 수행
     - cd ~/gps_ws/ && colcon build
     - source install/setup.bash
     - ros2 run gps_tutorial gps_odom_tf