

## Localization Monte Carlo Localization Discrete & Multimodal

Posterior Distribution 后验 belief, 在 sense measurement 完成后定义存在 Bad Measurement

→ Best representation of current belief 每个凸起代表了 robot 对自己和门相对位置的估计

Robot Movement Robot 移动, bumps 会随之移动. 但会更平缓 (Shift and Flatten)

- Shift and Flatten Robot Motion 的不确定性: Robot 不知道自己走了多远 因此 knowledge 不准确 Convolution 衡量两个函数的 Overlap

• 第一遍 measurement 之后, 第二次就有了 Prior information Posterior Function 和移动距离函数的卷积  
→ 结合后会出现一个峰值 有 Prior 的 Measurement 更准确 Histogram Filter

→ 每一次 Measurement, 对位置的确定性就越高 试验次数越多结果越精确 → Monte Carlo 方法

Probability After Sense 建立初始的 location belief 后, 在给定 measurement 的前提下更新 belief

$$P(x_i | z) = \frac{P(z|x_i)}{P(z)} \text{ Given measurement } z, \text{ from robot's sensors}$$

Bayes Rules

Function Sense Update the measurement

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]
world = ['green', 'red', 'red', 'green', 'green']
z = 'red'
pHit = 0.6 #匹配测量因子 Factor for Matching Measurement
pMiss = 0.2 #不匹配测量因子 Factor for non-Matching Measurement
# world 和 Measurement 是否相同 ⇒ 更可能归属哪个颜色
def sense(p, z):
    q = []
    for i in range(len(p)):
        hit = (z == world[i]) # measurement 和 world 相同为 1, 不同为 0
        q.append(p[i] * (hit * pHit + (1 - hit) * pMiss))
    return q
    print sense(p, z)
```

Input p & measurement z 形感应到的颜色

Output q p 和 pHit 的乘积 Product 根据 Bayes

Normalization 归一化

```
#First, compute the sum of vector q, using the sum function
s = sum(q)
for i in range(len(p)):
    # normalize by going through all the elements in q and divide
    q[i] = q[i] / s
```

Exact Motion Function Move

Robot 移动一次, Posterior 概率也会随之移动

```
def move(p, U):
    q = [] #Start with empty list
    for i in range(len(p)):
        #Go through all the elements in '.....'p'....'
        #Construct "'q'" element by element by accessing
        corresponding ''p'' which is shifted by 'U'
        q.append(p[(i-U) % len(p)]) #是从 p 的 cell 中提取的
    return q
```

右移 U 位就是从 p 的左边 U 位提取的

Inaccurate Robot Motion

→ 有一定概率执行错误的 action

可能会 undershoot 或 overshoot 更符合真实场景

卷积的  
全概率

def move(p, U):

