[16-833] Homework 1 : Written Report

Bharath Somayajula, Christopher Klammer

February 25, 2023

Contents

1 Motion Model

TODO Chris

2 Sensor Model

2.1 Sensor Model

2.1.1 Description

The Sensor Model is responsible for estimating how well each particle explains the observed range sensor data. For our implementation, we followed the range sensor model described in [1] where the probability distribution of sensor measurement is modeled as a

weighted average of four components that take into account randomness, presence of obstacles and errors in measurements leading to the sensor incorrectly measuring maximum range.

Algorithm beam_range_finder_model(z_t, x_t, m):

$q = 1$

for $k = 1$ *to* K *do*

compute z_t^{k*} *for the measurement* z_t^k *using ray casting*

$p = z_{\text{hit}} \cdot p_{\text{hit}}(z_t^k \mid x_t, m) + z_{\text{short}} \cdot p_{\text{short}}(z_t^k \mid x_t, m)$

$+ z_{\text{max}} \cdot p_{\text{max}}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$

$q = q \cdot p$

return q

Figure 1: Description of Sensor model

The Sensor Model has been implemented in *SensorModel* class in *sensor_model.py*. The constructor accepts the occupancy map and number of particles as inputs.

The function *beam_range_finder_model* is called from *main.py* everytime a laser reading is encountered. This function performs three important tasks:

1. Get ground truth range readings using ray casting module
2. Use ground truth and sensor readings to compute probabilities of each measurement for each particle
3. Aggregate probabilities *for* each particle using sum of logarithm of probabilities for numerical stability and then normalize probabilities *across* particles using softmax operation to ensure that probabilities sum to 1.

2.1.2 Optimization

The *SensorModel* module takes up almost all (99.1% according to our estimates) of execution time. This makes optimization absolutely essential. We made the following changes to naive implementation to speed up the code:

1. We vectorized the function *beam_range_finder_model* and all the functions it invokes to compute probabilities of all particles at once instead of invoking the function iteratively for each particle
2. The functions to compute the four components of probability- *p_hit, p_short, p_max, p_rand* have all been vectorized to compute probabilities for all particles simultaneously. An example is shown in figure below

- ```
  def p_short(self, z_t, z_gt):
          #SHORT PROBABILITY
          #initialize probabilties
          p2 = np.zeros_like(z_t)

          #mask for non zero probabilities
          mask = z_t <= z_gt

          #compute normalization factors
          eta = 1/(1 - np.exp(-self._lambda_short*z_gt[mask]))

          #compute probabilties
          p2[mask] =  eta * self._lambda_short*np.exp(-self._lambda_short*z_t[mask])

          return p2
  ```

Figure 2: Vectorized probability computation

2.2 Ray Casting

2.2.1 Description

RayCasting is needed for estimation of true ranges at various angles.

To make implementation easier, we created a *RayCasting* module that performs ray casting. The *get_true_ranges_vec* function is invoked to compute ranges at various angles for all particles.

2.2.2 Optimization

Ray Casting is the most computationally expensive part of the code. We implemented the following changes to naive implementation of ray casting to improve the performance.

- To avoid redundant computations, we perform ray casting only once in constructor function using the *relative_ray_casting* function. This function casts rays relative to the robot in it's canonical orientation. At all future time steps, the points along these rays are simply rotated and translated to match the position and orientation of robot! This led to massive gains in performance. The implementation of *relative_ray_casting* function is shown below.
  ```

```python
def relative_ray_casting(self):
    """Perform ray casting relative to robot's position

    Returns:
        np.ndarray: array containing x,y values of each point along each ray
    """
    #angles at which rays are cast
    # Take the angles going counter-clockwise
    angles = np.arange(-90, 90, self._subsampling)

    #distances at which z is computed along each ray
    diag_length = math.ceil(math.sqrt(self.h**2 + self.w**2))
    dists = np.arange(0, diag_length, 1)

    #number of angles and points along each ray
    num_angles = angles.size
    num_points = dists.size

    #create array to store x and y for each point in ray
    rays = np.zeros((num_angles, num_points, 2))

    #perform ray casting relative to robot's location
    for i, a in enumerate(angles):
        for j, d in enumerate(dists):
            x = d*np.cos((np.pi/180)*a)
            y = d*np.sin((np.pi/180)*a)
            rays[i, j] = [x, y]

    #duplicate rays for every particle
    rays = np.tile(rays, (self.num_particles, 1, 1, 1))

    #scale distances
    rays = self.resolution*rays

    return rays
```

Figure 3: Pre-computation of points along rays

- Once the rays have been oriented based on location and orientation of a particle, the logic used to measure the range along each ray at which it encounters an obstacle or goes out of the map has been vectorized. Compared to the naive implementation of this logic using nested for-loops, the fully vectorized implementation is approximately 9.5x faster. The vectorized logic is shown below

- Vectorize the code to enable computation of ranges for all particles simultaneously instead of performing ray casting in a loop seperately for each particle

```python
def get_true_ranges_vec(self, x_t):
        """Find true depths at all angles for a given robot state

        Args:
            x_t (list): states of robot represeented by particles
        """

