Lecture 3, Jan 7, 2026

Mathematical Background

- There is no representation of rotations that has exactly 3 parameters (and therefore no constraints) and is also free of singularities
- Rotation representations include axis-angle (4 parameters including a unit length $\boldsymbol{a}$ and angle ϕ), quaternions (4 parameters subject to unit length), and Gibbs vector $\boldsymbol{g} = \boldsymbol{a}\tan\frac{\phi}{2}$ (3 parameters but singularity at $\phi = \pi$)
  - $\boldsymbol{\varepsilon} = \boldsymbol{a}\sin\frac{\phi}{2}, \eta = \cos\frac{\phi}{2}$ to convert between axis angle and quaternions
- For small perturbations, we have approximately $\boldsymbol{C} = 1 - \boldsymbol{\theta}^\times$ where $\boldsymbol{\theta} = \phi\boldsymbol{a}$
  - In this case the product of the principal rotations (Euler angles) is approximately their sum
  - This is used a lot in optimization
- Time derivatives of vectors seen in different frames is related as $\dot{\underrightarrow{r}} = \underrightarrow{r}^\circ + \underrightarrow{\omega}_{21} \times \underrightarrow{r} \iff \dot{\boldsymbol{r}}_1$
  - In terms of coordinates, $\boldsymbol{C}_{12}(\dot{\boldsymbol{r}}_2 + \boldsymbol{\omega}_2^{21\times}\boldsymbol{r}_2)$
  - We often have $\underrightarrow{r}$ denoting some position, $\underrightarrow{\mathcal{F}}_1$ being an inertial frame and $\underrightarrow{\mathcal{F}}_2$ being a moving frame (e.g. vehicle frame)
- We can show that the rotation matrix obeys Possion's equation: $\dot{\boldsymbol{C}}_{21} = -\boldsymbol{\omega}_2^{21\times}\boldsymbol{C}_{21}$
  - This means we can obtain the rotation matrix in a navigation scenario by integrating the equation, with $\boldsymbol{\omega}$ obtained from a sensor attached to the vehicle
- Consider a point P, which is $\underrightarrow{r}^{pi}$ in frame i and $\underrightarrow{r}^{pv}$ in frame v; given the pose of frame v, $\{\boldsymbol{r}_i^{vi}, \boldsymbol{C}_{iv}\}$, the vectors can be related as $\boldsymbol{r}_i^{pi} = \boldsymbol{C}_{iv}r_v^{pv} + r_i^{vi}$
  - $\boldsymbol{r}_a^{bc}$ denotes the coordinates of the vector from c to b, expressed in frame a
  - Notice that the indices of the translation $\boldsymbol{r}_i^{vi}$ look reversed, since we need to add the coordinates of frame v relative to frame i to get from v to i
- As a homogeneous transformation, $\begin{bmatrix}\boldsymbol{r}_i^{pi}\\1\end{bmatrix} = \begin{bmatrix}\boldsymbol{C}_{iv} & \boldsymbol{r}_i^{vi}\\\boldsymbol{0}^T & 1\end{bmatrix}\begin{bmatrix}\boldsymbol{r}_v^{pv}\\1\end{bmatrix} = \boldsymbol{T}_{iv}\begin{bmatrix}\boldsymbol{r}_v^{pv}\\1\end{bmatrix}$
  - Note $\boldsymbol{T}_{iv}^{-1} = \begin{bmatrix}\boldsymbol{C}_{iv} & \boldsymbol{r}_i^{vi}\\\boldsymbol{0}^T & 1\end{bmatrix}^{-1} = \begin{bmatrix}\boldsymbol{C}_{iv}^T & -\boldsymbol{C}_{iv}^T\boldsymbol{r}_i^{vi}\\\boldsymbol{0}^T & 1\end{bmatrix} = \begin{bmatrix}\boldsymbol{C}_{vi} & -\boldsymbol{r}_v^{vi}\\\boldsymbol{0}^T & 1\end{bmatrix} = \begin{bmatrix}\boldsymbol{C}_{vi} & \boldsymbol{r}_v^{iv}\\\boldsymbol{0}^T & 1\end{bmatrix} = \boldsymbol{T}_{vi}$
- The generalization of angular velocity for poses is $\boldsymbol{\varpi}_v^{vi} = \begin{bmatrix}\boldsymbol{\nu}^{vi}\\\boldsymbol{\omega}_v^{vi}\end{bmatrix}$, consisting of the linear and angular velocities
  - $\dot{\boldsymbol{T}}_{vi} = \begin{bmatrix}\boldsymbol{\omega}_v^{vi\times} & -\boldsymbol{\nu}_v^{vi}\\\boldsymbol{0}^T & 0\end{bmatrix}\boldsymbol{T}_{vi}$

