proj_5

November 18, 2021

1 Project 5: Optimal Vehicle State Estimation

Max Schrader

[ ]:
```
from IPython.display import display

import sympy
import numpy as np
import plotly.graph_objects as go
import plotly.io as pio

sympy.init_printing(use_latex="mathjax")
pio.templates.default = "ggplot2"
pio.renderers.default = "notebook+pdf"
```

1.0.1 Importing my code from src/ because I miss VSCode Intellisense

[ ]:
```
from src.helpers import (
    calculate_dubins,
    RAD_2_DEGREE,
    confidence_ellipse,
    normalize_radians,
)
from src.ekf import EKF
from src.base import LTI, Radar
```

1.1 Finding Dubin's Path

[ ]:
```
R_path = 5  # given
q0 = (0, -15, -90)
q1 = (-5, 20, -180)
```

[ ]:
```
optimal_path = calculate_dubins(q0, q1, R_path, 0.01)
```

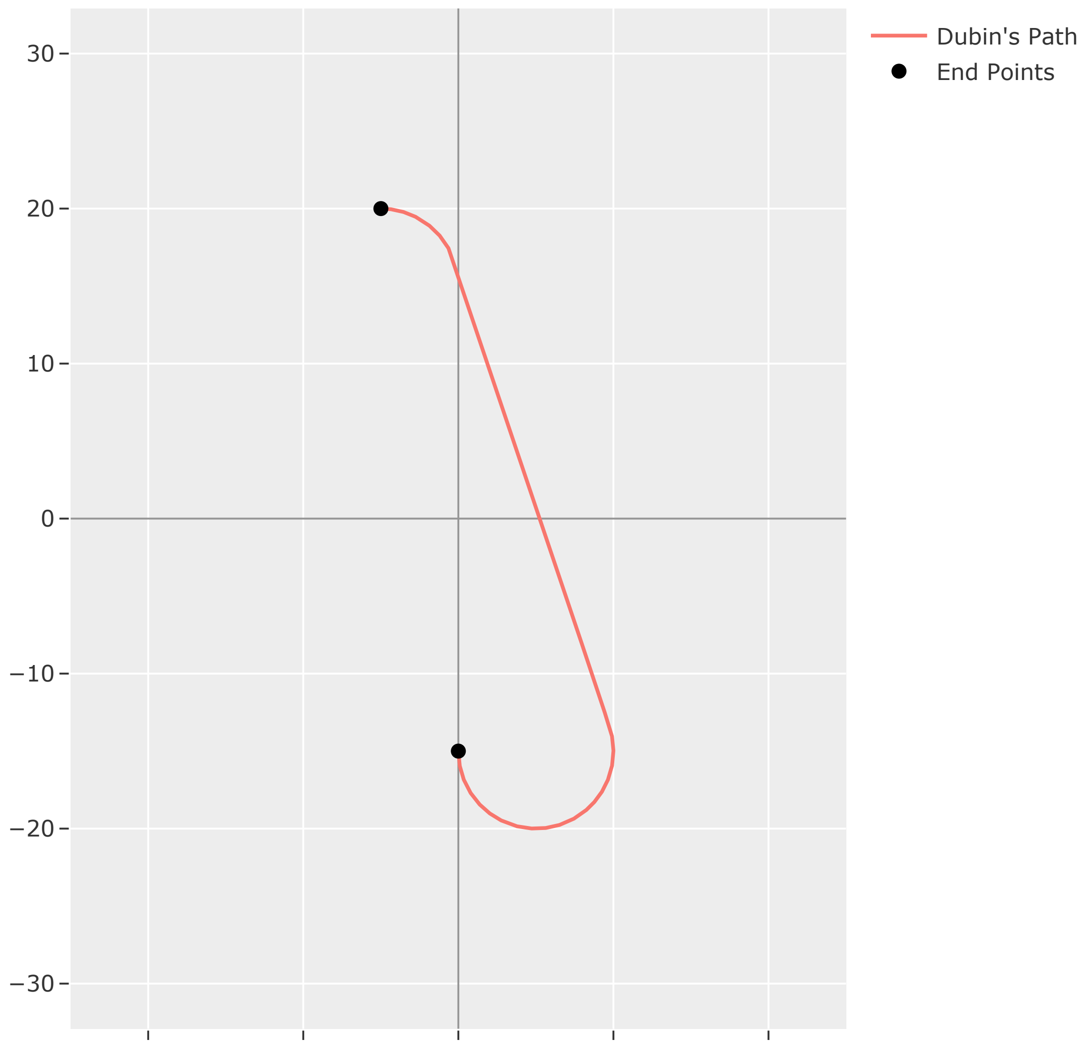
[ ]:
```
len(optimal_path)
```

[ ]: 5519

[ ]:
```python
def plot_dubin(*args):
    fig = go.Figure()

    for (
        name,
        mode,
        path_list,
    ) in args:
        _x = [p[0] for p in path_list]
        _y = [p[1] for p in path_list]
        _theta = [p[2] * RAD_2_DEGREE for p in path_list]

        fig.add_trace(
            go.Scatter(
                x=_x,
                y=_y,
                text=_theta,
                name=name,
                mode=mode,
                hovertemplate="Angle: %{text}<br>X: %{x}<br>Y: %{y}",
            )
        )

        if "dubin" in name.lower():
            fig.add_trace(
                go.Scatter(
                    x=_x[:: len(_x) - 1],
                    y=_y[:: len(_y) - 1],
                    name="End Points",
                    mode="markers",
                    marker_size=8,
                    marker_color="black",
                )
            )

    fig.update_layout(
        margin=dict(l=20, r=20, b=20, t=20),
        height=600,
        width=600,
        yaxis=dict(scaleanchor="x", scaleratio=1),
        xaxis_zeroline=True,
        yaxis_zeroline=True,
        xaxis_zerolinecolor="#969696",
        yaxis_zerolinecolor="#969696",
        xaxis_range=[-25, 25],
        yaxis_range=[-25, 25],
    )
```

```
    return fig
```

[ ]:
```
plot_dubin(["Dubin's Path", "lines", optimal_path]).show()
```



Loading [MathJax]/extensions/MathMenu.js

1.2 Creating the Radars

[ ]:
```
radar_1 = Radar(x=-15, y=-10, v=9)
radar_2 = Radar(x=-15, y=5, v=9)
```

1.3 Creating the Car Model

1.3.1 Symbolicaly

[ ]:
```python
from sympy.abc import alpha, x, y, v, w, R, theta
from sympy import symbols, Matrix
```

[ ]:
```python
d_t, s = symbols("dt, s")
a = Matrix(
    [[x + d_t * s * sympy.cos(theta)], [y + d_t * s * sympy.sin(theta)],␣
 ↪[theta]]
)

A = a.jacobian(Matrix([x, y, theta]))
A
```

[ ]:
$$\begin{bmatrix} 1 & 0 & -dts\sin(\theta) \\ 0 & 1 & dts\cos(\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

[ ]:
```python
y_1, x_1, y_2, x_2 = symbols("y_1, x_1, y_2, x_2")
b = Matrix(
    [[sympy.atan2((y - y_1), (x - x_1))], [sympy.atan2((y - y_2), (x - x_2))],␣
 ↪[theta]]
)
B = b.jacobian(Matrix([x, y, theta]))
B
```

[ ]:
$$\begin{bmatrix} \frac{-y+y_1}{(x-x_1)^2+(y-y_1)^2} & \frac{x-x_1}{(x-x_1)^2+(y-y_1)^2} & 0 \\ \frac{-y+y_2}{(x-x_2)^2+(y-y_2)^2} & \frac{x-x_2}{(x-x_2)^2+(y-y_2)^2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

1.3.2 Create the Car

[ ]:
```python
lti = LTI(
    s=1,
    s_var=0.1,
    dt=0.5,
    x0=optimal_path[0],
    dubins_path=optimal_path,
    q1=q1,
    radar_1=radar_1,
    radar_2=radar_2,
)
```

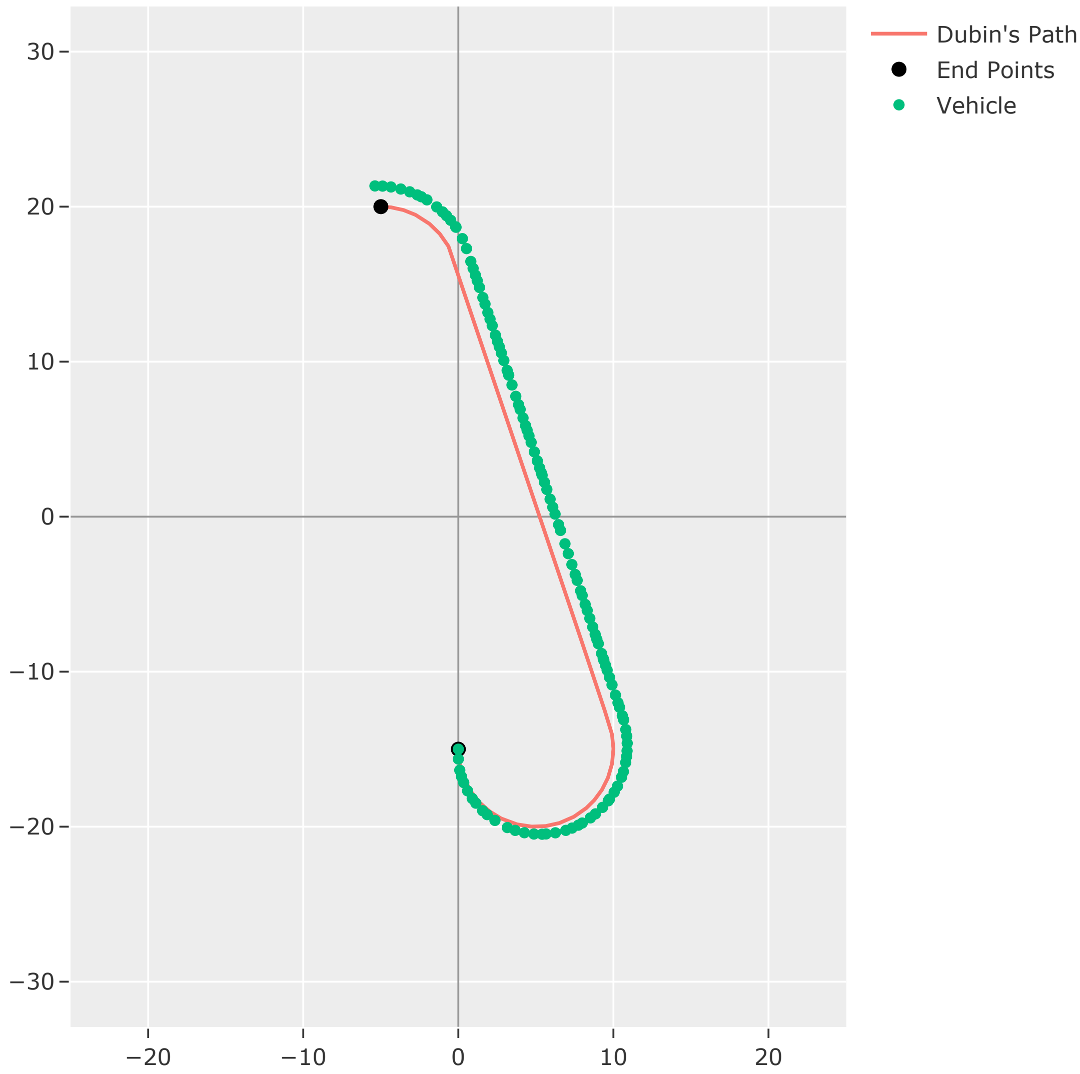
1.3.3 Calculating the True Vehicle Path

[ ]:
```python
lti.x_t_noise(
    x=[np.array(optimal_path[0])],
)
```

1.3.4 Plotting the "True" Vehicle Path

[ ]:
```
simple_trajectory = [list(a) for a in lti.trajectory]
```

[ ]:
```
plot_dubin(
    ["Dubin's Path", "lines", optimal_path], ["Vehicle", "markers",␣
 ↪simple_trajectory]
).show()
```