```
#Introduce auxiliary variable s
q= []
for i in range(len(p)):
    s = pExact * p[(i-U) % len(p)]
    s = s + pOvershoot * p[(i-U-1) % len(p)]
    s = s + pUndershoot * p[(i-U+1) % len(p)]
    q.append(s)
return q
```

[0.0, 0.1, 0.8, 0.1, 0.0]

Limit Distribution Quiz 只执行 motion, 从不执行 sense

→ 移动前的 Distribution 和移动后相同 Stationary State

→ 最终结果为 uniform distribution 移动不改变 Distribution

→ 移动两次后, Distribution 被展平且拓宽; 移动 1000 次后, 失去所有的位置信息. 定位需要持续 sense

Robot Sense and Movement

- Monte Carlo Localization Sense 和 move 的不断重复

Sense 获取信息, move 失去信息, → Entropy 用以衡量 Distribution 中带有的信息量

$$-\sum p(x_i) \log p(x_i)$$

Summary

- Localization → Robot 遍历所有可能的 Location 并持续更新关于 location 的 belief 每个 Location Updating probability distribution over sample space 部分配一个概率并更新

• Probability Distribution 的特征

1. It should have one sharp peak. This indicates that the car has a very specific belief about its current location. Distribution 需要有一个尖峰 Peak
2. The peak should be correct! If this weren't true, the car would believe—very strongly—that it was in the wrong location. This would be bad for Stanley.

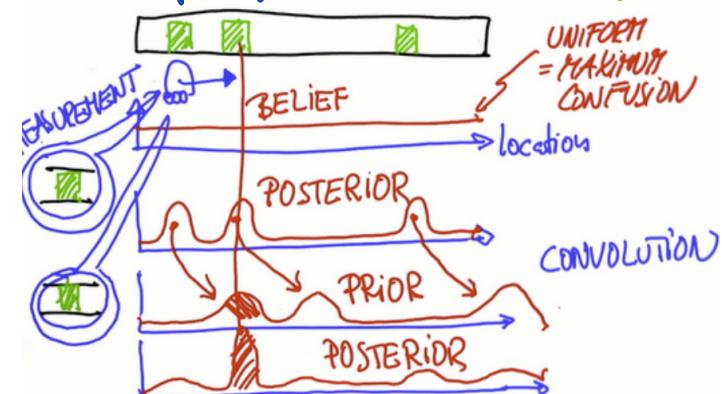
• Monte Carlo Location Procedure

1. Start with a belief set about our current location.
2. Multiply this distribution by the results of our sense measurement.
3. Normalize the resulting distribution.
4. Move by performing a convolution. This sounds more complicated than it really is: this step really involved multiplication and addition to obtain the new probabilities at each location.

Uniform Distribution  $\leftrightarrow$  Maximum Confusion

→ 初始状态不确定性最大, 假设为 Uniform 分布

可能有多个 Peak 但只有一个 Sharp Peak



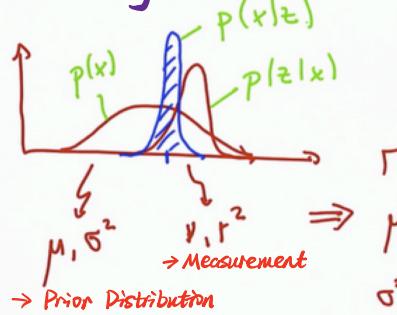
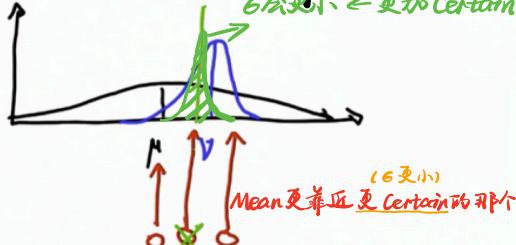
## Kalman Filters Continuous & Uni-modal 估测系统中的 State

Gaussian Distribution - 独立 GD 都由  $\mu, \sigma^2$  决定

- Histogram 将连该空间分为高数区域用以估测
- Kalman Filter 的任务：维持  $\mu$  和  $\sigma^2$  作为 Object 位姿的最佳估计
- Single Peak  $\Leftrightarrow$  Uni-modal Model  $\rightarrow$  Distribution Is Mode  
Mode 是一个 Distribution 中出现最多的数字 (最可能的)  $\mu$  时最大