        #adjust robot's state into laser's state
        x_t = self.sensor_location(x_t)

        #adjust rays based on robot's state
        rays = self.transform_rays(x_t)

        #find obstacle along each ray
        num_angles = rays.shape[1]

        #array to store ranges
        z_true = np.zeros((self.num_particles, num_angles))

        #convert cm to px
        x_int = np.round(rays[:,:,:,0]/self.resolution).astype(np.int32)
        y_int = np.round(rays[:,:,:,1]/self.resolution).astype(np.int32)

        #filter coordinates outside map
        m_lx = x_int < 0
        m_ly = y_int < 0
        m_hx = x_int >= self.w
        m_hy = y_int >= self.h

        # Filter out pixels on or beyond the boundary
        m_filter = np.logical_or.reduce((m_lx, m_ly, m_hx, m_hy))

        # Clip x and y coordinates to their max and min values
        x_int[m_filter] = 0
        y_int[m_filter] = 0

        #find coordinates that hit the obstacle
        obstacles = np.bitwise_or(self.map[y_int, x_int] == -1, \
                                  self.map[y_int, x_int] >= self._min_probability)

        #take intersection of filter and obstacle masks
        # This is all the occupied areas of the map
        occupied = np.bitwise_or(m_filter, obstacles).astype("int")

        # Travel along each ray for a given particle and angle
        # Once we hit a point that is intraversible, our cumsum will be 1
        m_overall_cumsum = np.cumsum(occupied, axis=2)

        # Look at the particle, angle, and distance indexes to look for the boundaries
        particles, angs, dists = np.where(m_overall_cumsum == 1)

        #populate z_true based on x and y
        z_true[particles, angs] = dists

        #scale range
        z_true *= self.resolution

        return z_true, x_int, y_int
```

Figure 4: Vectorized logic to compute ranges

The time taken to measure ground truth ranges at 36 angles for 500 particles using our

optimized implementation is just around 0.5 seconds on Apple Macbook M1 Pro system.

3 Resampling Process

Resampling is the process by which new particles are generated from a set of modified particles and their corresponding weights.

We used low variance resampling to preserve the diversity in particle set. The figures below show the algorithm and our implementation.

Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$):

$\bar{\mathcal{X}}_t = \emptyset$

$r = \text{rand}(0; M^{-1})$

$c = w_t^{[1]}$

$i = 1$

for $m = 1$ *to* M *do*

  $U = r + (m-1) \cdot M^{-1}$

  while $U > c$

    $i = i + 1$

    $c = c + w_t^{[i]}$

  endwhile

  add $x_t^{[i]}$ *to* $\bar{\mathcal{X}}_t$

endfor

return $\bar{\mathcal{X}}_t$

Figure 5: Description of Resampling algorithm

```python
def low_variance_sampler(self, X_bar):
    """
    Outcomes:

    - Select samples independently of one another
    - We want to make sure that when we have many of the same particles, we don't destroy the variance a
s an estimator
    - We want to make sure we want to not lose diversity and just create samples of the same particle ov
er and over again

    param[in] X_bar : [num_particles x 4] sized array containing [x, y, theta, wt] values for all partic
les
    param[out] X_bar_resampled : [num_particles x 4] sized array containing [x, y, theta, wt] values for
resampled set of particles
    """
    # Resampled particles
    X_bar_resampled =  np.zeros_like(X_bar)

    # Init variables
    M      = X_bar.shape[0]
    r      = np.random.uniform(0, 1.0/M)
    w      = X_bar[:, 3]
    w     /= np.sum(w)
    c      = w[0]
    i      = 0

    # Here we will go through the particles and start the interplay of c and U
    # U will increment a fixed amount each iteration

# c will essentially cumulatively keep track of the sum of the particles weights so far and will look to
exceed U

# Therefore, Very small weighted particles are unlikely but could possibly be sampled, higher weights ha
ve a proportionally better chance
    for m in range(M):
        # Upper bound, when c surpasses this threshold, sample the particle
        U = r + m * (1.0/M)

        while U > c:
            i += 1
            c += w[i]

        # c caught up to U, sample the particle
        X_bar_resampled[m] = X_bar[i]

    return X_bar_resampled
```

Figure 6: Implementation of resampling

4 Performance

The total time taken for execution of particle filter on *robotdata1.log* with 1000 particles on Apple Macbook M1 Pro is **19.6 min**. The execution time without the optimizations in ray casting is more than 3 hours(180 min).

5 Parameter Tuning

TODO Chris

6 Future Work

TODO Chris

1. Implementing ray casting on GPUs that specialize in matrix multiplications and additions can further reduce the execution time of particle filter

2. One interesting way to reduce time spent in ray casting is to approximate all obstacles in the map by a set straight lines. In this case, all the points that determine the ranges at various angles are simply intersections of any given ray and a set of lines! These points of intersection(and hence the ranges) can be computed at very little computational cost. This can potentially enable execution of particle filter in real time.

7 Extra Credit

7.1 Kidnapped robot problem

TODO Bharath

7.2 Adaptive number of particles

TODO Chris