1.4 Implementing a Simple, Noisy Bearing Calc

[ ]:
```
def two_radar_est(lti: LTI, radar_1: Radar, radar_2: Radar) -> list:
    from scipy.stats import multivariate_normal

    y = []
    f_var = multivariate_normal(
        mean=np.zeros(3),
        cov=np.diag(
            [
                radar_1.v / (RAD_2_DEGREE) ** 2,
                radar_2.v / (RAD_2_DEGREE) ** 2,
                5 / RAD_2_DEGREE,
            ]
        ),
    )
    for x in lti.trajectory:
        var = f_var.rvs()

        state = dict(x=x[0], y=x[1], theta=x[2])

        y_k = lti.measure_fast(**state, noise_matrix=var)

        y_k = np.r_[
            y_k * RAD_2_DEGREE,
            np.linalg.norm(np.array(x[:-1]) - np.array((radar_1.x, radar_1.y))),
            np.linalg.norm(np.array(x[:-1]) - np.array((radar_2.x, radar_2.y))),
        ]
        y.append(y_k)
    return y
```

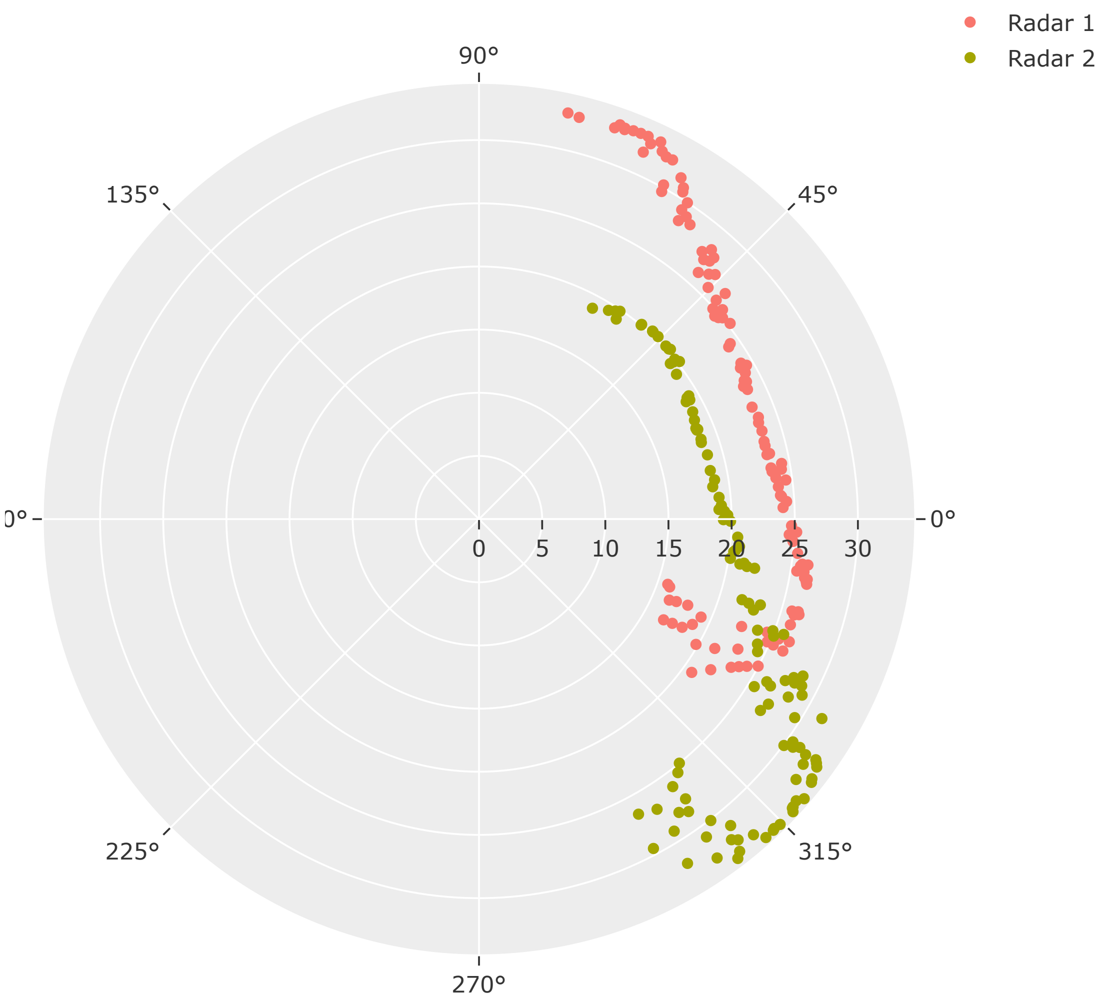
1.4.1 Radars at $(-15, -10)$ and $(-15, 5)$

[ ]:
```
bearing = two_radar_est(lti, radar_1, radar_2)
```

Plotting the Bearing

[ ]:
```
fig = go.Figure()

fig.add_trace(
    go.Scatterpolar(
        r=[b[3] for b in bearing],
        theta=[b[0] for b in bearing],
        mode="markers",
        name="Radar 1",
    )
)
```

```
fig.add_trace(
    go.Scatterpolar(
        r=[b[4] for b in bearing],
        theta=[b[1] for b in bearing],
        mode="markers",
        name="Radar 2",
    )
)

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    height=600,
    width=600,
    yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    xaxis_range=[-25, 25],
    yaxis_range=[-25, 25],
)

fig.show()
```

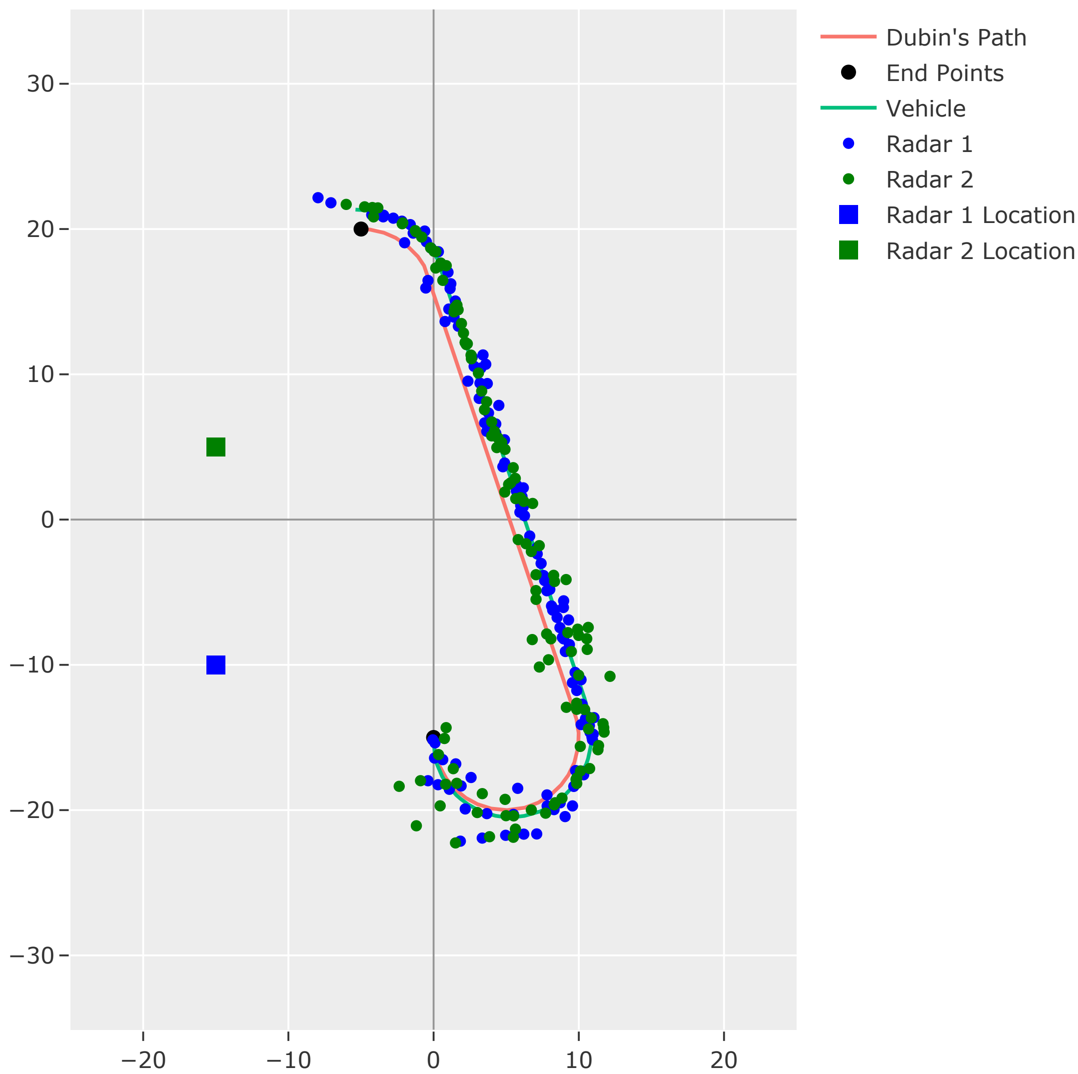


using the bearing and the distance to plot in XY

[ ]:
```
x_y_1_close = [
    (
        radar_1.x + np.cos(b[0] / RAD_2_DEGREE) * b[3],
        radar_1.y + np.sin(b[0] / RAD_2_DEGREE) * b[3],
        b[2],
    )
    for b in bearing
]
x_y_2_close = [
    (
        radar_2.x + np.cos(b[1] / RAD_2_DEGREE) * b[4],
```

```
        radar_2.y + np.sin(b[1] / RAD_2_DEGREE) * b[4],
        b[2],
    )
    for b in bearing
]
```

[ ]:
```
fig = plot_dubin(
    ["Dubin's Path", "lines", optimal_path],
    ["Vehicle", "lines", simple_trajectory],
    ["Radar 1", "markers", x_y_1_close],
    ["Radar 2", "markers", x_y_2_close],
)

fig.data[3].line.color = "blue"
fig.data[4].line.color = "green"

fig.add_trace(
    go.Scatter(
        x=[radar_1.x],
        y=[radar_1.y],
        name="Radar 1 Location",
        mode="markers",
        marker_size=10,
        marker_color="blue",
        marker_symbol="square",
    )
)

fig.add_trace(
    go.Scatter(
        x=[radar_2.x],
        y=[radar_2.y],
        name="Radar 2 Location",
        mode="markers",
        marker_size=10,
        marker_color="green",
        marker_symbol="square",
    )
)


fig.show()
```



1.5 Estimate the Postition of the Ground Vehicle

[ ]:
```
R = np.diag(
    [
        radar_1.v / (RAD_2_DEGREE ** 2),
        radar_2.v / (RAD_2_DEGREE ** 2),
        5 / (RAD_2_DEGREE ** 2),
    ]
)
Q = np.diag([0.1, 0.1, (1 / R_path) ** 2 * 0.5 ** 2])
```

[ ]:
```python
lti.trajectory = [
    np.array(
        (
            x[0],
            x[1],
            normalize_radians(x[2]),
        )
    )
    for x in lti.trajectory
]
```