- Covariance 方差 Uncertainty 的度量  $\sigma^2$  越大，对于实际 state 的不确定性越高

### Measuring Motion

- Measurement Update  $\rightarrow$  Total Probability, convolution, addition



New Measurement + Previous Belief  $\rightarrow$  New Belief  
(More certain than Previous)

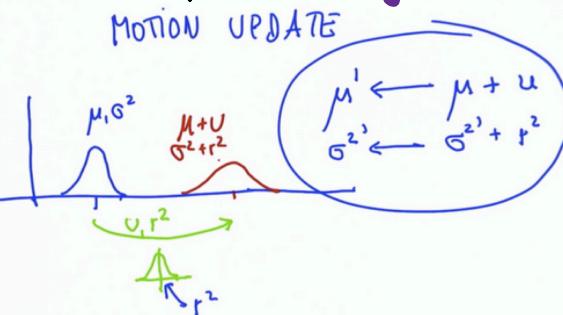
$$\mu' = \frac{r^2 \mu + \sigma^2 z}{r^2 + \sigma^2}$$

$$\sigma'^2 = \frac{1}{\frac{1}{r^2} + \frac{1}{\sigma^2}}$$

WHERE IS THE NEW MEAN  
Peak 更高  $\leftarrow$  多次 Measure 错误率更低

- Motion Update  $\rightarrow$  Bayes Rule, product, multiplication

Previous Belief + Inferred  $\rightarrow$  New Belief



$$\mu' \leftarrow \mu + u$$

$$\sigma'^2 \leftarrow \sigma^2 + r^2$$

### Kalman Filter Keys

$$x' = x + u$$

- 只用一个 Observed 的变量通过一组方程推断其他变量

Kalman Filter 的变量  $\rightarrow$  State ( $x$ )

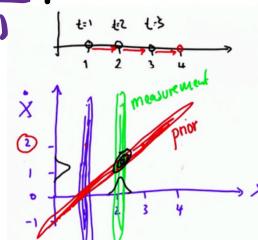
$\rightarrow$  反映了物理世界的状态。

- Observables Location

- Hidden Velocity

不能直接 measure

只能通过 location 推断



- Kalman Filter Design State Transition Function  $F$

$\rightarrow$  model-based Measurement Matrix  $H$

Uses a model of the system 对系统建模

Takes in noisy observations 用于估测系统中的当前状态。

The state,  $\bar{x}$ , evolution over time:

$$\bar{x}_k = F \bar{x}_{k-1} + w, \text{ and } w \sim N(0, Q)$$

(1)

The observations,  $\bar{z}$ , as a function of the state:

$$\bar{z}_k = H \bar{x}_k + v, \text{ and } v \sim N(0, R)$$

F 用来  $\bar{x}_{k-1} \rightarrow \bar{x}_k$

(2)

H 用于  $\bar{x}_k \rightarrow \bar{z}_k$

An estimate of the uncertainty,  $P$ , of the state:

$$P_k = cov(\bar{x}_k)$$

(3)

Where:

$F$  is known as the state transition matrix.  $F$  "takes" the state  $\bar{x}$  from one time step to the next.  $H$  transforms the state space,  $\bar{x}_k$ , into the observation (measurement) space of  $\bar{z}_k$ .

$Q$  is a covariance matrix that models the errors (or uncertainty) in the state dynamics model.

$R$  is a covariance matrix that models the errors in the observations.

$P \rightarrow$  State Uncertainty  $R \rightarrow$  Observation Uncertainty

$Q \rightarrow$  State Dynamics Model Uncertainty

(从  $k-1$  到  $k$  变化的不确定性)

上标  $\wedge$  表示预测

