AA 274A: Principles of Robot Autonomy I
Problem Set 4

Name: Li Quan Khoo
SUID: lqkhoo (06154100)

November 2, 2020

Problem 1: EKF Localization

(i) (code)

(ii) (code)

(iii) (code)

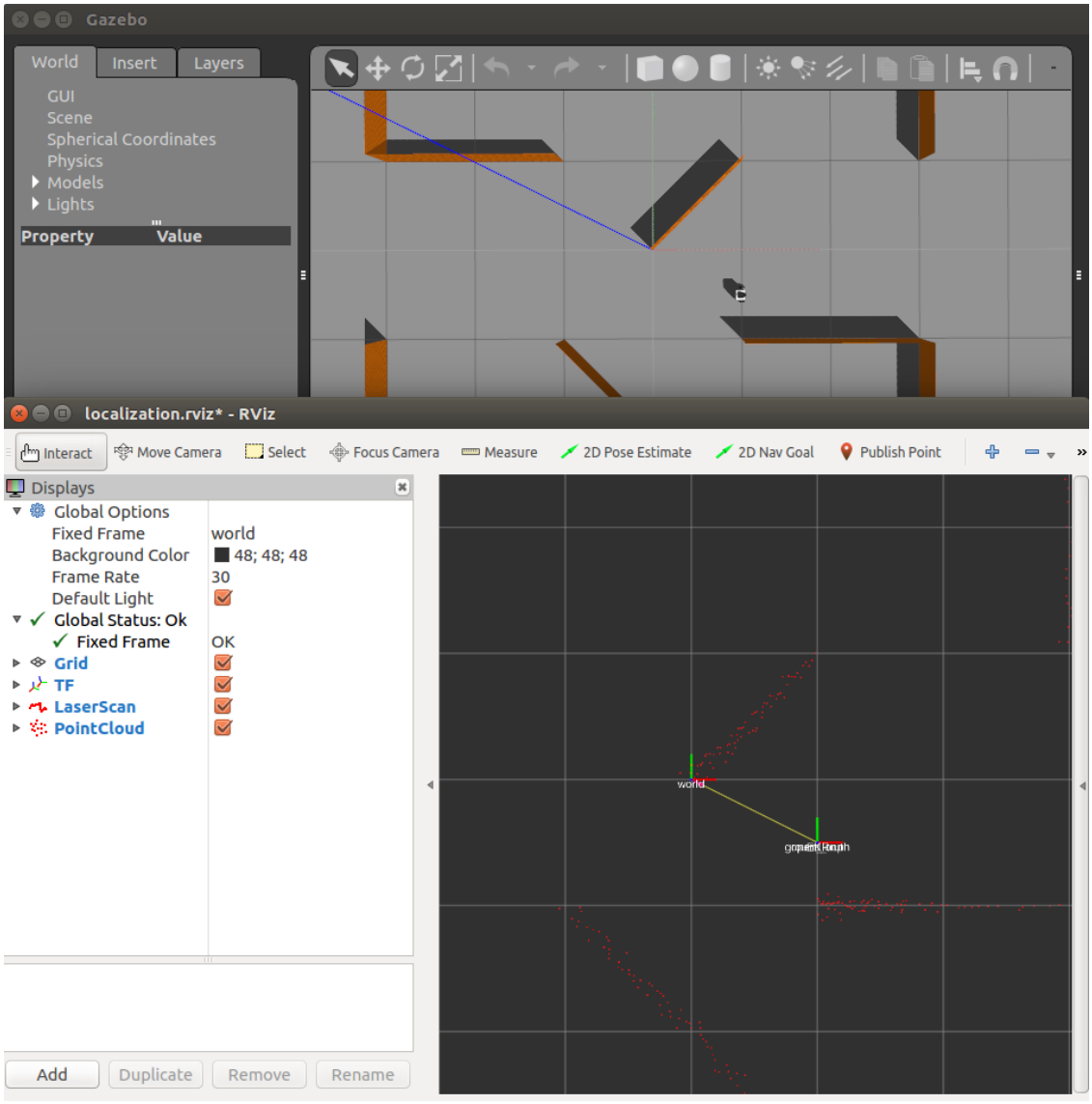
(iv) (code)

(v) (code)