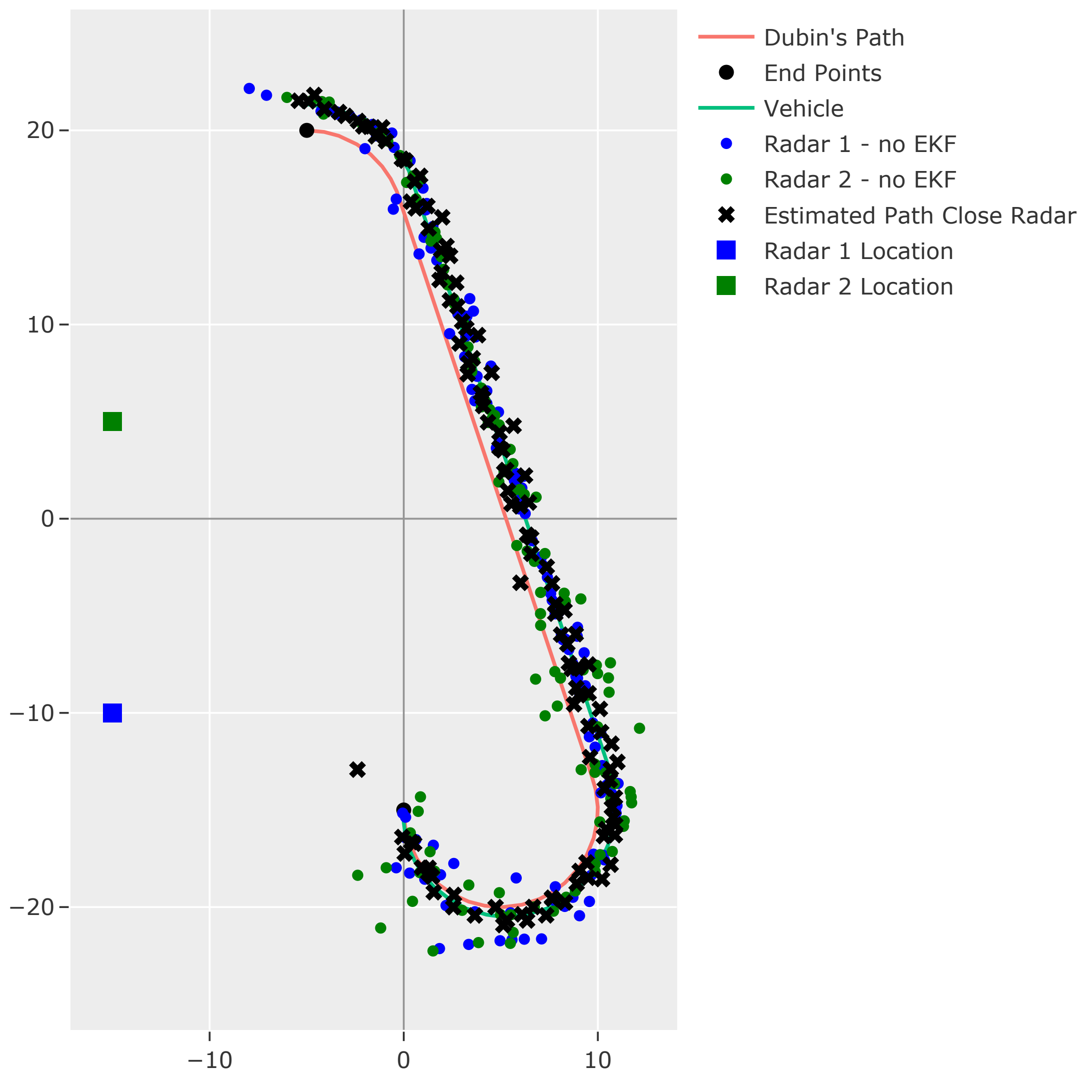
1.5.1 EKF

[ ]:
```python
ekf = EKF(lti, R=R, Q=Q, radars=(radar_1, radar_2))
ekf.run()
```

Plotting Computed Path vs. "Actual" Trajectory

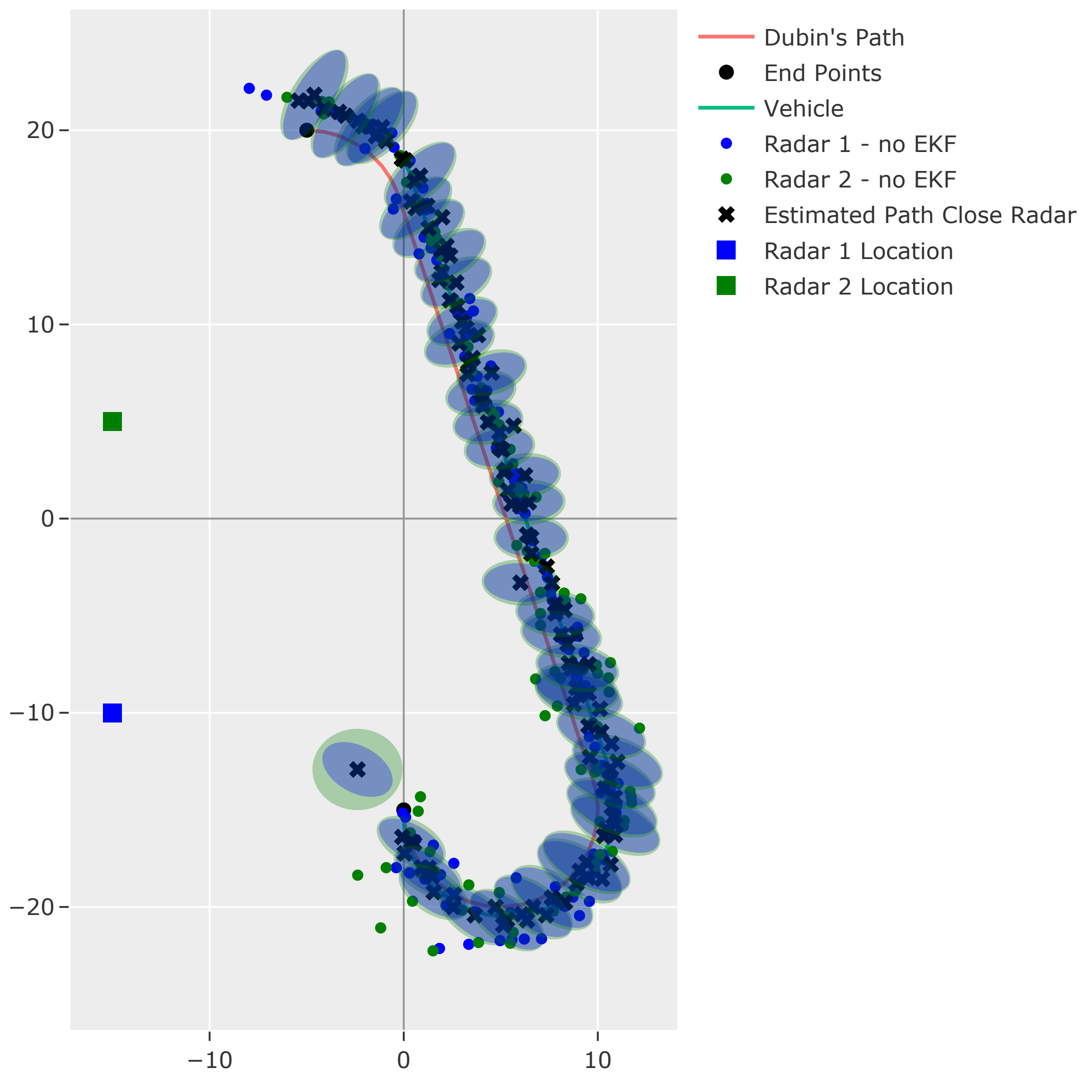
[ ]:
```python
fig = plot_dubin(
    ["Dubin's Path", "lines", optimal_path],
    ["Vehicle", "lines", lti.trajectory],
    ["Radar 1 - no EKF", "markers", x_y_1_close],
    ["Radar 2 - no EKF", "markers", x_y_2_close],
    ["Estimated Path Close Radar", "markers", [r.x_k_k for r in ekf.results]],
    # ["Estimated Path Far Radar", "markers", [r.x_k_k.T[0] for r in res_far]],
)

fig.data[3].line.color = "blue"
fig.data[4].line.color = "green"

fig.add_trace(
    go.Scatter(
        x=[radar_1.x],
        y=[radar_1.y],
        name="Radar 1 Location",
        mode="markers",
        marker_size=10,
        marker_color="blue",
        marker_symbol="square",
    )
)

fig.add_trace(
    go.Scatter(
        x=[radar_2.x],
        y=[radar_2.y],
        name="Radar 2 Location",
        mode="markers",
```

```
        marker_size=10,
        marker_color="green",
        marker_symbol="square",
    )
)

fig.data[-3].marker.symbol = "x"
fig.data[-3].marker.size = 8
fig.data[-3].marker.color = "black"

fig.update_layout(xaxis_range=None, yaxis_range=None)

fig.show()
```



Plotting with select covariance ellipse (every 3)

```
[ ]: for _res in ekf.results[::3]:
         path = confidence_ellipse(
             x=_res.x_k_k[0], y=_res.x_k_k[1], cov=_res.P_k_k[:2, :2], n_std=2,␣
      ↪size=50
         )
         priori_path = confidence_ellipse(
             x=_res.x_k_k[0], y=_res.x_k_k[1], cov=_res.P_k_k_1[:2, :2], n_std=2,␣
      ↪size=50
         )

         fig.add_shape(
             type="path",
             path=priori_path,
             line={"dash": "dot"},
             line_color="green",
             fillcolor="green",
         )

         fig.add_shape(type="path", path=path, line={"dash": "dot"},␣
      ↪fillcolor="blue")


     fig.show()
```



Plotting *a posteriori* states

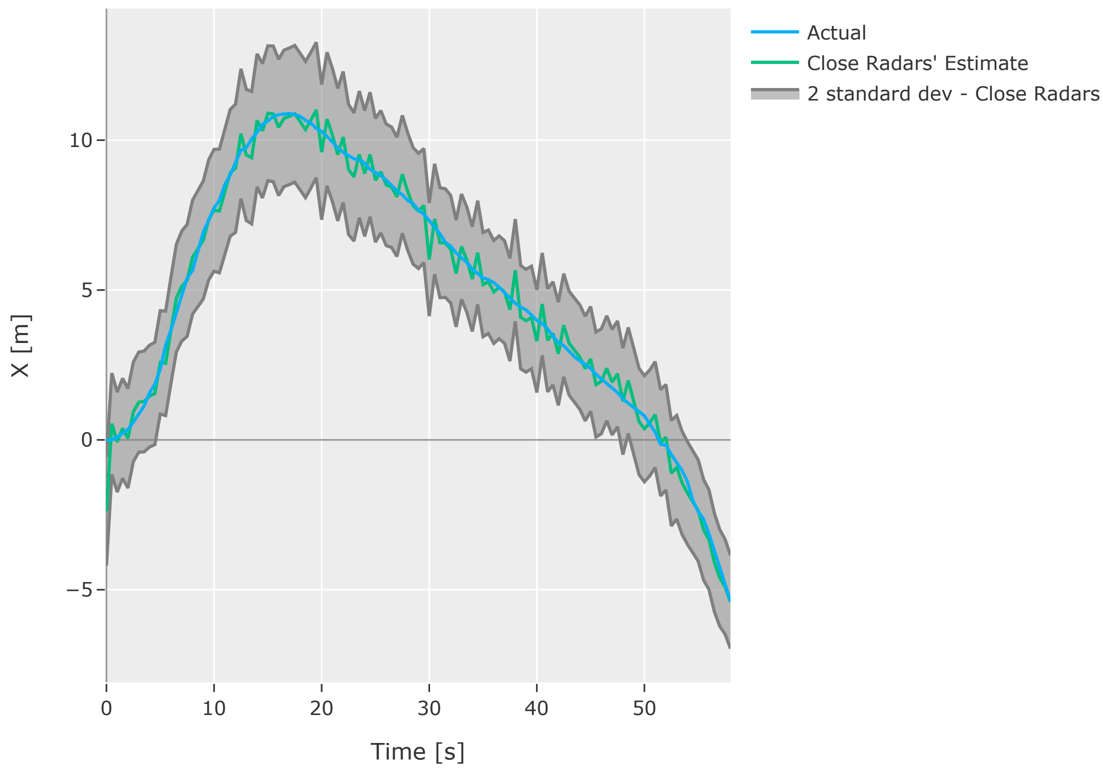
[ ]:
```
res = ekf.results
```

$\hat{x}_{k|k}$

[ ]:
```
t = [i * lti.d_t for i in range(len(res))]
x_k_k_close = [r.x_k_k[0] for r in res]
x_act = [x[0] for x in simple_trajectory]
x_std_close = [r.P_k_k[0, 0] ** (1 / 2) * 2 for r in res]
```