### Time Update (Prediction)

$$\hat{x}_k = F \bar{x}_{k-1} \quad \text{用 } k-1 \text{ 的状态预测 } k \text{ 时刻}$$

$$P_k = F P_{k-1} F^T + Q \quad \text{用 } k-1 \text{ 的 Uncertainty}$$

System Noise  $\downarrow$  预测  $k$  时刻

用新 Measurement 更新 state 前。Prediction 是用于对齐 state 和 measurement (X 加 Z)

### Observation Update 当有新 Measurement

Kalman Gain (  $P \gg R, H \rightarrow$  state & measure )

$$\hat{x}_k = \bar{x}_k + K (\bar{z}_k - H \bar{x}_k) \quad \text{Innovation } y$$

$$P_k = (I - K H) \hat{P}_k = P_k^\wedge - K H \hat{P}_k^\wedge \quad \text{(Reduced)}$$

$$K = P_k^\wedge H^T (H \hat{P}_k^\wedge H^T + R)^{-1} \quad \text{Inno Covariance } S$$

# Particle Filters

Filters 对比	State Space	Belief	Efficiency	Solutions
Histogram Filters	Discrete	Multimodal	Exponential	Approximate World is not Discrete
Kalman Filters	Continuous	Unimodal	Quadratic	Approximate
Particle Filters	Continuous	Multimodal	不确定 有时 Exponential. 特点以上不准确	Approximate World is nonlinear Tracking Domain 效果更好

## Global Localization

- Robot 对自己的位置一无所知
- 只能依靠 sensory measurement 定位

## Particles

- 和 Robot 本身相同的 Class

### 初始化

- 生成 N 个 Particles

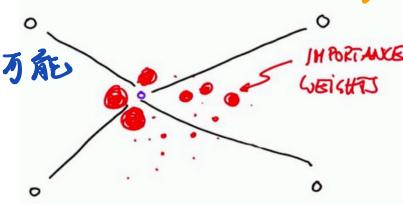
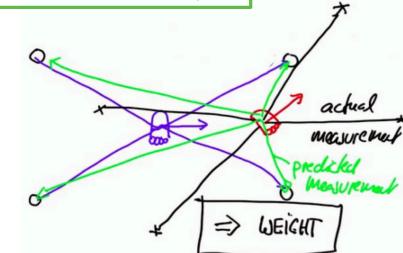
- 每个 Particle 都可以测量一个 measurement

### 计算 weight

通过对比 Particle 和 Robot 的 measurement 计算 Probability

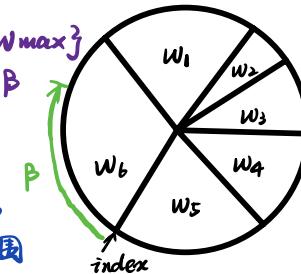
$$Gaussian(Particle\_measurement, \sigma, Robot\_measurement) \rightarrow Predicted\ measurements$$

- 这个概率和 Importance Weight 挂钩
- 离正确的位置越近，这个 Particle 给出的 measurement 越可能
- Resample 时越 important 的 Particle 越能存活下来
- 有放回抽样 (with replacement) → 重要的可抽取多次
- 使得 Particles 在 Posterior Probability 更高的区域聚集



## Resample Wheel

- Guess a Particle Index  $index = U[1 \dots N]$
- 建立一个 Function  $\beta$ .  $\beta = 0$ . for  $i$  in  $[1, N]$ :  $\beta \leftarrow \beta + w[i] / \sum w$
- 若当前 Particle 的 weight 不足以到达  $\beta$  的末端  $w[index] < \beta$   
 $\rightarrow$  则  $\beta = \beta - w[index]$ ,  $index = index + 1$
- 直至  $w[index] \geq \beta$ . 将  $p[index]$  取出至新的 list 中  
weight 小的 particles 全部被跳过的. 留下的都是 weight 较大的
- 重复 N 次 生成 N 个新的 Particles, 聚拢在概率最大的位置周围