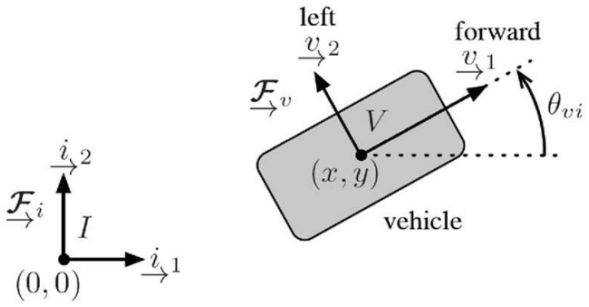

Figure 1: Illustration of the unicycle model.

- Example: we can derive the unicycle model by considering a 2D robot with position (x, y) and angle θ_{vi}, where axis z comes out of the plane

- $\boldsymbol{r}_i^{vi} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}, \boldsymbol{C}_{vi} = \boldsymbol{C}_3(\theta_{vi}) = \begin{bmatrix} \cos\theta_{vi} & \sin\theta_{vi} \\ 0 & -\sin\theta_{vi} \end{bmatrix} \cos\theta_{vi} 0001$
  - Note the notation for elementary rotations is different; $\boldsymbol{C}_3(\theta)$ does not denote a rotational transformation by θ, rather it denotes the transformation from the current frame to a frame obtained by rotating by θ, so the formula for the rotation matrices is transposed compared to the usual ones
- For the pose we want $\boldsymbol{T}_{iv} = \begin{bmatrix} \boldsymbol{C}_{iv} & \boldsymbol{r}_i^{vi} \\ \boldsymbol{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta_{vi} & -\sin\theta_{vi} & 0 & x \\ \sin\theta_{vi} & \cos\theta_{vi} & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
  - Note this is not $\boldsymbol{T}_{vi}$, so we use $\boldsymbol{C}_{iv}$ and not $\boldsymbol{C}_{vi}$!
  - Usually the third row and column is omitted for 2D
- For kinematics we constrain $\boldsymbol{\varpi}_v^{vi} = \begin{bmatrix} v & 0 & 0 & 0 & 0 & \omega \end{bmatrix}^T$, i.e. only forward movement and rotation along z
- Using the relation between $\dot{\boldsymbol{T}}_{iv}$ and $\boldsymbol{\varpi}_v^{vi}$ we get $\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \end{cases}$

Lecture 4, Jan 12, 2026

Wheel Differential Kinematics

- For the standard wheel, we assume rolling without slipping, so $v_x = \dot{\varphi} r$ and $v_y, v_z = 0$

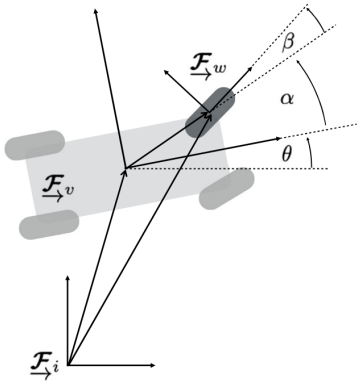

Figure 2: Derivation of the standard wheel model.

