# Reward Function 1 (Trials 1-7)

* Proximity reward (5)
* Crash Penalty (-30)
* Goal Reward (30)
* Throttle Reward (1.1)
* Steer Reward (0.5)
  + Non-zero average steer penalty

    def \_get\_reward(self, new\_state: np.ndarray, old\_state: np.ndarray, action: np.ndarray) -> float:

        """

        the state is a memory buffer of size self.memory\_size \* self.observation\_count that holds the past 'self.memory\_size' measurements for

        each of the 'self.observation\_count' observations.

        Example: Let self.observation\_count = 2 so we can use 1 target distance measurement and 1 proximity sensor measurement.

                 Then, let self.memory\_size = 3, which means at step n, we have a state array of the form:

                 state = [target\_distance\_n, target\_distance\_(n-1), target\_distance\_(n-2), sensor\_measurement\_n, sensor\_measurement\_(n-1), sensor\_measurement\_(n-2)]

                 The latest measurement (measurement\_n) for each type of observation is always at the beginning of its respective section and can be accessed through its respective 'base index' ('bi')

        """

        # proximity reward

        w\_p = 5

        r\_p = 0

        for i in reversed(range(self.d\_bi + 1, self.d\_bi + self.memory\_size)):

            r\_p += (self.current\_state[i] - self.current\_state[i-1])

        self.\_update\_terminal\_flag(new\_state)

        # NOTE: r\_p is used in some of the remaning rewards as a 'scaling factor' to make sure all rewards are of approximately the same scale

        # crash penalty

        w\_c = -30

        r\_c = int(self.bad\_terminal)

        # goal reward

        w\_g = 30

        r\_g = int(self.good\_terminal)

        # throttle reward

        w\_t = 1.1

        if action[0] < 0:

            r\_t = action[0] \* abs(r\_p) # <- penalize backward motion,

        else:

            r\_t = 0 # <- nothing for forward motion

        # steer reward

        if len(self.steer\_history) > 1:

            avg\_steer = np.array(self.steer\_history).mean() # <- calculate average steer

        else:

            avg\_steer = 0

        w\_s = 0.5

        r\_s = (-abs(avg\_steer)) \* abs(r\_p) # <- penalize non-zero average steer

        return r\_p\*w\_p + r\_g\*w\_g + r\_c\*w\_c + r\_t\*w\_t + r\_s\*w\_s

# Reward Function 2

* Proximity reward (**1**)
* Crash Penalty (**-2**)
* Goal Reward (30)
* Throttle Reward (1.1)
* Steer Reward (0.5)
  + Non-zero average steer penalty

We hope that decreasing the crash penalty dramatically on top of changing the simulation so that it doesn’t reach a terminal state after hitting an obstacle would incentivize the car to navigate obstacles better. We also changed the proximity reward to 1 as reward balancing.

    def \_get\_reward(self, new\_state: np.ndarray, old\_state: np.ndarray, action: np.ndarray) -> float:

        """

        the state is a memory buffer of size self.memory\_size \* self.observation\_count that holds the past 'self.memory\_size' measurements for

        each of the 'self.observation\_count' observations.

        Example: Let self.observation\_count = 2 so we can use 1 target distance measurement and 1 proximity sensor measurement.

                 Then, let self.memory\_size = 3, which means at step n, we have a state array of the form:

                 state = [target\_distance\_n, target\_distance\_(n-1), target\_distance\_(n-2), sensor\_measurement\_n, sensor\_measurement\_(n-1), sensor\_measurement\_(n-2)]

                 The latest measurement (measurement\_n) for each type of observation is always at the beginning of its respective section and can be accessed through its respective 'base index' ('bi')

        """

        # proximity reward

        w\_p = 1

        r\_p = 0

        for i in reversed(range(self.d\_bi + 1, self.d\_bi + self.memory\_size)):

            r\_p += (self.current\_state[i] - self.current\_state[i-1])

        self.\_update\_terminal\_flag(new\_state)

        # NOTE: r\_p is used in some of the remaning rewards as a 'scaling factor' to make sure all rewards are of approximately the same scale

        # crash penalty

        w\_c = -2

        r\_c = int(self.collided)

        # goal reward

        w\_g = 30

        r\_g = int(self.good\_terminal)

        # throttle reward

        w\_t = 1.1

        if action[0] < 0:

            r\_t = action[0] \* abs(r\_p) # <- penalize backward motion,

        else:

            r\_t = 0 # <- nothing for forward motion

        # steer reward

        if len(self.steer\_history) > 1:

            avg\_steer = np.array(self.steer\_history).mean() # <- calculate average steer

        else:

            avg\_steer = 0

        w\_s = 0.50

        r\_s = (-abs(avg\_steer)) \* abs(r\_p) # <- penalize non-zero average steer

        return r\_p\*w\_p + r\_g\*w\_g + r\_c\*w\_c + r\_t\*w\_t + r\_s\*w\_s

# Reward Function 3

* Proximity reward (1)
* **Distance-Based Crash Penalty** (-2)
* Goal Reward (30)
* Throttle Reward (1.1)
* Steer Reward (0.5)
  + Non-zero average steer penalty

    def \_get\_reward(self, new\_state: np.ndarray, old\_state: np.ndarray, action: np.ndarray) -> float:

        """

        the state is a memory buffer of size self.memory\_size \* self.observation\_count that holds the past 'self.memory\_size' measurements for

        each of the 'self.observation\_count' observations.