```

p3 = []
index = int(random.random() * N)
beta = 0.0
mw = max(w)
for i in range(N):
    beta += random.random() * 2.0 * mw
    while beta > w[index]:
        beta -= w[index]
        index = (index + 1) % N
    p3.append(p[index])
p = p3
  
```

Fuzzing 在 Resample 后的位置对 x, y 设置 Noise, 使生成的 Particles 更有可能与目标位置重合

## Recap The Math Behind It All

$$P(z|x)$$

$$P(x|z)$$

- Measurement Updates → Posterior over state, given a measurement update

给定 state 和 measurement Particle 的分布 Prior X

→ Important Weight 由  $P(z|x)$  表示

→ Particles with Important Weight 表示一种分布

给定 measurement z 和 state update Posterior x|z

→ Resample 将 Importance 加入 Particles 集合中

- Motion Updates → Posterior over distribution one time step later Posterior x'

$$P(x') = \sum P(x'|x) P(x)$$

转移概率的 convolution  $x'|x$  带有 noise 的 motion model 作用后生成的 Particles x'

## Kinematic Bicycle Model

### Parameters

- Control  $\alpha$  radian ↓

Steering Angle  $\alpha$

Forward Movement d

- Robot Pose  $x, y, \theta$  (theta)

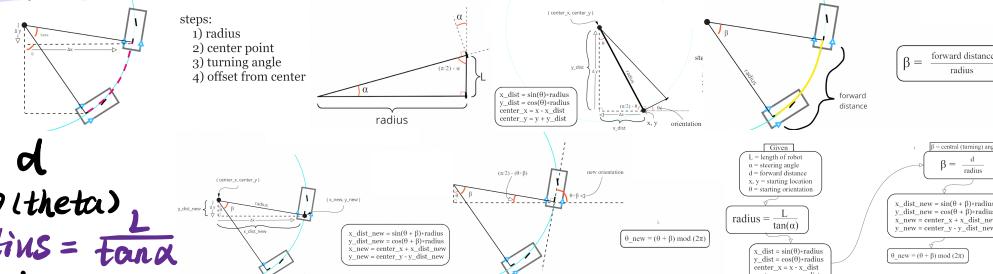
- Radius radius =  $\frac{L}{\tan \alpha}$

- Center Point (center x, center y)

- Turning Angle  $\beta = \text{arc length} / \text{radius} = \text{forward distance} / \text{radius}$

- Offset From Center ( $x_{new}, y_{new}$ )

New Orientation  $\theta_{new} = (\theta + \beta) \bmod(2\pi)$



$$\beta = \frac{\text{forward distance}}{\text{radius}}$$

$$\beta = \frac{d}{\text{radius}}$$

$$\beta = \frac{\text{control turning angle}}{\text{radius}}$$

$$\beta = \frac{d - \text{steering distance}}{\text{radius}}$$

$$\beta = \frac{d - \text{steering oscillation}}{\text{radius}}$$

$$\beta = \frac{d}{\tan(\alpha)}$$

## Search

### Motion Planning

- 计划找到目标 target
- Given: Map 所处的世界  
起始点, Starting Location  
目标位置 Goal Location

Cost 行驶某条 route 所花费的时间

- Goal: Find the minimum cost path

Parameters Search Program 中用到的一些参数定义

- Open list: 存放 nodes, 初始化为  $[(0, 0)]$   $[[g, x, y]]$

open.sort()

取出 open 里 g 值最小的作为下一个 expand 的 node  
并从 open 中删除这个 node

next = open.pop(0)

- delta List: 所有可能的 motion

- closed List: 和 Grid 大小相同

每个 cell 为 1 or 0

→ 检查该位置的 node 是否已被 expand  
该 node 是否已到达过

- found Flag: Search 是否完成 到达了 Goal

- resign Flag: 如果找不到可以 expand 的 nodes  
则重新分配 到达不了 Goal

- expand List: 保存该 cell 在哪一步被 expand 的信息

→ 和 Grid 的大小完全一致  $expand[x][y] = count$

→ expand 过的填步数, 没有 expand 过的填 -1

- action List: 每个 cell 中都记录是如何来到这个 cell 的  
记录的是每一个被 expand 的 cell 的 action, 并不是最终的 path

- policy List: 从 Goal 开始 Backtrack 找到 Final Path

→ 和 Grid 大小一致, 初始时全填 " " (空字符串)

→ 起点, 设为 Goal Node Goal's value 为 \*

→ 从 Goal Node 开始, 减去 action 中对应位置的值

从哪里来, 回到哪里去 → 找到来时的 Cell

→ 记录如何从上一个 cell 来到当前 cell, 并回到上一个 cell

→ 重复以上步骤, 直至回到起点

### A\* Search Algorithm (A star)

- Basic Principle: 使用最小的工作量最大化前往目标的进展

- Heuristic Function • 每个 cell 中有一个 value

全为 0 → 和初始算法相同

- 对与 Goal 之间距离的最乐观的估计

$h(x, y) \leq \text{actual goal distance from } x, y$

→ 为没有障碍时, 到达 Goal's Distance

- 不需要准确 比真实情况更加乐观

- f-value  $f = g + h(x, y)$  不再 expand g-value 最小的 node

而是 expand f-value 最小的 node

→ 在实际 Path 外的任何内容都不会被 Expand

### Dynamic Programming

- 给定 map 和 goal, 不限于固定的 Starting Location

- 为每个 cell 都提供一个最优的 action (Policy)  $x, y \rightarrow action$

Value Function  $f(x, y) \rightarrow$  与每个 cell 到达 Goal 最短 Path 的长度相关

### Search Program Breadth-first Planning

- 检查当前 node 是否是 goal
- 将当前 node 从 open 中取出, 并 expand
- Expand 具有最小的 g-value 的 node

$[2, 0], [1, 1]$  的 g-value  $\rightarrow 2$

- Expand 该 node 的后续 nodes

从 open 中删除当前 node

- 记下到达 goal 在了多少次 Expand g-value

Planning 结束时, g-value 为 path 的长度

→ open 是空的 → 没找到 Goal, 且没有 node 可以展开, return "Fail"  
→ 否则将找到 open 中 g 值最小的 node 进行 expand