- Consider a vehicle with heading θ, with a wheel at angle α and distance $\|\underset{\rightarrow}{r}^{wv}\| = d$ relative to vehicle frame, with steering angle β
  - $\underset{\rightarrow}{r}^{wi} = \underset{\rightarrow}{r}^{vi} + \underset{\rightarrow}{r}^{wv} \implies \boldsymbol{v}_w^{wi} = \boldsymbol{v}_w^{vi} + \boldsymbol{\omega}_w^{vi\times} \boldsymbol{r}_w^{wv}$ in the wheel frame
  - This becomes $\begin{bmatrix} \dot{\varphi} r \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{C}_3(\alpha+\beta)\boldsymbol{v}_v^{vi} + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix}^\times \begin{bmatrix} d\cos\beta \\ -d\sin\beta \\ 0 \end{bmatrix}$
  - Simplify: $\begin{bmatrix} \dot{\varphi} r \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \cos(\alpha+\beta) & \sin(\alpha+\beta) & 0 \\ -\sin(\alpha+\beta) & \cos(\alpha+\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{v}_v^{vi} + \begin{bmatrix} d\sin\beta \\ d\cos\beta \\ 0 \end{bmatrix} \dot{\theta}$
    - The 3 equations express the rolling without sliding, no sideways sliding, and contact with ground constraints

– Let $\dot{\boldsymbol{q}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$ be the pose rate in inertial frame, so in vehicle frame $\dot{\boldsymbol{\xi}} = \begin{bmatrix} v \\ u \\ \omega \end{bmatrix} = \boldsymbol{C}_3(\theta)\dot{\boldsymbol{q}}$ (where $\omega = \dot{\theta}$)
  * $\begin{bmatrix} \cos(\alpha+\beta) & \sin(\alpha+\beta) & d\sin\beta \end{bmatrix} \dot{\boldsymbol{\xi}} = \dot{\varphi} r$
  * $\begin{bmatrix} -\sin(\alpha+\beta) & \cos(\alpha+\beta) & d\cos\beta \end{bmatrix} \dot{\boldsymbol{\xi}} = 0$

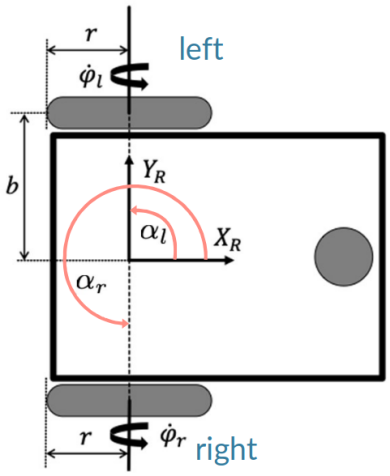

Figure 3: Differential drive robot model.

- Example: differential drive model
  – $\begin{bmatrix} \cos(\alpha_r+\beta_r) & \sin(\alpha_r+\beta_r) & d_r\sin\beta_r \\ \cos(\alpha_l+\beta_l) & \sin(\alpha_l+\beta_l) & d_l\sin\beta_l \\ -\sin(\alpha_r+\beta_r) & \cos(\alpha_r+\beta_r) & d_r\cos\beta_r \\ -\sin(\alpha_l+\beta_l) & \cos(\alpha_l+\beta_l) & d_l\cos\beta_l \end{bmatrix} \begin{bmatrix} v \\ u \\ \omega \end{bmatrix} = \begin{bmatrix} \dot{\varphi}_r r \\ \dot{\varphi}_l r \\ 0 \\ 0 \end{bmatrix}$
  – Substituting and simplifying: $\begin{bmatrix} 1 & 0 & b \\ 1 & 0 & -b \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} v \\ u \\ \omega \end{bmatrix} = \begin{bmatrix} \dot{\varphi}_r r \\ \dot{\varphi}_l r \\ 0 \\ 0 \end{bmatrix}$
    * These are known as the *differential kinematics* of the robot, relating the body-centric velocity to the wheel speeds
    * e.g. If $\dot{\varphi}_r = \dot{\varphi}_l$, we get $\omega = 0$ which intuitively makes sense
  – Solving for wheel rates gives us *inverse differential kinematics*: $\begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \end{bmatrix} = \frac{1}{r}\begin{bmatrix} 1 & b \\ 1 & -b \end{bmatrix}\begin{bmatrix} v \\ \omega \end{bmatrix}$
  – Solving for vehicle speed gives us *forward differential kinematics*: $\begin{bmatrix} v \\ \omega \end{bmatrix} = \frac{1}{2}\begin{bmatrix} r & r \\ r/b & -r/b \end{bmatrix}\begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \end{bmatrix}$