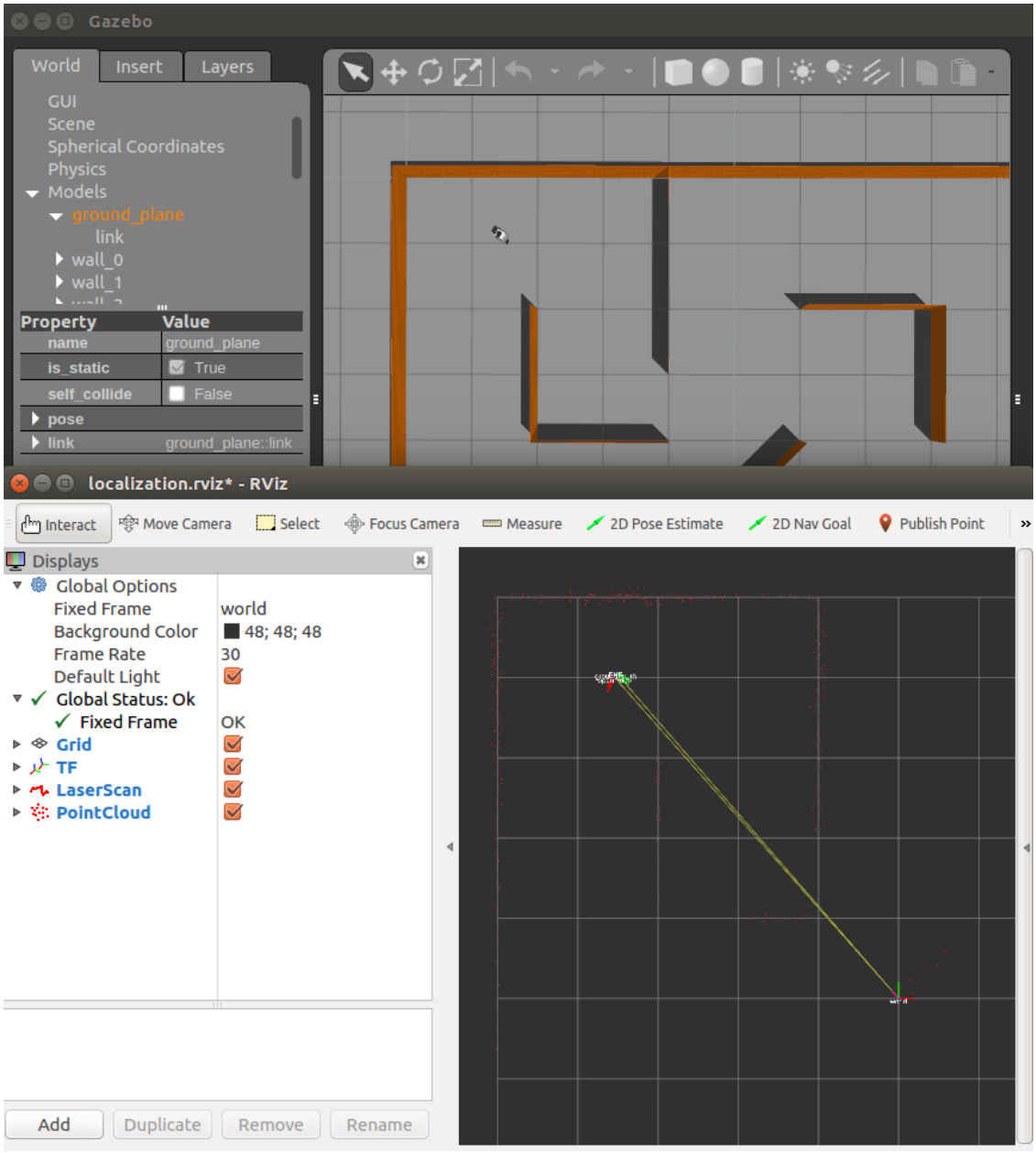
(vi) (code)

(vii) (code)

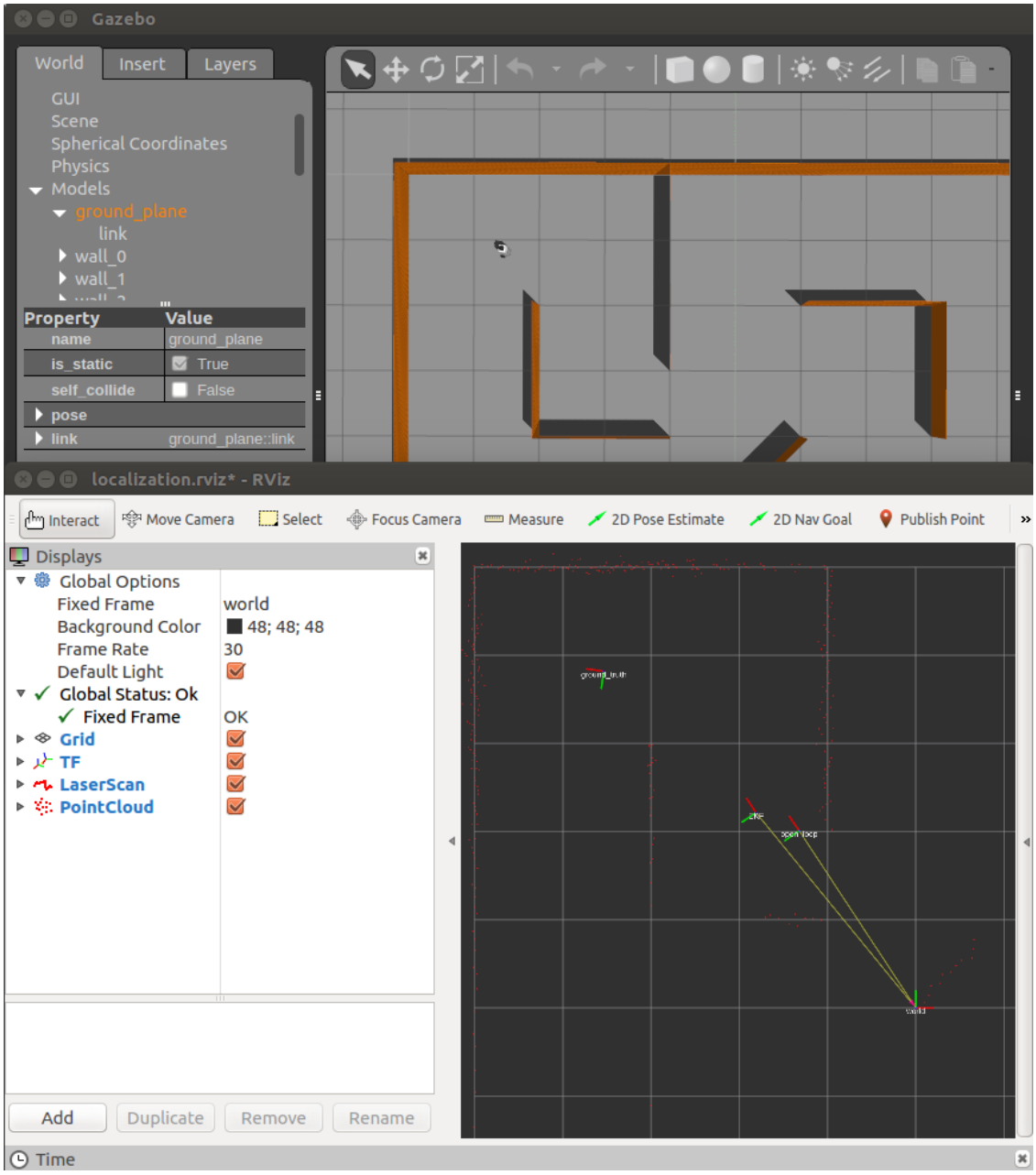
(viii) Fast movements, abrupt stops, and cumulative errors from having moved a long distance, all contribute to state divergence. Fast rotations are especially bad because by the time the update step is over, the furthest detected points could have been rotated quite far.