```
if x == goal[0] and y == goal[1]:
```

found = True

else:

for i in range(len(delta)):

x2 = x + delta[i][0] 检查 • 是否还在 Grid 中

y2 = y + delta[i][1] • 该位置是否已被 expand 与能否通行

if x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 < len(grid[0]):

if closed[x2][y2] == 0 and grid[x2][y2] == 0:

g2 = g + cost g-value ← 加上这一步的 cost = 1

open.append([g2, x2, y2])

closed[x2][y2] = 1 该 node 被 closed 为 1

+ 不是 EX [y]

action[x2][y2] = i 记录 action

count += 1

policy = [[' ' \* len(grid[0])] for i in grid]

x = goal[0]

y = goal[1]

policy[x][y] = '\*'

while x != init[0] or y != init[1]: 找到来时的 cell

x2 = x - delta[action[x][y]][0] 前一个 cell

y2 = y - delta[action[x][y]][1]

policy[x2][y2] = delta\_name[action[x][y]]

x = x2

记录如何来到当前 cell

y = y2 回到来时的 cell

	0	1	2	3	4	5
0	S ✓ ✓					
1	✓ ✓					
2						
3						
4						

0 1 2 3 4 5

0 S | | | | | |

1 | | | | | |

2 | | | | | |

3 | | | | | |

4 | | | | | |

g2 = g + cost

[4, 1]  $g=5, f=9$

[4, 2]  $f=6+3=9$

[3, 2]  $f=7+4=11$

[4, 3]  $f=7+2=9$

h2 = heuristic[x2][y2]

f2 = g2 + h2

open.append([f2, g2, h2, x2, y2])

closed[x2][y2] = 1

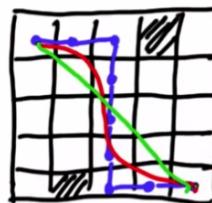
## PID

### Robot Motion

- 将 Path 转为控制 robot 的指令
  - 生成 smooth path
  - 控制 robot  $\rightarrow$  PID
- Smooth Path**

$\rightarrow$  从  $x_i$  开始,  $i=0 \dots n \rightarrow y_i$  (initialize 为  $x_i$ )

$$\rightarrow \min \|x_i - y_i\|^2, \|y_i - y_{i+1}\|^2$$



- $c_1 \gg 1 \rightarrow$  更接近原点 point
- $c_2 \gg 1 \rightarrow$  更 smooth
- $c_1 = 0 \rightarrow$  single point
- 若固定  $y_0$  和  $y_n \rightarrow$  straight line
- $c_2 = 0 \rightarrow y_i = x_i$  原 path

$$(y_{i+1} - y_i)^2$$

$$(x_{i+1} - x_i)^2$$

change = tolerance  
while (change >= tolerance):  
 change = 0  
 for i in range(1, len(path)-1):  
 for j in range(len(path[0])):  
 d1 = weight\_data \* (path[i][j] - newpath[i][j])  
 d2 = weight\_smooth \* (newpath[i-1][j] +  
 newpath[i+1][j] -  
 2 \* newpath[i][j])  
  
 change += abs(d1 + d2)  
 newpath[i][j] += d1 + d2

return newpath # Leave this line for the grader!