- Swedish wheels (or Mecanum wheels) have rollers on the wheel which allows for sideways motion at an angle γ
  – Following the same derivation gives us the following set of constraints:
    * $\begin{bmatrix} \cos(\alpha+\beta+\gamma) & \sin(\alpha+\beta+\gamma) & d\sin(\beta+\gamma) \end{bmatrix} \dot{\boldsymbol{\xi}} = \dot{\varphi} r\cos\gamma + \dot{\varphi}_s r_s$
    * $\begin{bmatrix} -\sin(\alpha+\beta+\gamma) & \cos(\alpha+\beta+\gamma) & d\cos(\beta+\gamma) \end{bmatrix} \dot{\boldsymbol{\xi}} = -\dot{\varphi} r\sin\gamma$
    * Note we can recover the standard wheel model by simply setting $\dot{\varphi}_s = 0$
  – Note since the small wheels are passive, $\dot{\varphi}_s$ can be anything, so the first equation does not constrain the motion and just acts as another degree of freedom, i.e. we usually only have the lateral constraint

- Example: vehicle with 4 Swedish wheels
  – Using the first configuration and expanding the lateral constraints (since the rollers are unconstrained):

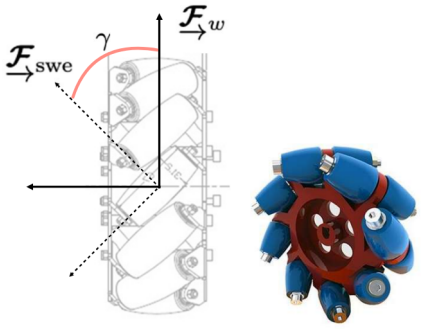

Figure 4: Swedish wheel.

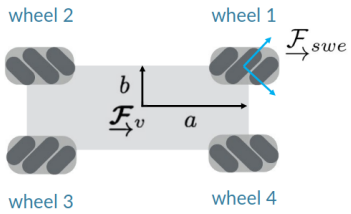

Figure 5: Swedish wheel vehicle.

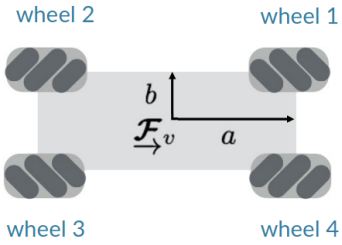

Figure 6: Swedish wheel configuration without degeneracy (bottom-up view).

* $\begin{bmatrix} 1 & 1 & -(b-a) \\ 1 & -1 & -(b-a) \\ 1 & 1 & b-a \\ 1 & -1 & b-a \end{bmatrix} \begin{bmatrix} v \\ u \\ \omega \end{bmatrix} = \begin{bmatrix} \dot{\varphi}_1 r \\ \dot{\varphi}_2 r \\ \dot{\varphi}_3 r \\ \dot{\varphi}_4 r \end{bmatrix}$
  * Notice that in the case of $a = b$, the last column is cleared out and we no longer have control over ω; intuitively this is because when the wheels are symmetric about the centre, the vehicle can be rotated freely regardless of wheel rotation
- Using the second configuration we can avoid the degeneracy:
  * $\begin{bmatrix} \dot{\varphi}_1 \\ \dot{\varphi}_2 \\ \dot{\varphi}_3 \\ \dot{\varphi}_4 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} 1 & -1 & -(a+b) \\ 1 & 1 & -(a+b) \\ 1 & -1 & (a+b) \\ 1 & 1 & (a+b) \end{bmatrix} \begin{bmatrix} v \\ u \\ \omega \end{bmatrix}$
- Since we can individually control all 4 wheels but the vehicle only has 3 degrees of freedom, the forward kinematics are not unique; we can use the psuedoinverse to recover the forward model:
  * $\begin{bmatrix} v \\ u \\ \omega \end{bmatrix} = \frac{r}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -\frac{1}{a+b} & -\frac{1}{a+b} & \frac{1}{a+b} & \frac{1}{a+b} \end{bmatrix} \begin{bmatrix} \dot{\varphi}_1 \\ \dot{\varphi}_2 \\ \dot{\varphi}_3 \\ \dot{\varphi}_4 \end{bmatrix}$