Initial state



After moving far



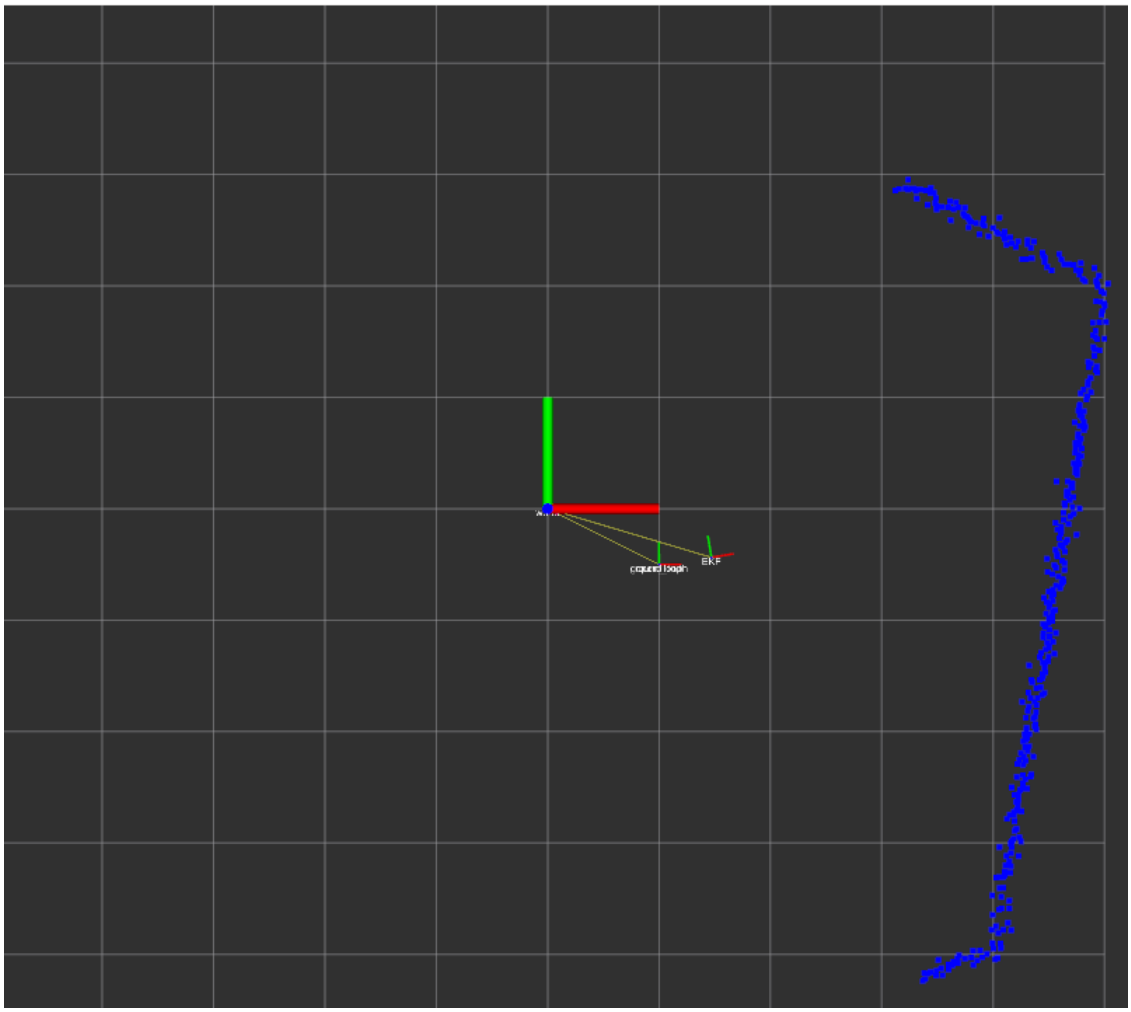
State drift

Problem 2: EKF SLAM

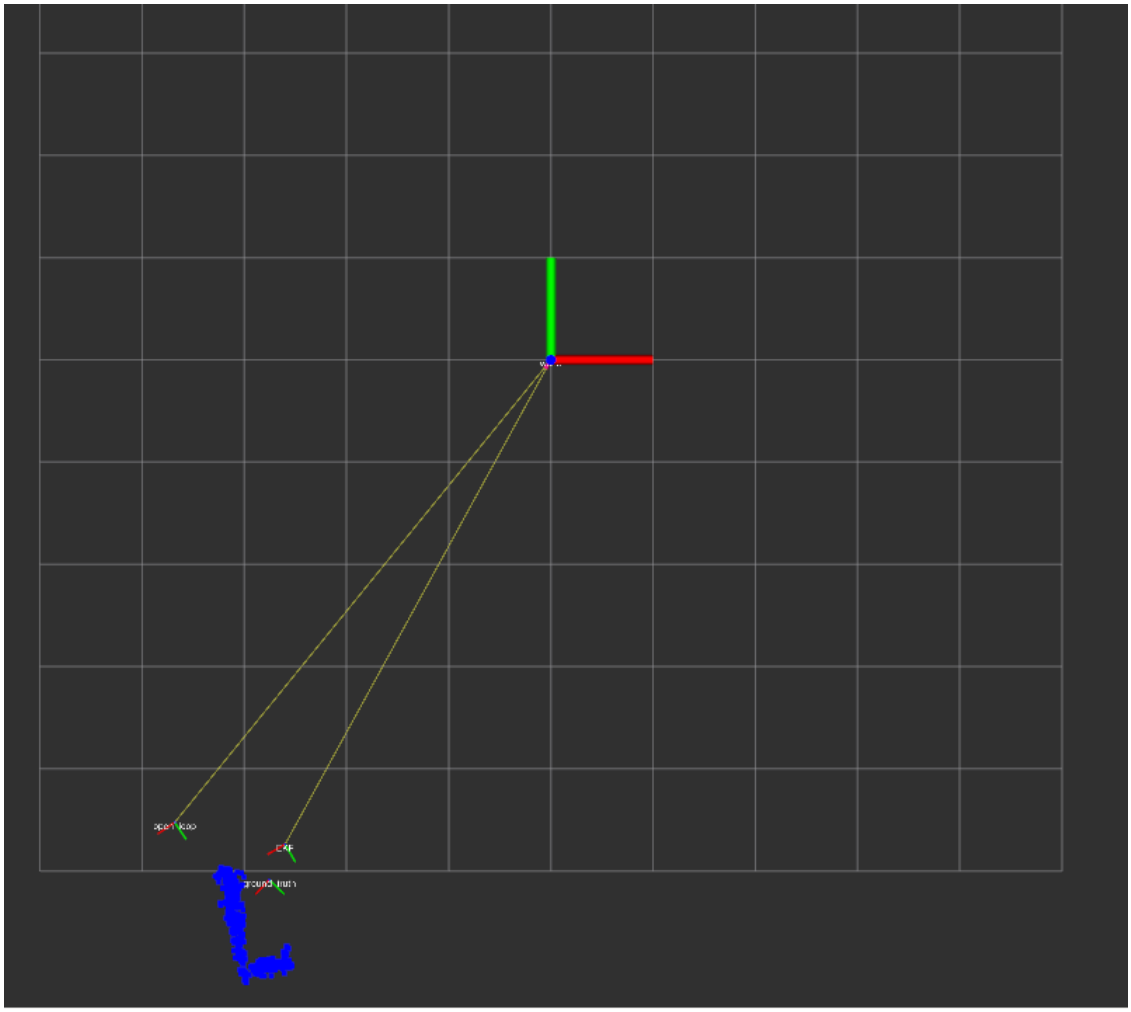
(i) (code)

(ii) (code)

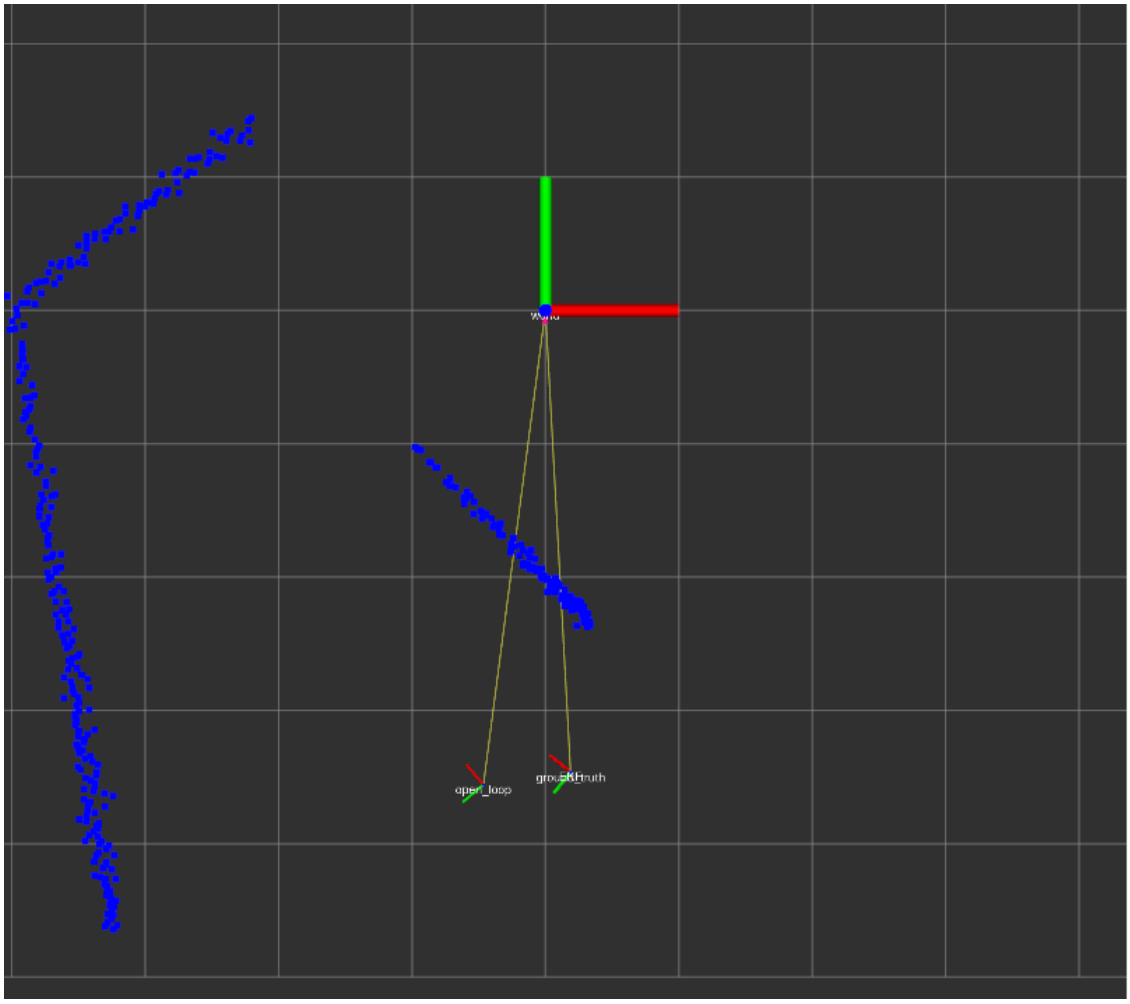
(iii) It's the same as before; fast movements, abrupt stops, and cumulative errors from having moved a long distance, all contribute to state divergence. As mentioned in the pset, if the robot can only view one wall, then it will be uncertain about its position with respect to the direction parallel to the wall. This is due to state aliasing.



