class FEKFSLAMFeature(MapFeature):

This class extends the `MapFeature` class to implement the Feature EKF SLAM algorithm.
The `MapFeature` class is a base class providing support to localize the robot using a map of point features.
The main difference between FEKMBL and FEAKFSLAM is that the former uses the robot pose as a state variable,
while the latter uses the robot pose and the feature map as state variables. This means that few methods provided by
class need to be overridden to gather the information from state vector instead that from the deterministic map.

def hfj(self, xk_bar, Fj):

...

This method implements the direct observation model for a single feature observation z_{f_i} , so it implements its related
observation function (see eq. `eq-FEKFSLAM-hfj`). For a single feature observation z_{f_i} of the feature ${}^N x_{F_j}$ the method computes its
expected observation from the current robot pose ${}^N x_B$.
This function uses a generic implementation through the following equation:

$$z_{f_i} = h_{Fj}(x_k, v_k) = s2o(\ominus^N x_B \boxplus^N x_{F_j}) + v_{f i_k}$$

Where ${}^N x_B$ is the robot pose and ${}^N x_{F_j}$ are both included within the state vector:

$$x_k = [{}^N x_B^T \ \cdots \ {}^N x_{F_j} \ \cdots \ {}^N x_{F_{nf}}]^T$$

and `s2o` is a conversion function from the store representation to the observation representation.

The method is called by `FEKFSLAM.hf` to compute the expected observation for each feature observation contained in the observation vector $z_f = [z_{f_1}^T \ \cdots \ z_{f_i}^T \ \cdots \ z_{f_{nzf}}^T]^T$.

- **Parameters**:
  - `xk_bar` : mean of the predicted state vector

- `Fj` : map index of the observed feature.
- **Returns**:
  - expected observation of the feature ${}^N x_{F_j}$

```python
def hfj(self, xk_bar, Fj):
        ## To be completed by the student
        NxB = xk_bar[0:3].reshape(3,1)
        NxFj = xk_bar[3+2*Fj:3+2*Fj+2]
        print("xbar: ", xk_bar)
        print("Index", 3+2*Fj, ":", 3+2*Fj+2)
        NxFj = CartesianFeature(NxFj.reshape(2,1))
        NxB_inv = Pose3D(NxB).ominus()
        print(NxFj.shape)
        zFi = self.s2o(NxFj.boxplus(NxB_inv))
        return zFi
```

def Jhfjx():

....

Jacobian of the single feature direct observation model `hfj` (eq. `eq-FEKFSLAM-hfj`) with respect to the state vector $\bar{x}_k$:

$$x_k = \left[{}^N x_B^T \ \cdots \ {}^N x_{F_j} \ \cdots \ {}^N x_{F_{nf}}\right]^T$$

$$J_{hfjx} = \frac{\partial h_{f_{zfi}}(x_k, v_k)}{\partial x_k} = \frac{\partial s2o(\ominus^N x_B \boxplus^N x_{F_j}) + v_{fi_k}}{\partial x_k}$$

$$👉 \left[\frac{\partial h_{F_j}(x_k,v_k)}{\partial^N x_{B_k}} \quad \frac{\partial h_{F_j}(x_k,v_k)}{\partial^N x_{F_1}} \quad \cdots \quad \frac{\partial h_{F_j}(x_k,v_k)}{\partial^N x_{F_j}} \quad \cdots \quad \frac{\partial h_{F_j}(x_k,v_k)}{\partial^N x_{F_n}}\right]$$

$$👉 \left[J_{s2o} J_{1\boxplus} J_{\ominus} \quad 0 \quad \cdots \quad J_{s2o} J_{2\boxplus} \quad \cdots \quad 0\right]$$

where we have used the abbreviation:

$$J_{s2o} \equiv J_{s2o}(\ominus^N x_B \boxplus^N x_{F_j})$$

$$J_{1\boxplus} \equiv J_{1\boxplus}(\ominus^N x_B, {}^N x_{F_j})$$

$$J_{2\boxplus} \equiv J_{2\boxplus}(\ominus^N x_B, {}^N x_{F_j})$$

- **Parameters**:

- `xk` : state vector mean
- `Fj` : map index of the observed feature

- **Returns**:
  - Jacobian matrix defined in eq. `eq-Jhfjx`

```
## !To be completed by the student
NxB = xk[0:3].reshape(3,1)
NxFj = xk[3+2*Fj:3+2*Fj+2]

NxFj = CartesianFeature(NxFj.reshape(2,1))

NxB_inv = Pose3D(NxB).ominus()

Js2o = self.J_s2o(NxFj.boxplus(NxB_inv))
J1boxplus = NxFj.J_1boxplus(NxB_inv)
J2boxplus = NxFj.J_2boxplus(NxB_inv)



jhfjx = np.zeros((2, len(xk)))
jhfjx[0:2, 0:3] = Js2o @ J1boxplus @ NxB_inv.J_ominus()
print("index: ", 3+2*Fj)
jhfjx[0:2, 3+2*Fj:3+2*Fj+2] = Js2o @ J2boxplus

return jhfjx
```