        Example: Let self.observation\_count = 2 so we can use 1 target distance measurement and 1 proximity sensor measurement.

                 Then, let self.memory\_size = 3, which means at step n, we have a state array of the form:

                 state = [target\_distance\_n, target\_distance\_(n-1), target\_distance\_(n-2), sensor\_measurement\_n, sensor\_measurement\_(n-1), sensor\_measurement\_(n-2)]

                 The latest measurement (measurement\_n) for each type of observation is always at the beginning of its respective section and can be accessed through its respective 'base index' ('bi')

        """

        # proximity reward

        w\_p = 1

        r\_p = 0

        for i in reversed(range(self.d\_bi + 1, self.d\_bi + self.memory\_size)):

            r\_p += (self.current\_state[i] - self.current\_state[i-1])

        self.\_update\_terminal\_flag(new\_state)

        # NOTE: r\_p is used in some of the remaining rewards as a 'scaling factor' to make sure all rewards are of approximately the same scale

        initial\_distance = np.sqrt((self.target\_x - self.initial\_x)\*\*2 + (self.target\_y - self.initial\_y)\*\*2)

        final\_distance = np.sqrt((self.target\_x - self.car.P\_f[0])\*\*2 + (self.target\_y - self.car.P\_f[1])\*\*2)

        # crash penalty

        w\_c = -2

        r\_c = int(self.bad\_terminal) \* abs(final\_distance - initial\_distance)

        # goal reward

        w\_g = 30

        r\_g = int(self.good\_terminal)

        # throttle reward

        w\_t = 1.1

        if action[0] < 0:

            r\_t = action[0] \* abs(r\_p) # <- penalize backward motion,

        else:

            r\_t = 0 # <- nothing for forward motion

        # steer reward

        if len(self.steer\_history) > 1:

            avg\_steer = np.array(self.steer\_history).mean() # <- calculate average steer

        else:

            avg\_steer = 0

        w\_s = 0.5

        r\_s = (-abs(avg\_steer)) \* abs(r\_p) # <- penalize non-zero average steer

        return r\_p\*w\_p + r\_g\*w\_g + r\_c\*w\_c + r\_t\*w\_t + r\_s\*w\_s

# Reward Function 4

* Proximity reward (1)
* Distance-Based Crash Penalty (**-1**)
* Goal Reward (30)
* Throttle Reward (1.1)
* Steer Reward (0.5)
  + Non-zero average steer penalty

    def \_get\_reward(self, new\_state: np.ndarray, old\_state: np.ndarray, action: np.ndarray) -> float:

        """

        the state is a memory buffer of size self.memory\_size \* self.observation\_count that holds the past 'self.memory\_size' measurements for

        each of the 'self.observation\_count' observations.

        Example: Let self.observation\_count = 2 so we can use 1 target distance measurement and 1 proximity sensor measurement.

                 Then, let self.memory\_size = 3, which means at step n, we have a state array of the form:

                 state = [target\_distance\_n, target\_distance\_(n-1), target\_distance\_(n-2), sensor\_measurement\_n, sensor\_measurement\_(n-1), sensor\_measurement\_(n-2)]

                 The latest measurement (measurement\_n) for each type of observation is always at the beginning of its respective section and can be accessed through its respective 'base index' ('bi')

        """

        # proximity reward

        w\_p = 1

        r\_p = 0

        for i in reversed(range(self.d\_bi + 1, self.d\_bi + self.memory\_size)):

            r\_p += (self.current\_state[i] - self.current\_state[i-1])

        self.\_update\_terminal\_flag(new\_state)

        # NOTE: r\_p is used in some of the remaining rewards as a 'scaling factor' to make sure all rewards are of approximately the same scale

        initial\_distance = np.sqrt((self.target\_x - self.initial\_x)\*\*2 + (self.target\_y - self.initial\_y)\*\*2)

        final\_distance = np.sqrt((self.target\_x - self.car.P\_f[0])\*\*2 + (self.target\_y - self.car.P\_f[1])\*\*2)

        # crash penalty

        w\_c = -1

        r\_c = int(self.bad\_terminal) \* abs(final\_distance - initial\_distance)

        # goal reward

        w\_g = 30

        r\_g = int(self.good\_terminal)

        # throttle reward

        w\_t = 1.1

        if action[0] < 0:

            r\_t = action[0] \* abs(r\_p) # <- penalize backward motion,

        else:

            r\_t = 0 # <- nothing for forward motion

        # steer reward

        if len(self.steer\_history) > 1:

            avg\_steer = np.array(self.steer\_history).mean() # <- calculate average steer

        else:

            avg\_steer = 0

        w\_s = 0.5

        r\_s = (-abs(avg\_steer)) \* abs(r\_p) # <- penalize non-zero average steer

        return r\_p\*w\_p + r\_g\*w\_g + r\_c\*w\_c + r\_t\*w\_t + r\_s\*w\_s