Initial state



After moving far



Convergence

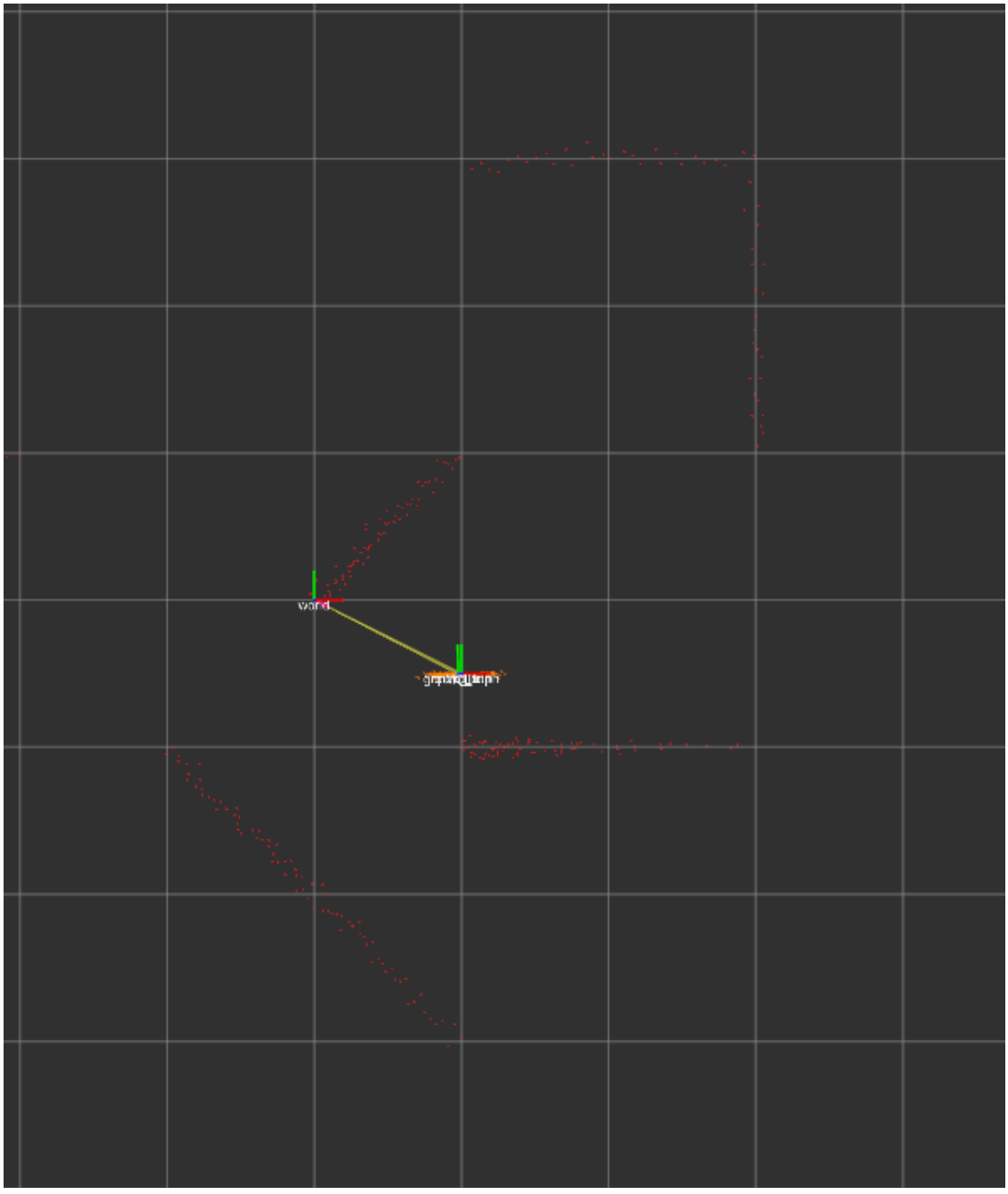
Extra Credit: Monte Carlo Localization

(i) (code)