```
# Make a deep copy of path into newpath
newpath = [[0 for col in range(len(path[0]))]
           for row in range(len(path))]

for i in range(len(path)):
    for j in range(len(path[0])):
        newpath[i][j] = path[i][j]
```

### Gradient Descent

$$y_i = y_i + \alpha(x_i - y_i) + \beta(y_{i-1} - 2y_i + y_{i+1})$$

从  $y_i$  移向  $x_i$  ↓  
↓ 从  $y_i$  向  $y_{i-1}, y_{i+1}$  靠近  
 $\rightarrow$  同步 minimize  $\|x_i - y_i\|^2$  和  $\|y_i - y_{i+1}\|^2$   
 $\rightarrow$  只更新  $i=1 \dots n-1$ , 维持  $i=0, i=n$  不动  
 起点和终点, 必须和原先保持一致

## PID Control

给定 velocity 和 reference trajectory (smoother path)  $\rightarrow$  控制 robot 的 steering angle ( $\alpha$ )

P Controller 和 cross-track error (CTE) 成比例

$$\alpha = -\tau_p CTE_t$$

$\tau_p$  越大  $\rightarrow$  oscillate 越快

(P  $\Rightarrow$  Proportional) robot 和 reference trajectory 间的 Distance

- 和 system error 成比例
- 会与 reference trajectory 形成 overshoot 在 trajectory 上下 oscillate

$\rightarrow$  当 steering 是向前时, car 已经在 trajectory 上了, 即使  $\tau_p$  很小, 也会轻微 overshoot marginally stable

PD-Control 避免 overshoot 问题

- 不仅与 CTE 成比例, 还和  $\frac{dCTE}{dt}$  成比例

不仅会尝试靠近 trajectory, 还会注意到 error 已被减小  
 与 CTE 和当前及先前 step 间的 Difference 成比例

- 合理设置  $\tau_p$  和  $\tau_d$   $\rightarrow$  使 car 优先靠近 trajectory

- Systematic Bias e.g. car 的轮子不对齐 steering drift  
 $\rightarrow$  更大的 CTE Bias 发生在 steering 上, 但导致了 CTE 上升

PID Control PD 可以减小 Systematic Bias 但不能解决

- 朝目标 trajectory 转向更多以补偿 Bias  $\alpha = -\tau_p CTE_t - \tau_i \sum CTE_t - \tau_d \frac{d}{dt} CTE_t$

• 认别 sustained Bias  $\rightarrow \sum CTE_t$  Proportional Integral Differential

$\rightarrow$  随着 Bias 的累积而增加, 最终补偿并修正 motion

## Parameter Optimization

Twiddle  $\rightarrow$  每次只修改一个参数

P List : 目前为止最优的参数集合

dP List : 即将尝试的对参数的调整变化

### Steps

- 取 P 中的一个参数  $p[i]$
- 取 dP 中对应的  $dP[i] \rightarrow p[i] += dP[i]$
- 使用 P 运行 PID Planner, 若 error 更优  $\rightarrow dP[i] *= 1.1$   
 $\rightarrow$  否则  $\rightarrow p[i] -= 2 * dP[i]$
- 再次运行 PID Planner, 若 error 更优  $\rightarrow dP[i] *= 1.1$   
 $\rightarrow$  否则  $\rightarrow p[i] += dP[i]$
- iterate  $2 * N$  steps  
 $\rightarrow$  前  $N$  步 CTE 忽略不计  $\rightarrow$  Controller 时间收敛