Lecture 5, Jan 12, 2026

Vehicle Models

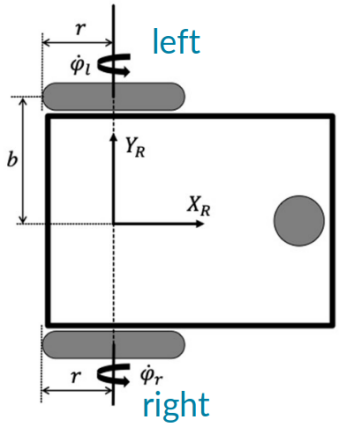

Figure 7: Differential drive robot.

- Consider the unicycle model kinematics we derived previously for differential drive (note we redefine b to be half the distance between wheels) $\begin{bmatrix} v \\ \omega \end{bmatrix} = \frac{1}{2} \begin{bmatrix} r & r \\ r/2b & -r/2b \end{bmatrix} \begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \end{bmatrix}$
  - Define the inertial frame configuration $\boldsymbol{q} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$
  - To handle the nonholonomic constraint, consider $\dot{\boldsymbol{q}}$ in the vehicle frame, since we are constrained to have $u = 0$
    * $\dot{\boldsymbol{q}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ 0 \\ \omega \end{bmatrix}$
  - The overall kinematics in inertial frame is $\dot{\boldsymbol{q}} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r/2 & r/2 \\ r/2b & r/2b \end{bmatrix} \begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \end{bmatrix} = \boldsymbol{G}(q)\boldsymbol{p}$