[ ]:
```
fig = go.Figure()
```

```python
fig.add_trace(
    go.Scatter(
        x=t,
        y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='tonexty',
        line_color="grey",
        showlegend=False,
    )
)

fig.add_trace(
    go.Scatter(
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="X [m]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```

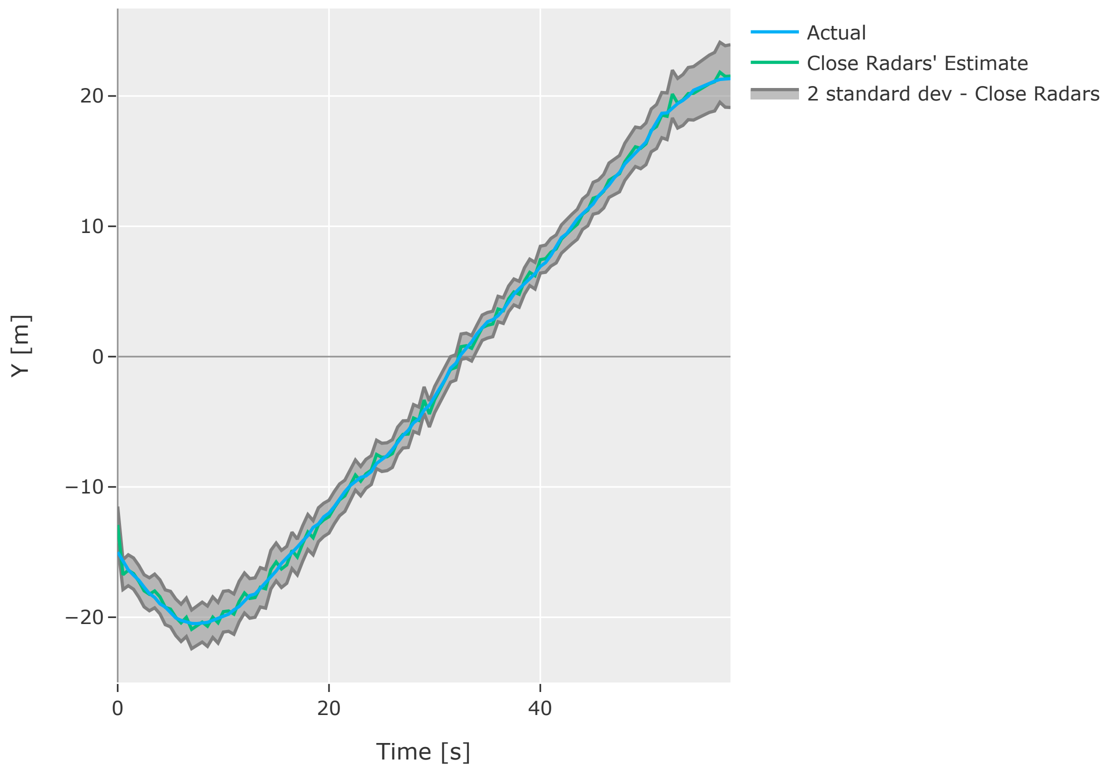


$\hat{y}_{k|k}$

```
[ ]: t = [i * lti.d_t for i in range(len(res))]
     x_k_k_close = [r.x_k_k[1] for r in res]
     x_act = [x[1] for x in simple_trajectory]
     x_std_close = [r.P_k_k[1, 1] ** (1 / 2) * 2 for r in res]
```

```
[ ]: fig = go.Figure()


     fig.add_trace(
         go.Scatter(
             x=t,
             y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
             # fill='tonexty',
             line_color="grey",
             showlegend=False,
         )
     )

     fig.add_trace(
         go.Scatter(
```

```python
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="Y [m]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```

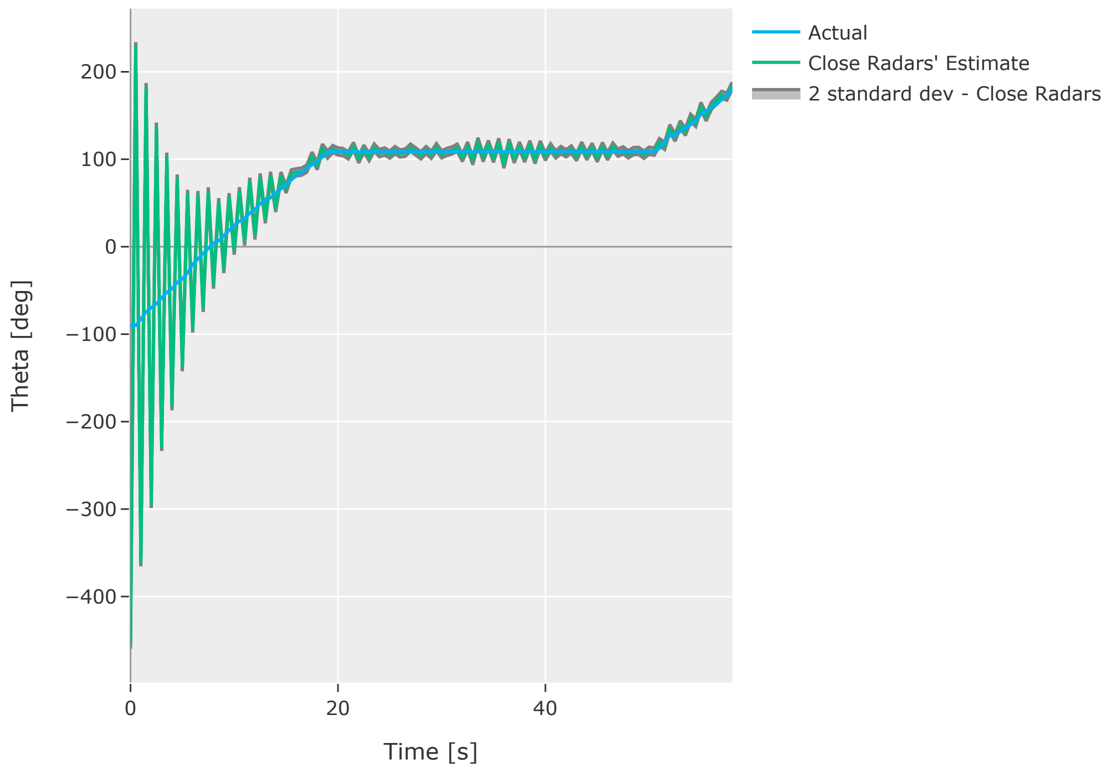


$\hat{\theta}_{k|k}$

```
[ ]: t = [i * lti.d_t for i in range(len(res))]
     x_k_k_close = [r.x_k_k[2] * RAD_2_DEGREE for r in res]
     x_act = [x[2] * RAD_2_DEGREE for x in lti.trajectory]
     x_std_close = [r.P_k_k[2, 2] ** (1 / 2) * 2 * RAD_2_DEGREE for r in res]
```

```
[ ]: fig = go.Figure()


     fig.add_trace(
         go.Scatter(
             x=t,
             y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
             # fill='tonexty',
             line_color="grey",
             showlegend=False,
         )
     )

     fig.add_trace(
         go.Scatter(
```

```
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="Theta [deg]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```



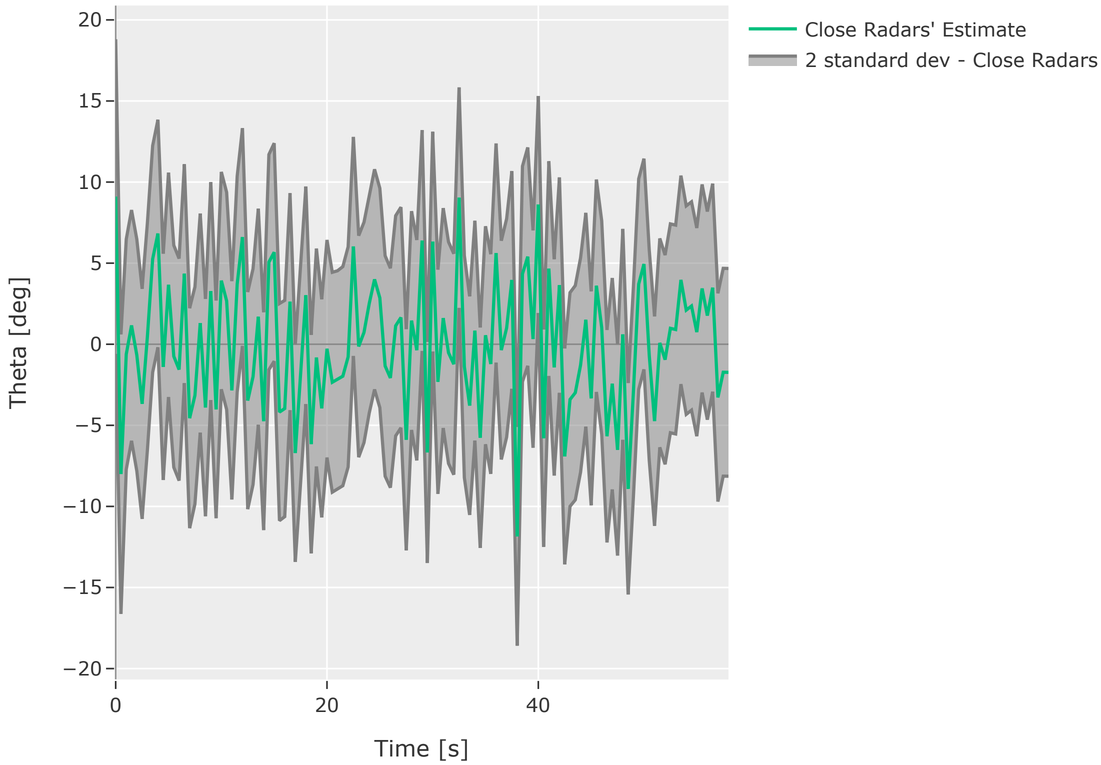
Plotting the Innovation

$\beta_{1,k}$

```
[ ]: t = [i * lti.d_t for i in range(len(res))]
x_k_k_close = [r.y_k[0] * RAD_2_DEGREE for r in res]
x_std_close = [r.S_k[0, 0] ** (1 / 2) * 2 * RAD_2_DEGREE for r in res]
```

```
[ ]: fig = go.Figure()


fig.add_trace(
    go.Scatter(
        x=t,
        y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='tonexty',
        line_color="grey",
        showlegend=False,
    )
)

fig.add_trace(
```

```python
    go.Scatter(
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)

fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="Theta [deg]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```