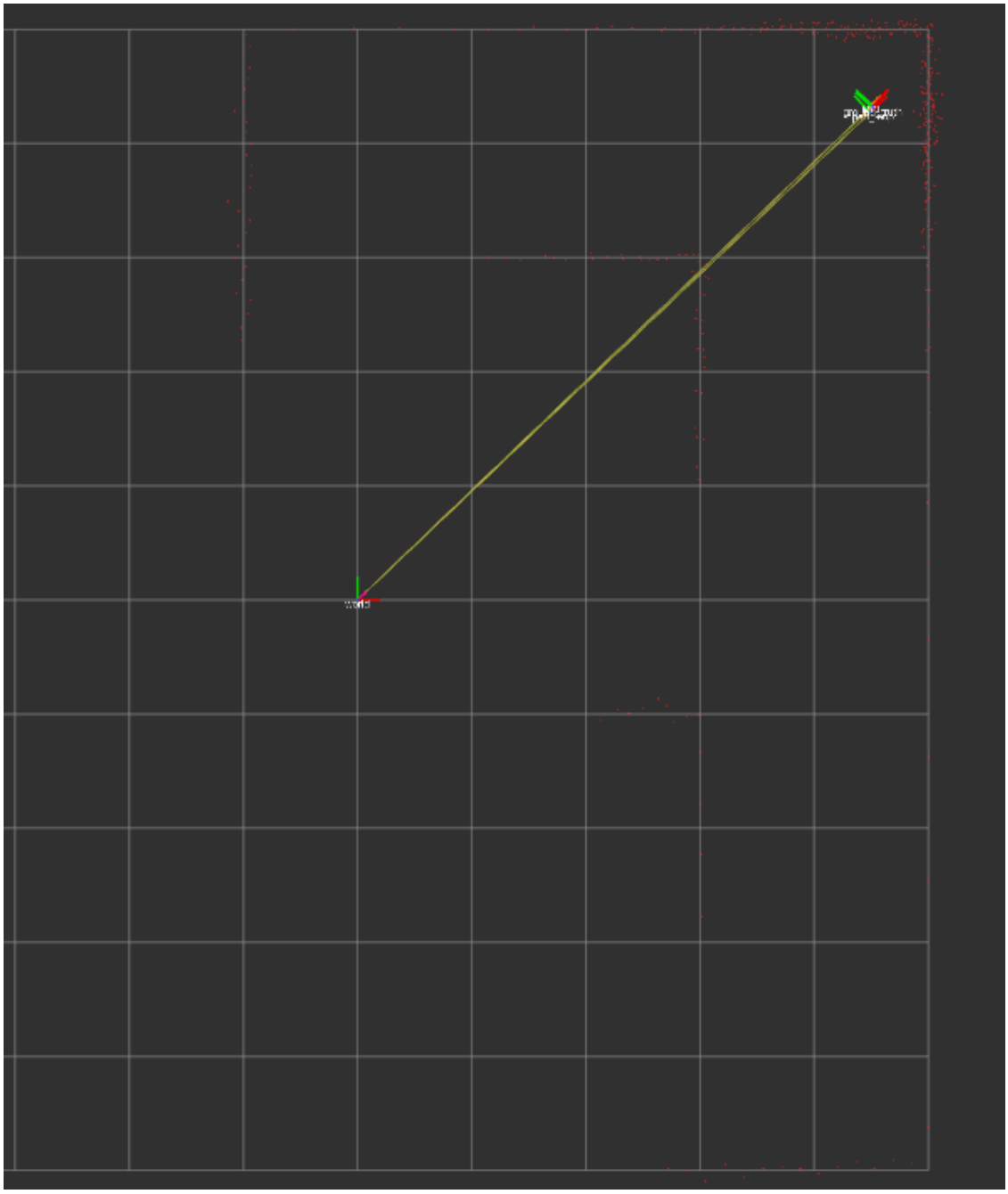
(ii) (code)

(iii) (code)