Prev Jacobian jhfix

Computes the Jacobian of the feature observation function `hf` (eq. `eq-hf`), with respect to the state vector $\bar{x}_k$:

$$J_{hfx} = \frac{\partial h_f(x_k, v_k)}{\partial x_k} = \begin{bmatrix} \frac{\partial h_{F_a}(x_k, v_k)}{\partial x_k} \\ \vdots \\ \frac{\partial h_{F_b}(x_k, v_k)}{\partial x_k} \\ \vdots \\ \frac{\partial h_{F_c}(x_k, v_k)}{\partial x_k} \end{bmatrix} = \begin{bmatrix} J_{h_{F_a}} \\ \vdots \\ J_{h_{F_b}} \\ \vdots \\ J_{h_{F_c}} \end{bmatrix}$$

where $J_{h_{F_j}}$ is the Jacobian of the observation function `hfj` (eq. `eq-Jhfjx`) for the feature F_j and observation z_{f_i}.

To do it, given a vector of observations $z_f = \begin{bmatrix} z_{f_1} & \cdots & z_{f_i} & \cdots & z_{f_{n_{zf}}} \end{bmatrix}$ this method iterates over each feature observation z_{f_i} calling the method `Jhfj` to compute
the Jacobian of the observation function for each feature observation (J_{hfj}), collecting all them in the returned Jacobian matrix J_{hfx}.

- **Parameters**:
  - `xk` : state vector mean $\hat{x}_k$.
- **Returns**:
  - Jacobian of the observation function `hf` with respect to the robot pose $J_{hfx} = \frac{\partial h_f(\bar{x}_k, v_{f_k})}{\bar{x}_k}$

```
# TODO: To be implemented by the student
xB_dim = np.shape(xk)[0]

J = self.Jhfjx(xk, 0)

# Number of feature loop
for i in range(1,self.nf):
    J = np.block([[J],[self.Jhfjx(xk, i)]])
return J
```

Data association algorithm.
Given state vector (x_k and P_k) including the robot pose and a set of feature observations z_f and its covariance matrices R_f, the algorithm computes the expected feature observations h_f and its covariance matrices P_f. Then it calls an association algorithms like ICNN (JCBB, etc.) to build a pairing hypothesis associating the observed features z_f
with the expected features observations h_f.

The vector of association hypothesis H is stored in the H attribute and its dimension is the number of observed features within z_f. Given the j^{th} feature observation z_{f_j}, *self.H[j]=i*
means that z_{f_j} has been associated with the i^{th} feature . If *self.H[j]=None* means that z_{f_j}
has not been associated either because it is a new observed feature or because it is an outlier.

Parameters:

- x_k: mean state vector including the robot pose
- P_k: covariance matrix of the state vector
- z_f: vector of feature observations
- R_f: Covariance matrix of the feature observations

Returns:

- The vector of association hypothesis H

```
# TODO: To be completed by the student

hF = []
PF = []
xF = xk[self.xBpose_dim:]

for i in range(0, len(xF), self.xF_dim):
    hF_i = self.hfj(xk, i)
    PF_i = self.Jhfjx(xk, i) @ Pk @ self.Jhfjx(xk, i).T

    hF.append(hF_i)
    PF.append(PF_i)
H = self.ICNN(hF, PF, zf, Rf)
self.H = H
return H
```

GetFeatures

Reads the Features observations from the sensors. For all features within the field of view of the sensor, the method returns the list of robot-related poses and the covariance of their corresponding observation noise in **2D Cartesian coordinates**.

Returns zk, Rk: list of Cartesian features observations in the B-Frame and the covariance of their corresponding observation noise:

- $z_k = [^B x_{F_i}^T \cdots ^B x_{F_j}^T \cdots ^B x_{F_k}^T]^T$
- $R_k = block\_diag([R_{F_i} \cdots R_{F_j} \cdots R_{F_k}])$

```python
# TODO: To be implemented by the student

zf, Rf = self.robot.ReadFeatureCartesian2D()
Hf = []
Vf = []
# Raise flag got feature
if len(zf) != 0:
    self.featureData = True
return zf, Rf, Hf, Vf
```

`def ICNN(hf, Phf, zf, Rf)` :

Individual Compatibility Nearest Neighbor (ICNN) data association algorithm. Given a set of expected feature
observations h_f and a set of feature observations z_f, the algorithm returns a pairing hypothesis H that associates each feature observation z_{f_i} with the expected feature observation h_{f_j} that minimizes the Mahalanobis distance D^2_{ij}.

Parameters:

- hf: vector of expected feature observations
- Phf: Covariance matrix of the expected feature observations
- zf: vector of feature observations
- Rf: Covariance matrix of the feature observations
- dim: feature dimensionality

Returns:

- The vector of association hypothesis H

```
# TODO: To be completed by the student

Hp = []
for j in range(len(zf)):
    nearest = 0
    D2_min = np.inf
    for i in range(len(hf)):
        D2_ij = self.SquaredMahalanobisDistance(hf[i], Phf[i], zf[j], Rf[j])
        if self.IndividualCompatibility(D2_ij, self.xF_dim, self.alpha) and D2_ij < D2_
            nearest = i+1
            D2_min = D2_ij
    Hp.append(nearest)
return Hp
```