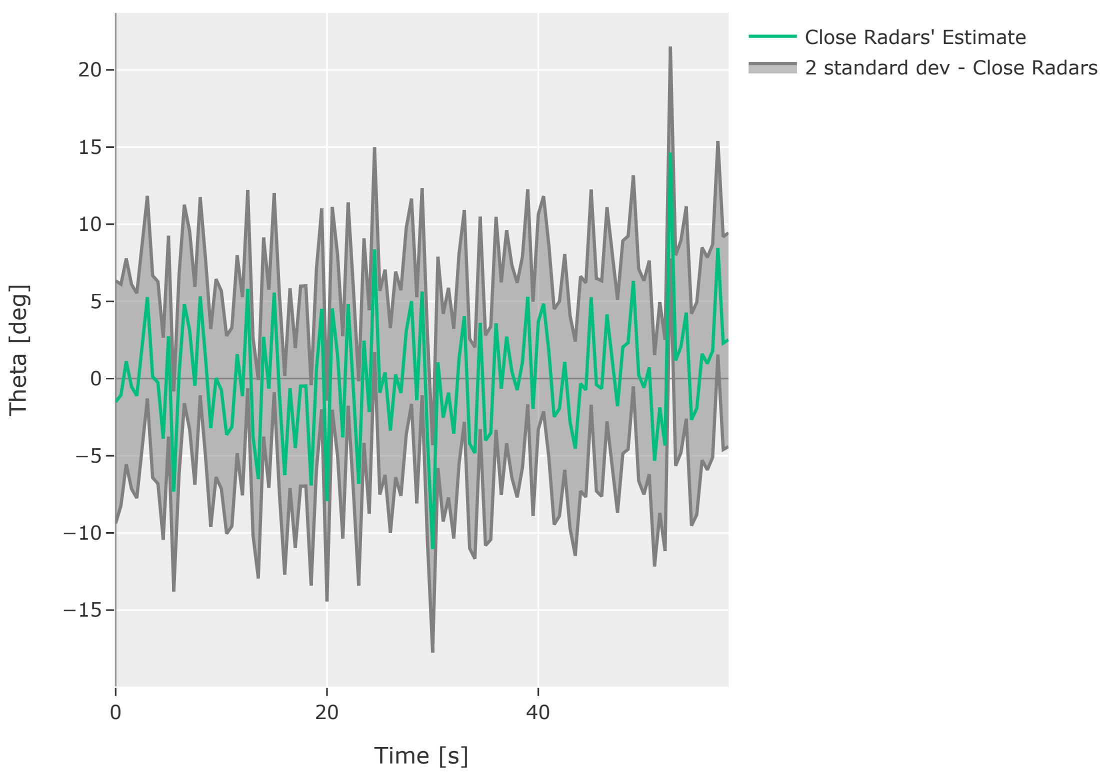


$\beta_{2,k}$

```
[ ]: t = [i * lti.d_t for i in range(len(res))]
     x_k_k_close = [r.y_k[1] * RAD_2_DEGREE for r in res]
     x_std_close = [r.S_k[1, 1] ** (1 / 2) * 2 * RAD_2_DEGREE for r in res]
```

```
[ ]: fig = go.Figure()


     fig.add_trace(
         go.Scatter(
             x=t,
             y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
             # fill='tonexty',
             line_color="grey",
             showlegend=False,
         )
     )

     fig.add_trace(
         go.Scatter(
             x=t,
```

```python
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)

fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="Theta [deg]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```



1.5.2 UKF

[ ]:
```python
from src.ukf import UKF, SigmaPoints
```

[ ]:
```python
sp = SigmaPoints(dim=lti.A.shape[0], alpha=0.5, beta=2, kappa=0.01)

ukf = UKF(
    sigma_obj=sp,
    x0=lti.x0,
    dim_x=lti.A.shape[0],
    dim_y=lti.C.shape[0],
    R=R,
    Q=Q,
    fx=lti.f_fast,
    hx=lti.measure_fast,
)

# ukf.P_priori = np.diag([0.01, 0.01, 0.01])
```

[ ]:
```python
ukf.run(lti)
```

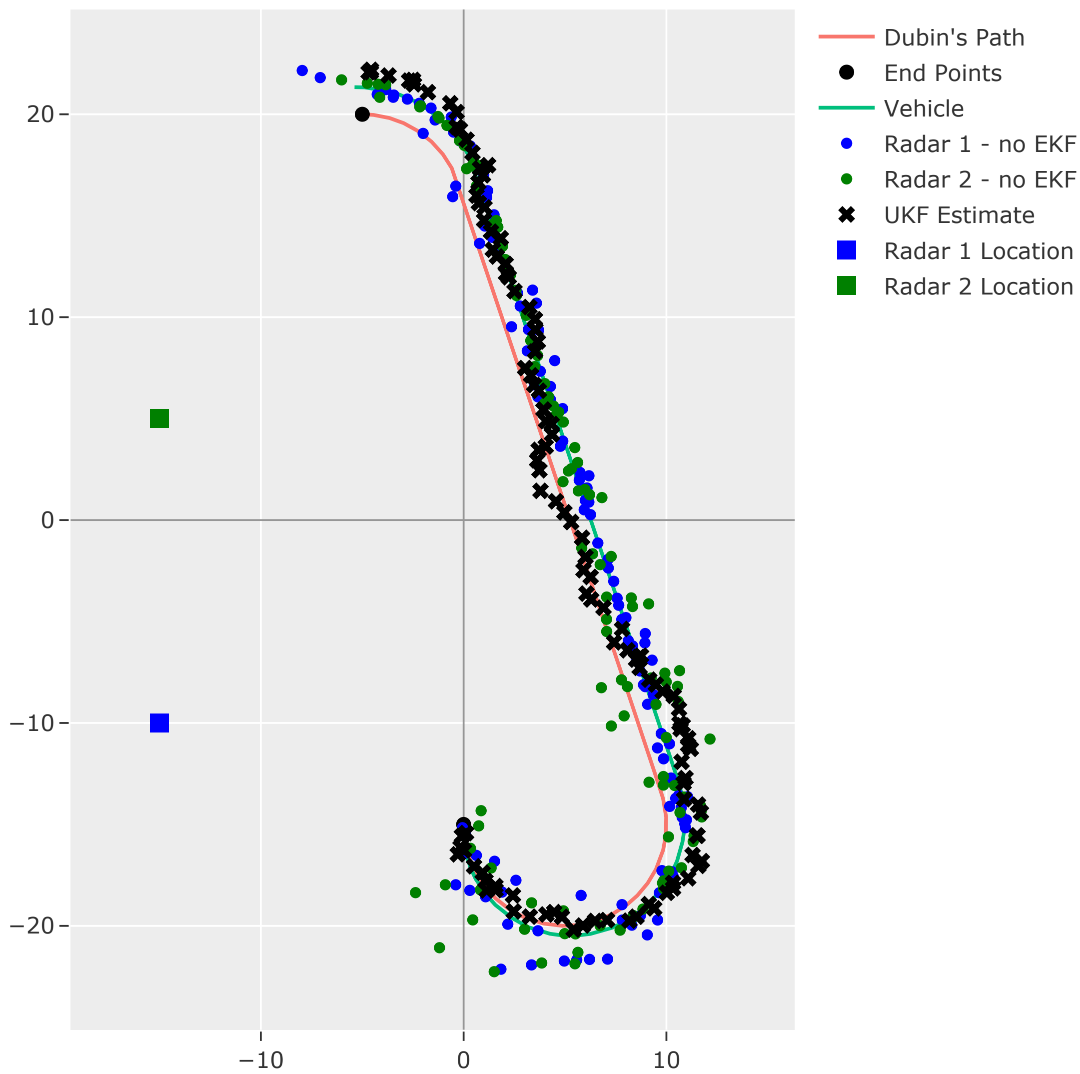
[ ]:
```python
ukf.ss.keys()
```

[ ]:
```
dict_keys(['P_posteriori', 'P_priori', 'x_priori', 'x', 'K', 'z_res',
'z_measure', 'x_posteriori'])
```

Plotting Computed Path vs. "Actual" Trajectory

[ ]:
```python
fig = plot_dubin(
    ["Dubin's Path", "lines", optimal_path],
    ["Vehicle", "lines", simple_trajectory],
    ["Radar 1 - no EKF", "markers", x_y_1_close],
    ["Radar 2 - no EKF", "markers", x_y_2_close],
    ["UKF Estimate", "markers", [r for r in ukf.ss["x"]]],
)

fig.data[3].line.color = "blue"
fig.data[4].line.color = "green"

fig.add_trace(
    go.Scatter(
        x=[radar_1.x],
        y=[radar_1.y],
        name="Radar 1 Location",
        mode="markers",
        marker_size=10,
```

```python
        marker_color="blue",
        marker_symbol="square",
    )
)

fig.add_trace(
    go.Scatter(
        x=[radar_2.x],
        y=[radar_2.y],
        name="Radar 2 Location",
        mode="markers",
        marker_size=10,
        marker_color="green",
        marker_symbol="square",
    )
)

fig.data[-3].marker.symbol = "x"
fig.data[-3].marker.size = 8
fig.data[-3].marker.color = "black"

fig.update_layout(xaxis_range=None, yaxis_range=None)

fig.show()
```



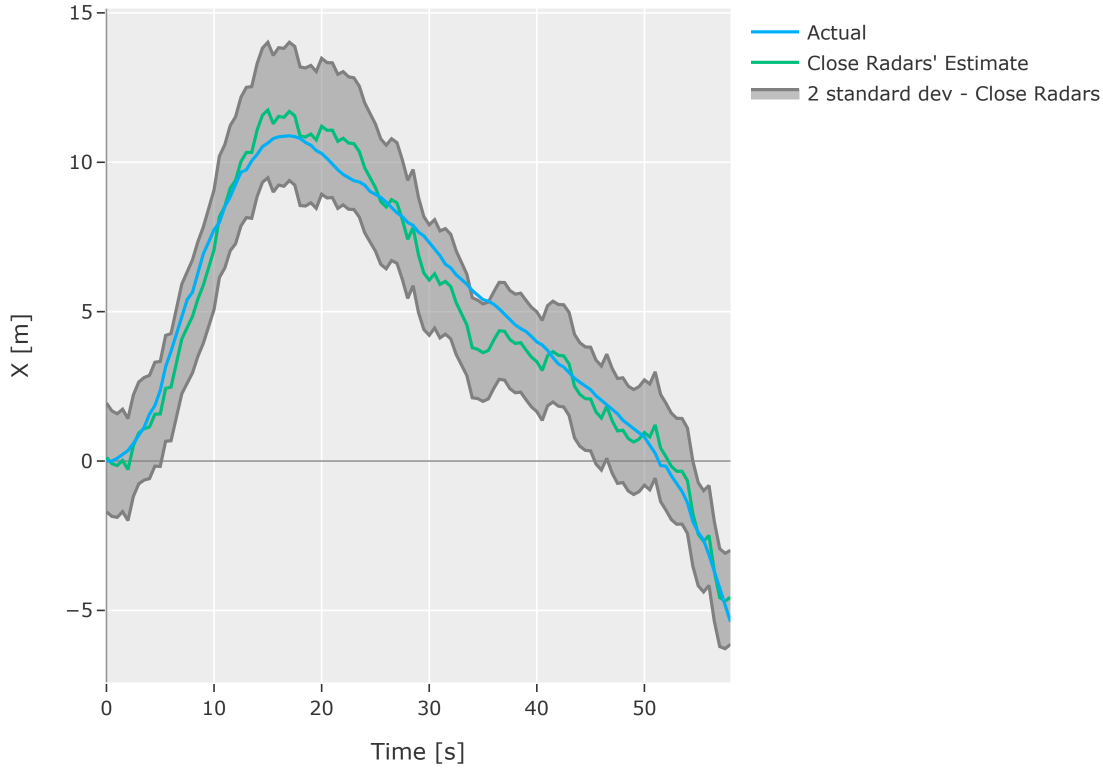
Plotting *a posteriori* states

$\hat{x}_{k|k}$

[ ]:
```
t = [i * lti.d_t for i in range(len(res))]
x_k_k_close = [r[0] for r in ukf.ss["x"]]
x_act = [x[0] for x in lti.trajectory]
x_std_close = [r[0, 0] ** (1 / 2) * 2 for r in ukf.ss["P_posteriori"]]
```

[ ]:
```
fig = go.Figure()


fig.add_trace(
```

```python
    go.Scatter(
        x=t,
        y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='tonexty',
        line_color="grey",
        showlegend=False,
    )
)

fig.add_trace(
    go.Scatter(
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="X [m]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```