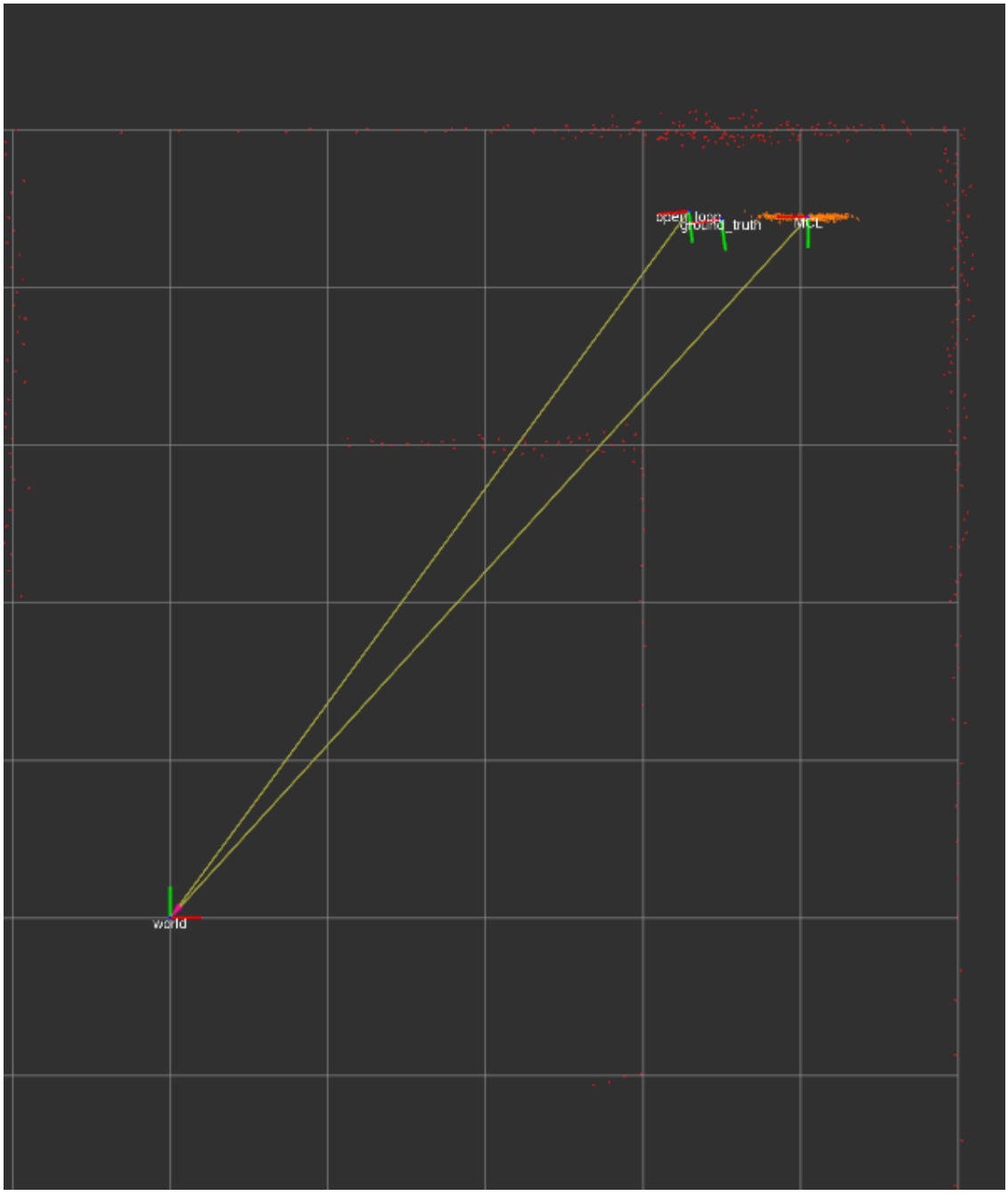
(iv) Causes of divergence are the same as before. However, when running with 1k particles or above, MCL is much more resilient to divergences in state, and the estimated state converges back to ground truth a lot sooner. However, due to the stochastic nature of the algorithm, the state estimate can jitter in place when neither the robot or the world is moving. As expected, convergence is better with more particles by the very nature of Monte Carlo sampling.