```
def twiddle(tol = 0.2): # Make this tolerance bigger if you're timing out!
    n_params = 3
    dparams = [1.0 for row in range(n_params)]
    params = [0.0 for row in range(n_params)]
    best_error = run(params)
    n = 0
    while sum(dparams) > tol:
        for i in range(len(params)):
            params[i] += dparams[i]
            err = run(params)
            if err < best_error:
                best_error = err
                dparams[i] *= 1.1
            else:
                params[i] -= 2.0 * dparams[i]
                err = run(params)
                if err < best_error:
                    best_error = err
                    dparams[i] *= 1.1
                else:
                    params[i] += dparams[i]
                    dparams[i] *= 0.9
```

value List: 和 Grid 大小相同, 初始化为 99 与实际 value 不冲突 与该 cell 到 Goal 的距离正相关  
 change Flag: 初始化为 True, Loop 内部设为 False  
 value 中有变化时 设为 True

### Steps

按一定顺序遍历所有 cell

- 是否是 Goal Cell

是 → 是否  $value > 0 \rightarrow$  将 value 设为 0, change 设为 1

不是 → 遍历所有 action 并预测 next cell

- next cell 是否有效 不需要 Backtrack, 本身就是一个 Backtrack

计算新 value  $v_2$  next cell 的原 value + cost ←

若  $v_2 <$  现 cell 的 value → 将  $v_2$  赋予现 cell

说明这个 action 引起了 value to improve

→ 将这个 action 记录到现 cell

```

for x in range(len(grid)):
    for y in range(len(grid[0])):
        if goal[0] == x and goal[1] == y: 确定所有 Grid 中的 Cell
            if value[x][y] > 0:
                value[x][y] = 0
                policy[x][y] = '*'
                change = True
        elif grid[x][y] == 0: 是否是无障碍的 Cell
            for a in range(len(delta)): 遍历所有 action
                x2 = x + delta[a][0]
                y2 = y + delta[a][1]
               采取 action 后是否到达了有效 cell If x2 >= 0 and x2 < len(grid) and y2 >= 0 and y2 <
                len(grid[0]) and grid[x2][y2] == 0: 计算 new value
                v2 = value[x2][y2] + cost_step
                if v2 < value[x][y]: 该 action 使得从当前 cell
                    change = True 向 Goal 更进一步
                    value[x][y] = v2
                    policy[x][y] = delta_name[a]

for i in range(len(value)):
    print policy[i]
    
```

### Recap

#### Localization

QUIZ	Multi-modal?	Exponential?	Useful?
Kalman			YES
Histogram	YES	YES	YES
Particle	YES	YES	YES

#### Controller PID

QUIZ	Avoiding overshoot	Minimizing error	Compensating drift
P		YES	
I			YES
D	YES		

### Planning

QUIZ	Continuous	Optimal	Universal	Local
Breadth First		YES		
A-star		YES		
Dynamic Programming		YES	YES	
Smoothing	YES			YES

### SLAM Simultaneous Localization And Mapping

→ 一种 Mapping 的方法 以便未知 map 时完成 localization

→ 不仅要明白环境是不确定的, 还要知道 Robot 在一个不确定的轨迹上运动进行 mapping 以及同时进行定位

#### Graph SLAM

- 根据一系列的 constraint 计算概率 获取 constraints, 根据沿途的 landmarks 找到最可能的路径配置  
 → Initial Location Constraint Robot 的初始位置

→ Relative Motion Constraints 当前 pose 和先前 pose 的关联

→ Relative Measurement Constraints 从 Robot 的当前 pose 可以测量某些 landmarks

- 每个 Observation → 矩阵中四个元素间的加法 (Additions)

→ 代表起点, 其 pose 为正

→ 代表终点, 其 pose 为正

→ 对角线上的元素一定为正, 同行非对角线元素一定为负

只会影响由 source 和 destination 构成的子矩阵, 和对应的  $x_i$  元素

都是局部约束 (local constraints) motion 连接两个 location, measurement 连接 location to landmark

- Omega &  $x_i$

$$(S_2) \quad (\Sigma) \rightarrow \mu = S^{-1} \Sigma$$