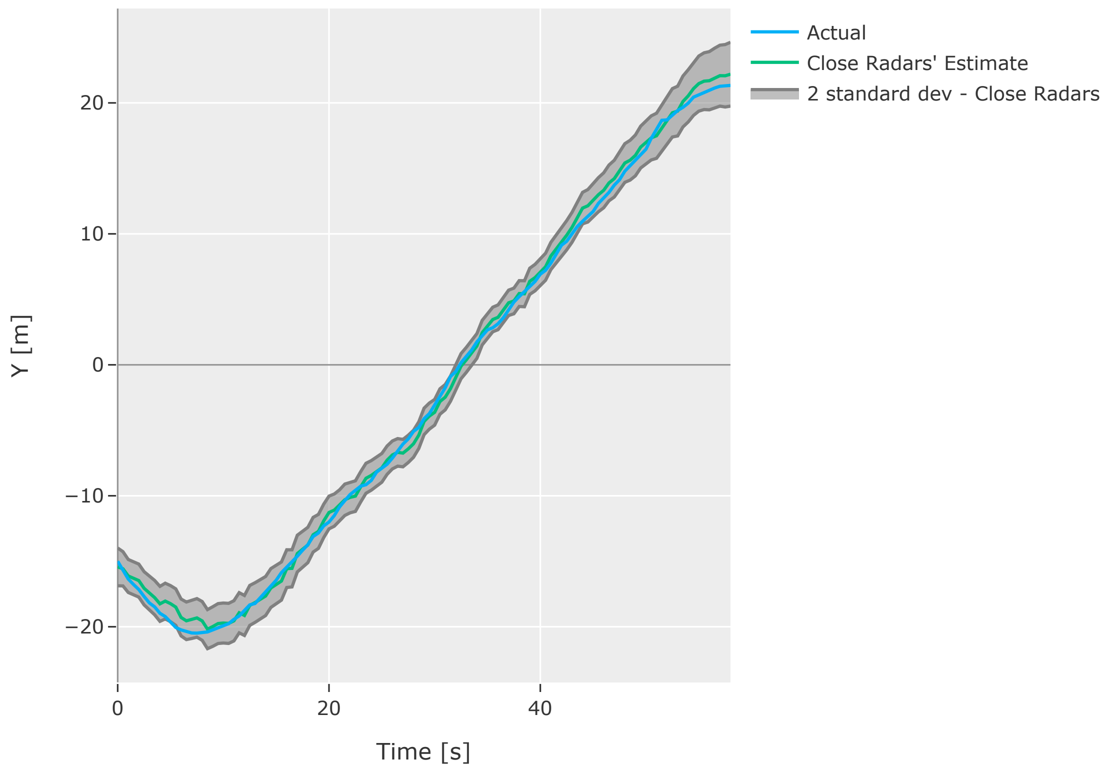


$\hat{y}_{k|k}$

```
[ ]: t = [i * lti.d_t for i in range(len(res))]
     x_k_k_close = [r[1] for r in ukf.ss["x"]]
     x_act = [x[1] for x in lti.trajectory]
     x_std_close = [r[1, 1] ** (1 / 2) * 2 for r in ukf.ss["P_posteriori"]]
```

```
[ ]: fig = go.Figure()


     fig.add_trace(
         go.Scatter(
             x=t,
             y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
             # fill='tonexty',
             line_color="grey",
             showlegend=False,
         )
     )

     fig.add_trace(
         go.Scatter(
```

```
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="Y [m]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```

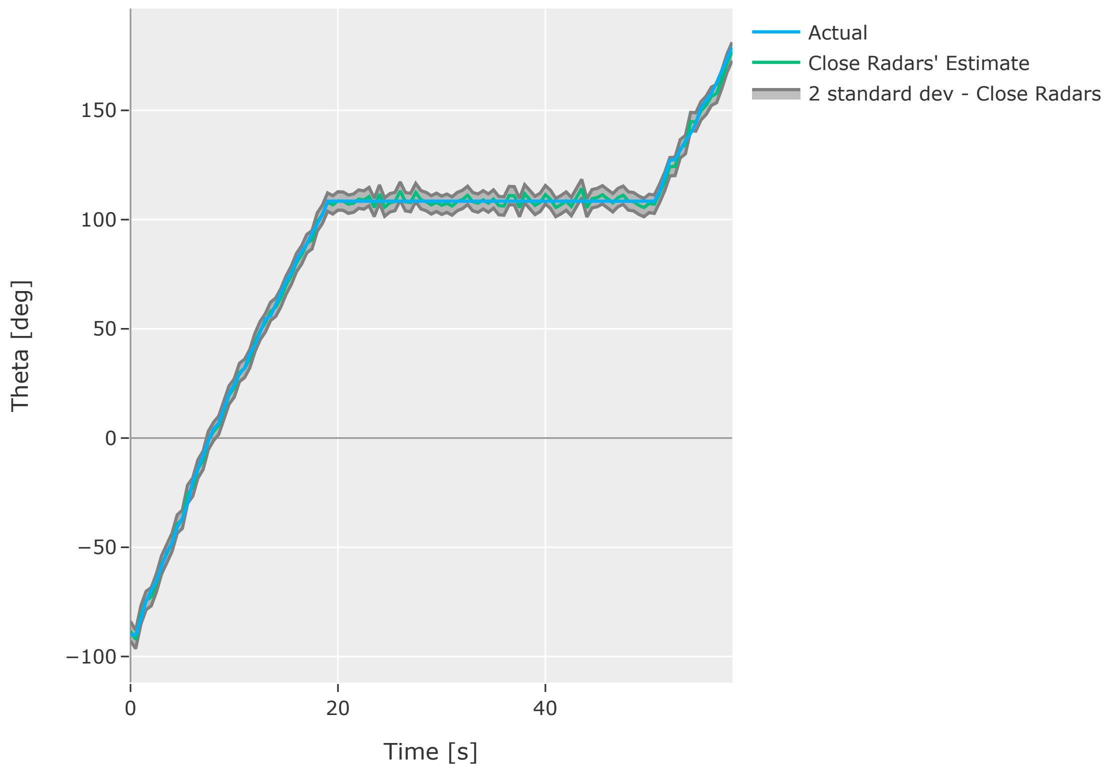


$\hat{\theta}_{k|k}$

```
[ ]: t = [i * lti.d_t for i in range(len(res))]
     x_k_k_close = [r[2] * RAD_2_DEGREE for r in ukf.ss["x"]]
     x_act = [x[2] * RAD_2_DEGREE for x in lti.trajectory]
     x_std_close = [r[2, 2] ** (1 / 2) * 2 * RAD_2_DEGREE for r in ukf.
      ↪ss["P_posteriori"]]
```

```
[ ]: fig = go.Figure()


     fig.add_trace(
         go.Scatter(
             x=t,
             y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
             # fill='tonexty',
             line_color="grey",
             showlegend=False,
         )
     )

     fig.add_trace(
```

```python
    go.Scatter(
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="Theta [deg]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```



1.5.3 PF

[ ]:
```
from src.pf import PF
```

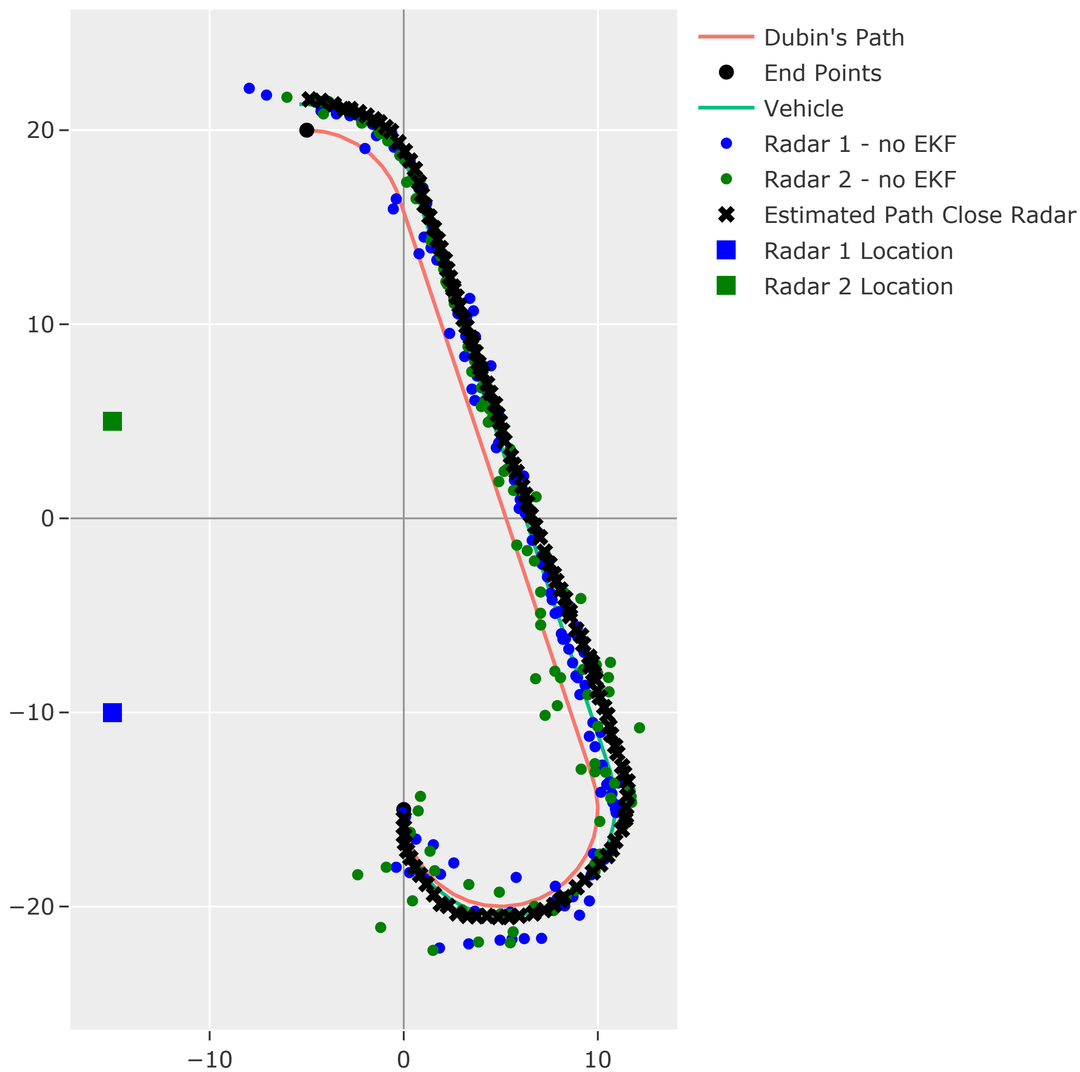
[ ]:
```
pf = PF(
    num_p=1000,
    alpha=0.01,
    x0=lti.x0,
    dim_x=lti.A.shape[0],
    dim_y=lti.C.shape[0],
    R=R,
    Q=Q,
    fx=lti.f_fast,
    hx=lti.measure_fast,
)
```