Initial state



After moving far



State divergence

(v)

```
def resample(self, xs, ws):
    """
    Resamples the particles according to the updated particle weights.
    Inputs:
        xs: np.array[M,3] - matrix of particle states.
        ws: np.array[M,]  - particle weights.
    Output:
        None - internal belief state (self.xs, self.ws) should be updated.
    """
    r = np.random.rand() / self.M
    ########## Code starts here ##########
    # The way to see the algorithm is that the random value of r generates
    # a sampling 'sieve' which we then use to pick out particles which are
    # represented in terms of their weight on a sampling interval [0, 1].
    # This sieve has as many points as we have particles.

    # r ~ U[0, 1/n]
    n = self.M
    m = np.linspace(0, n, n, endpoint=False) # {0, ..., n-1}
    sieve = r + m/n
    u = np.sum(ws) * sieve # Normalization step. Maintains [0, 1]
    csum = np.cumsum(ws)
    idx = np.searchsorted(csum, u)
    self.xs = xs[idx]
    self.ws = ws[idx]

    ########## Code ends here ##########
```

```
def transition_model(self, us, dt):
    """
    Unicycle model dynamics.
    Inputs:
        us: np.array[M,2] - zero-order hold control input for each particle.
        dt: float         - duration of discrete time step.
    Output:
        g: np.array[M,3] - result of belief mean for each particle
                           propagated according to the system dynamics with
                           control u for dt seconds.
    """
    ########## Code starts here ##########
    # TODO: Compute g.

    # We don't use numpy.where here as arrays are not lazy-evaluated.

    U, X = us.T, self.xs.T
    n = self.M      # num of particles
    V_all, om_all = U       # All of shape (n, )
    x_all, y_all, th_all = X

    # First we need to split up the particles depending on |om|
    # to use either the normal formulae or after applying l'Hopitals
    idx = np.linspace(0, n, n, endpoint=False, dtype=np.int)
    cond = np.absolute(om_all) > EPSILON_OMEGA

    # Preallocate output
    x_til = np.zeros(n)
    y_til = np.zeros(n)
    th_til = np.zeros(n)

    # Normal case
    i1 = idx[cond]
    V, om = V_all[i1], om_all[i1]
    x, y, th = x_all[i1], y_all[i1], th_all[i1]
    # We preserve particle ordering to appease the validator
    th_til[i1]  = th + om*dt
    x_til[i1]   = x + V/om * (np.sin(th+om*dt) - np.sin(th))
    y_til[i1]   = y - V/om * (np.cos(th+om*dt) - np.cos(th))

    # l'Hopital's case
    i2 = idx[~cond]
    V, om = V_all[i2], om_all[i2]
    x, y, th = x_all[i2], y_all[i2], th_all[i2]
    th_til[i2]  = th + om*dt
    x_til[i2]   = x + V*dt*np.cos(th)
    y_til[i2]   = y + V*dt*np.sin(th)

    g = np.column_stack([x_til, y_til, th_til])

    ########## Code ends here ##########
    return g
```

```
def compute_innovations(self, z_raw, Q_raw):
    """
    Given lines extracted from the scanner data, tries to associate each one
    to the closest map entry measured by Mahalanobis distance.
    Inputs:
        z_raw: np.array[2,I]   - I lines extracted from scanner data in
                                 columns representing (alpha, r) in the scanner frame.
        Q_raw: np.array[I,2,2] - I covariance matrices corresponding
                                 to each (alpha, r) column of z_raw.
    Outputs:
        vs: np.array[M,2I] - M innovation vectors of size 2I
                             (predicted map measurement - scanner measurement).
    """
    ########## Code starts here ##########
    # TODO: Compute vs (with shape [M x I x 2]).

    n     = self.M                  # Num of particles. M.
    n_lin = self.map_lines.shape[1] # Num of known lines on map. J.
    n_mea = z_raw.shape[1]          # Num of scanned lines. I.

    z_raw = z_raw.T                     # shape(n_mea, 2)
    # Q_raw                             # shape(n_mea, 2, 2)
    hs = self.compute_predicted_measurements().transpose(0, 2, 1) # shape(n, n_lin, 2)

    z_mat = z_raw[None, None, :, :]     # shape(1, 1,     n_mea, 2)
    h_mat = hs[:, :, None, :]           # shape(n, n_lin, 1,     2)

    # Vectorized angle_diff()
    z_alp, h_alp = z_mat[..., 0], h_mat[..., 0]     # shape(n, n_lin, n_mea)
    z_alp, h_alp = z_alp % (2.*np.pi), h_alp % (2.*np.pi)
    diff = z_alp - h_alp
    idx = np.abs(diff) > np.pi
    sign = 2. * (diff[idx] < 0.) - 1.
    diff[idx] += sign * 2. * np.pi
    v_alp = diff
    # Reconstruct v
    v_r = z_mat[..., 1] - h_mat[..., 1]
    v_mat = np.stack((v_alp, v_r), axis=3)

    v_fat = v_mat[..., None]                # shape(n, n_lin, n_mea, 2, 1)
    Q_inv = np.linalg.inv(Q_raw)            # shape(      n_mea, 2, 2)
    Q_inv = Q_inv[None, None, :, :, :]      # shape(1, 1, n_mea, 2, 2) # PEP20

    d_mat = np.matmul(v_fat.transpose(0, 1, 2, 4, 3), Q_inv)
    d_mat = np.matmul(d_mat, v_fat)         # shape(n, n_lin, n_mea, 1, 1)
    d_mat = d_mat.reshape((n,n_lin,n_mea))  # shape(n, n_lin, n_mea)

    # For each particle, for each scanned line, this returns the index
    # of the best known line.
    d_argmin = np.argmin(d_mat, axis=1)                 # shape(n, n_mea)
    d_argmin = d_argmin[:, None, :, None]               # shape(n, 1, n_mea, 1)
    vs = np.take_along_axis(v_mat, d_argmin, axis=1)    # shape(n, 1, n_mea, 2)
    vs = vs.reshape((n, n_mea, 2))                      # shape(n, n_mea, 2)

    ########## Code ends here ##########

    # Reshape [M x I x 2] array to [M x 2I]
    return vs.reshape((self.M,-1))  # [M x 2I]
```

```
def compute_predicted_measurements(self):
    """
    Given a single map line in the world frame, outputs the line parameters
    in the scanner frame so it can be associated with the lines extracted
    from the scanner measurements.
    Input:
        None
    Output:
        hs: np.array[M,2,J] - J line parameters in the scanner (camera) frame for M particles.
    """
    ########## Code starts here ##########
    # TODO: Compute hs.

    # n = self.M                      # Num of particles
    # d = self.xs.shape[1]            # 3 for (x, y, th)
    # n_lin = self.map_lines.shape[1] # Num of lines on map. This is our pset fudge.
    #                                 # We're not generally supposed to know this.

    hs = self.map_lines.T             # shape(n_lin, 2)
    alp, r = hs[:, 0], hs[:, 1]

    x, y, th = self.xs.T              # shapes(3, )
    xcam_R, ycam_R, thcam_R = self.tf_base_to_camera # Camera pose. in Robot frame.

    xcam = xcam_R*np.cos(th) - ycam_R*np.sin(th) + x
    ycam = xcam_R*np.sin(th) + ycam_R*np.cos(th) + y

    # shapes(n, n_lin)
    alp_C = alp[None, :] - th[:, None] - thcam_R
    r_C = (r[None, :] - xcam[:, None]*np.cos(alp)[None, :] -
                        ycam[:, None]*np.sin(alp)[None, :])

    # Vectorized tb.normalize_line_parameters
    cond = r_C < 0
    alp_C[cond] += np.pi
    r_C[cond]   *= -1
    alp_C = (alp_C + np.pi) % (2*np.pi) - np.pi

    hs = np.array([alp_C, r_C]).transpose(1, 0, 2)  # shape(n, 2, n_lin)

    ########## Code ends here ##########
    return hs
```