[ ]:
```
pf.run(lti)
```

Plotting Computed Path vs. "Actual" Trajectory

[ ]:
```
fig = plot_dubin(
    ["Dubin's Path", "lines", optimal_path],
```

```
    ["Vehicle", "lines", lti.trajectory],
    ["Radar 1 - no EKF", "markers", x_y_1_close],
    ["Radar 2 - no EKF", "markers", x_y_2_close],
    ["Estimated Path Close Radar", "markers", [r for r in pf.ss["mean_x"]]],
)

fig.data[3].line.color = "blue"
fig.data[4].line.color = "green"

fig.add_trace(
    go.Scatter(
        x=[radar_1.x],
        y=[radar_1.y],
        name="Radar 1 Location",
        mode="markers",
        marker_size=10,
        marker_color="blue",
        marker_symbol="square",
    )
)

fig.add_trace(
    go.Scatter(
        x=[radar_2.x],
        y=[radar_2.y],
        name="Radar 2 Location",
        mode="markers",
        marker_size=10,
        marker_color="green",
        marker_symbol="square",
    )
)

fig.data[-3].marker.symbol = "x"
fig.data[-3].marker.size = 8
fig.data[-3].marker.color = "black"

fig.update_layout(xaxis_range=None, yaxis_range=None)

fig.show()
```



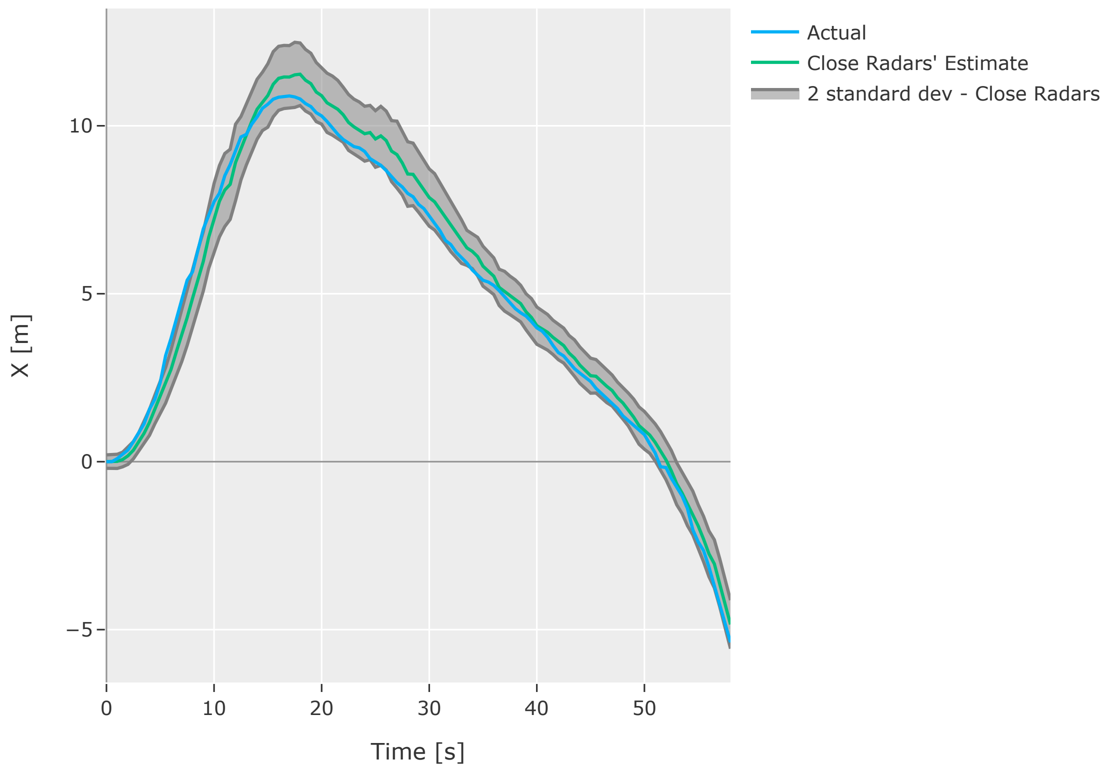
Plotting *a posteriori* states

$\hat{x}_{k|k}$

```
t = [i * lti.d_t for i in range(len(res))]
x_k_k_close = [r[0] for r in pf.ss["mean_x"]]
x_act = [x[0] for x in lti.trajectory]
x_std_close = [abs(r[0]) ** (1 / 2) * 2 for r in pf.ss["mean_var"]]
```

```
fig = go.Figure()

fig.add_trace(
    go.Scatter(
```

```python
        x=t,
        y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='tonexty',
        line_color="grey",
        showlegend=False,
    )
)

fig.add_trace(
    go.Scatter(
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="X [m]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```

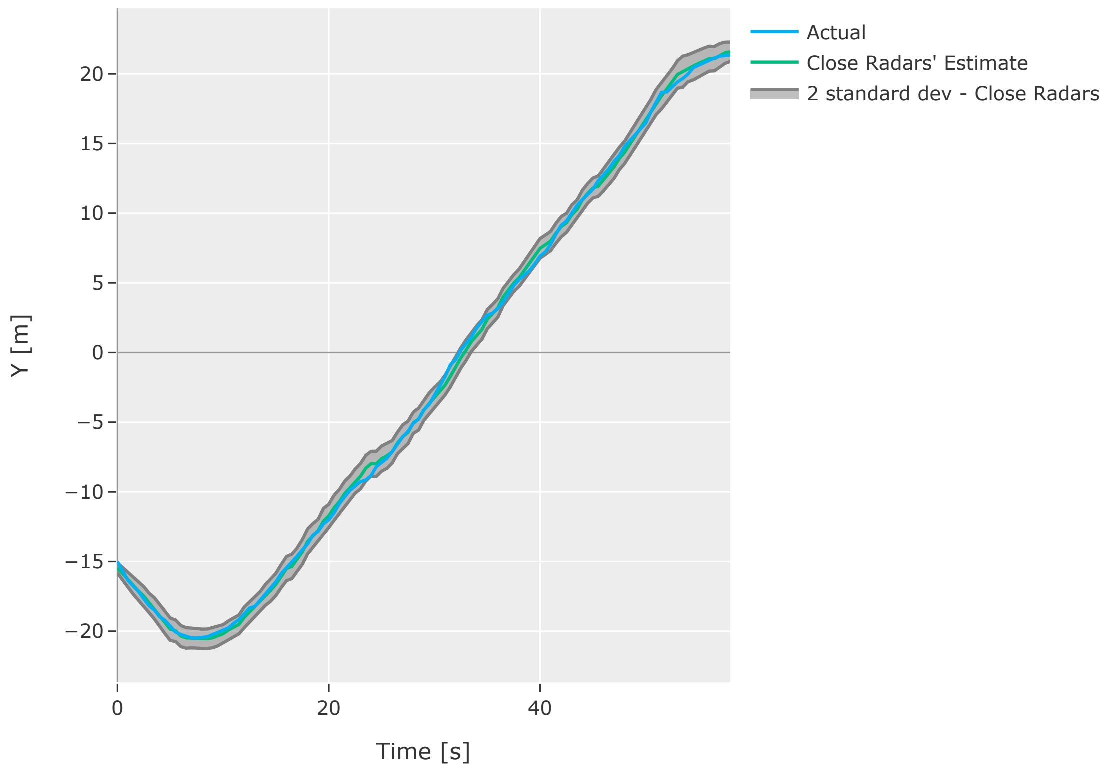


$\hat{y}_{k|k}$

[ ]:
```
t = [i * lti.d_t for i in range(len(res))]
x_k_k_close = [r[1] for r in pf.ss["mean_x"]]
x_act = [x[1] for x in lti.trajectory]
x_std_close = [abs(r[1]) ** (1 / 2) * 2 for r in pf.ss["mean_var"]]
```

[ ]:
```
fig = go.Figure()


fig.add_trace(
    go.Scatter(
        x=t,
        y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='tonexty',
        line_color="grey",
        showlegend=False,
    )
)

fig.add_trace(
    go.Scatter(
```

```python
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="Y [m]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```

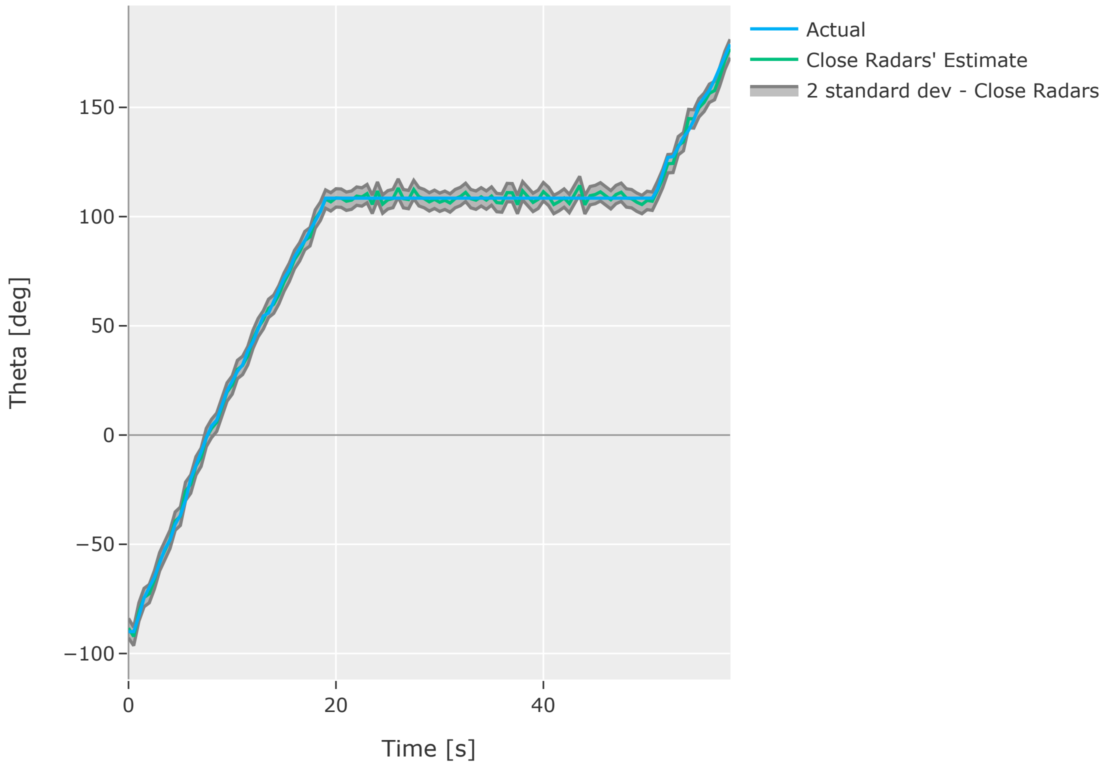


$\hat{\theta}_{k|k}$

[ ]:
```
t = [i * lti.d_t for i in range(len(res))]
x_k_k_close = [r[2] * RAD_2_DEGREE for r in ukf.ss["x"]]
x_act = [x[2] * RAD_2_DEGREE for x in lti.trajectory]
x_std_close = [r[2, 2] ** (1 / 2) * 2 * RAD_2_DEGREE for r in ukf.
 ↪ss["P_posteriori"]]
```

[ ]:
```
fig = go.Figure()


fig.add_trace(
    go.Scatter(
        x=t,
        y=[x + std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='tonexty',
        line_color="grey",
        showlegend=False,
    )
)

fig.add_trace(
```

```
    go.Scatter(
        x=t,
        y=[x - std for x, std in zip(x_k_k_close, x_std_close)],
        # fill='',
        fill="tonexty",
        line_color="grey",
        name="2 standard dev - Close Radars",
    )
)


fig.add_trace(go.Scatter(x=t, y=x_k_k_close, name="Close Radars' Estimate"))

fig.add_trace(go.Scatter(x=t, y=x_act, name="Actual"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="Theta [deg]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```



1.6 Comparing all 3 filters

[ ]:
```
common_args = dict(
    x0=lti.x0,
    dim_x=lti.A.shape[0],
    dim_y=lti.C.shape[0],
    R=R,
    Q=Q,
    fx=lti.f_fast,
    hx=lti.measure_fast,
)
```

1.6.1 Creating the EKF

[ ]:
```
ekf = EKF(lti, R=R, Q=Q, radars=(radar_1, radar_2))
```

1.6.2 Creating the UKF

[ ]:
```
ukf = UKF(
    sigma_obj=SigmaPoints(dim=lti.A.shape[0], alpha=0.5, beta=2, kappa=0.01),
    **common_args
```

```
)
```

1.6.3 Creating the PF

[ ]:
```
pf = PF(num_p=int(1e4), alpha=3, **common_args)
```

1.6.4 Using the same measurement data between the three

[ ]:
```
measurements = [lti.measure_fast(*x, pf.R_func.rvs()) for x in lti.trajectory]
```

1.6.5 Running and Timing the Filters

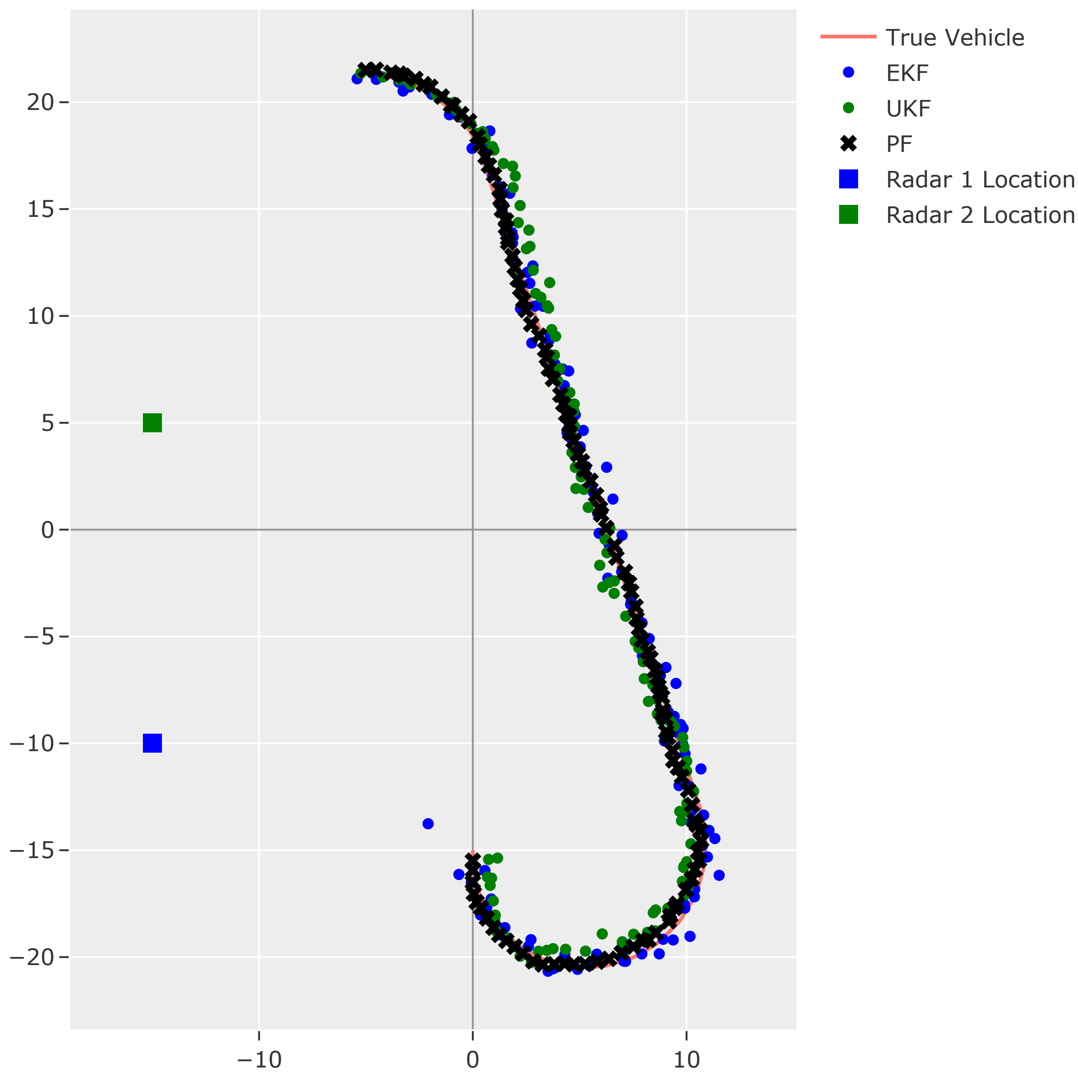
[ ]:
```
import time
```

[ ]:
```
for c in ((ekf, measurements), (ukf, lti, measurements), (pf, lti,␣
 ↪measurements)):
    t0 = time.time()
    c[0].run(c[1])
    print(c[0].__class__.__name__, f"took {time.time() - t0} to run")
```

```
EKF took 3.8233931064605713 to run
UKF took 0.8378481864929199 to run
PF took 82.29645895957947 to run
```

[ ]:
```
fig = plot_dubin(
    ["True Vehicle", "lines", lti.trajectory],
    # ["Radar 1 - no filter", "markers", x_y_1_close],
    # ["Radar 2 - no filter", "markers", x_y_2_close],
    ["EKF", "markers", [r.x_k_k for r in ekf.results]],
    ["UKF", "markers", [r for r in ukf.ss["x"]]],
    ["PF", "markers", [r for r in pf.ss["mean_x"]]],
)

fig.data[1].line.color = "blue"
fig.data[2].line.color = "green"

fig.add_trace(
    go.Scatter(
        x=[radar_1.x],
        y=[radar_1.y],
        name="Radar 1 Location",
        mode="markers",
        marker_size=10,
        marker_color="blue",
        marker_symbol="square",
    )
)
```

```python
fig.add_trace(
    go.Scatter(
        x=[radar_2.x],
        y=[radar_2.y],
        name="Radar 2 Location",
        mode="markers",
        marker_size=10,
        marker_color="green",
        marker_symbol="square",
    )
)

fig.data[-3].marker.symbol = "x"
fig.data[-3].marker.size = 8
fig.data[-3].marker.color = "black"

fig.update_layout(xaxis_range=None, yaxis_range=None)

fig.show()
```

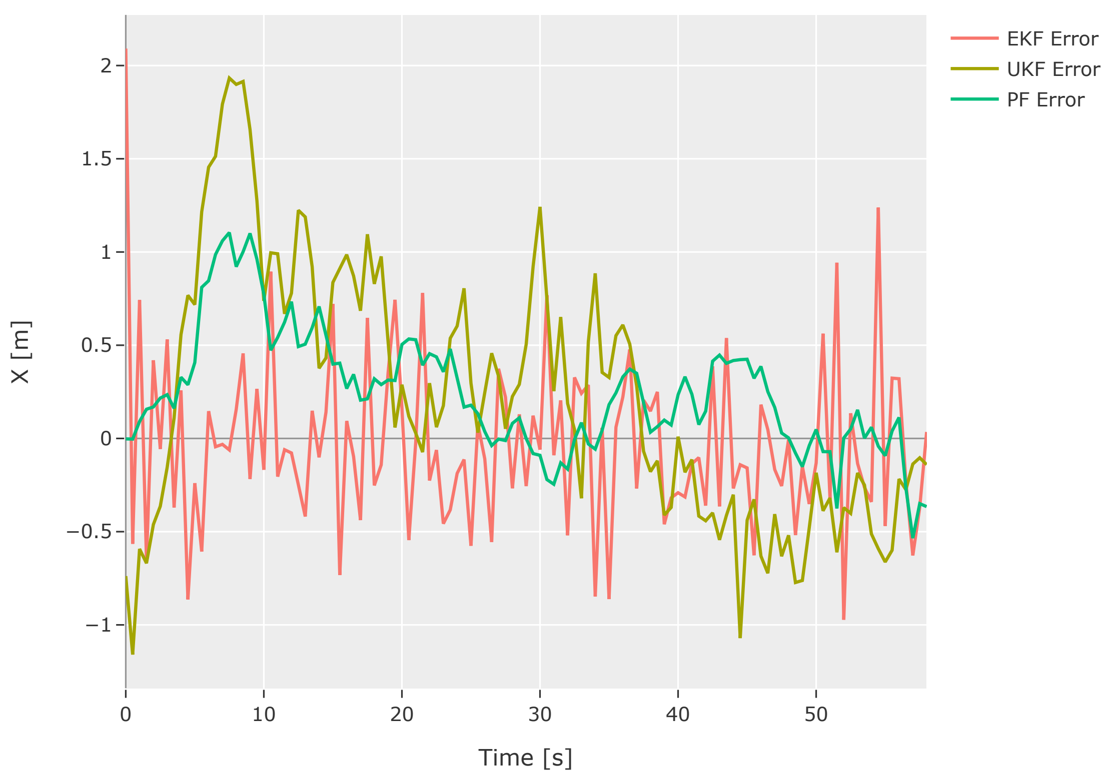


1.6.6 Plotting the Error of the Algorithims

[ ]:
```
errors = []
for x_est in [[res.x_k_k for res in ekf.results], ukf.ss["x"], pf.ss["mean_x"]]:
    errors.append(
        [
            [x[i] - _x_est[i] for x, _x_est in zip(lti.trajectory, x_est)]
            for i in range(3)
        ]
    )
```

1.6.7 Plotting X Error

[ ]:
```
fig = go.Figure()

fig.add_trace(go.Scatter(x=t, y=errors[0][0], name="EKF Error"))

fig.add_trace(go.Scatter(x=t, y=errors[1][0], name="UKF Error"))

fig.add_trace(go.Scatter(x=t, y=errors[2][0], name="PF Error"))

fig.update_layout(
    margin=dict(l=20, r=20, b=20, t=20),
    # height=600, width=600,
    # yaxis=dict(scaleanchor="x", scaleratio=1),
    xaxis_zeroline=True,
    yaxis_zeroline=True,
    xaxis_zerolinecolor="#969696",
    yaxis_zerolinecolor="#969696",
    yaxis_title="X [m]",
    xaxis_title="Time [s]"
    # xaxis_range=[-25, 25],
    # yaxis_range=[-25, 25],
)
```



1.7 Comment on the plots and the differences between the filters

The best filter in terms of minimizing tracking error is the PF, albeit marginally. There is however a tradeoff, as the PF takes ~**105x** longer to run than the UKF (though none of my implementations are optimized for speed). As it is currently configured, the PF would not be able to run in real-time. That being said, there is likely hyperparameter tuning that could be done to both the UKF and PF, which may increase the accuracy of the UKF and the speed of the PF (an easy one being reducing the # of particles).

Its interesting that you can acheive higher accuracy with the PF and the UKF, even though they have less information about the system. The angles in this system introduce the non-linearities, which decreases the accuracy of the EKF as it relies on linearization about a point.

I'm not completely happy with the filtering abilities of my implementation. I'm not sure if there is a small implementation error from my side or if the "made up" noise is high in relation to the model. It seems like radars should perform better than this…