- Generally we can write the nonholonomic constraints in matrix form as $\boldsymbol{H}(q)^T\dot{\boldsymbol{q}} = 0$, so the admissible velocities consists of the null space of $\boldsymbol{H}(\boldsymbol{q})^T$
  - Let the generalized velocity $\boldsymbol{p}$, then the null space of $\boldsymbol{H}(\boldsymbol{q})^T$ is $\dot{\boldsymbol{q}} = \boldsymbol{G}(\boldsymbol{q})\boldsymbol{p}$ where $\boldsymbol{H}(\boldsymbol{q})^T\boldsymbol{G}(\boldsymbol{q}) = \boldsymbol{0}$
  - For the unicycle model, $\boldsymbol{G}(\boldsymbol{q}) = \begin{bmatrix}\cos\theta & 0\\ \sin\theta & 0\\ 0 & 1\end{bmatrix}$
- To model the vehicle dynamics as well, we usually use Euler-Lagrange: $\dfrac{\mathrm{d}}{\mathrm{d}t}\left(\dfrac{\partial^T L}{\partial\dot{\boldsymbol{q}}}\right) - \dfrac{\partial^T L}{\partial \boldsymbol{q}} + \boldsymbol{\tau} + \boldsymbol{H}(\boldsymbol{q})\boldsymbol{\lambda}$ where the $\boldsymbol{H}(\boldsymbol{q})\boldsymbol{\lambda}$ models nonholonomic constraints, with $\boldsymbol{\lambda}$ being the Lagrange multipliers
  - The generalized forces can be expressed as $\boldsymbol{\tau} = \boldsymbol{G}(\boldsymbol{q})\boldsymbol{\nu}$ so that the nonholonomic constraints are satisfied
- For the unicycle model:
  - $T = \dfrac{1}{2}m(\dot{x}^2 + \dot{y}^2) + \dfrac{1}{2}I\dot{\theta}^2, V = 0$
  - The Lagrange multiplier in this case represents the constraint forces arising from the vehicle kinematics (i.e. force from the wheel no-slip constraints), but we don't know this force so we try to eliminate it
  - $\dfrac{\mathrm{d}}{\mathrm{d}t}\left(\dfrac{\partial^T L}{\partial\dot{\boldsymbol{q}}}\right) = \begin{bmatrix}m\ddot{x}\\ m\ddot{y}\\ I\ddot{\theta}\end{bmatrix}, \dfrac{\partial^T L}{\partial \boldsymbol{q}} = \boldsymbol{0}, \boldsymbol{\tau} = \boldsymbol{G}(\boldsymbol{q})\boldsymbol{\nu}$
  - To eliminate the unknown Lagrange multiplier, premultiply the EL equation by $\boldsymbol{G}(\boldsymbol{q})^T$:
    * $$\boldsymbol{G}(\boldsymbol{q})^T\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial^T L}{\partial\dot{\boldsymbol{q}}}\right) - \boldsymbol{G}(\boldsymbol{q})^T\frac{\partial^T L}{\partial \boldsymbol{q}} = \boldsymbol{G}(\boldsymbol{q})^T\boldsymbol{\tau} + \boldsymbol{G}(\boldsymbol{q})^T\boldsymbol{H}(\boldsymbol{q})\boldsymbol{\lambda}$$
      $$\implies \begin{bmatrix}\cos\theta & \sin\theta & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}m\ddot{x}\\ m\ddot{y}\\ I\ddot{\theta}\end{bmatrix} = \boldsymbol{\nu}$$
    * Note in this case $\boldsymbol{G}(q)^T\boldsymbol{G}(q) = \boldsymbol{1}$, but this is not true in general
  - Recall the generalized velocity is $\boldsymbol{p} = \begin{bmatrix}v\\ \omega\end{bmatrix} \implies \dot{\boldsymbol{p}} = \begin{bmatrix}\ddot{x}\cos\theta + \ddot{y}\sin\theta\\ \ddot{\theta}\end{bmatrix}$
  - Therefore $\begin{bmatrix}m & 0\\ 0 & I\end{bmatrix}\begin{bmatrix}\dot{v}\\ \dot{\omega}\end{bmatrix} = \boldsymbol{\nu} \iff \boldsymbol{M}\dot{\boldsymbol{p}} = \boldsymbol{\nu}$
  - The complete model is $\begin{bmatrix}\dot{\boldsymbol{q}}\\ \dot{\boldsymbol{p}}\end{bmatrix} = \begin{bmatrix}\boldsymbol{0} & \boldsymbol{G}(\boldsymbol{q})\\ \boldsymbol{0} & -\boldsymbol{M}^{-1}\boldsymbol{D}\end{bmatrix}\begin{bmatrix}\boldsymbol{q}\\ \boldsymbol{p}\end{bmatrix} + \begin{bmatrix}\boldsymbol{0}\\ \boldsymbol{M}^{-1}\end{bmatrix}\boldsymbol{u}$
    * This uses a damped model for forces $\boldsymbol{\nu} = -\boldsymbol{D}\boldsymbol{p} + \boldsymbol{u}$ where $\boldsymbol{u} = \begin{bmatrix}f\\ g\end{bmatrix}$, f is some longitudinal thrust and g is some steering torque

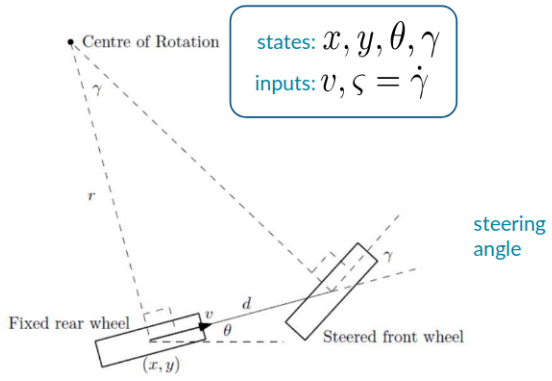

Figure 8: Bicycle model derivation.

- For the bicycle model:

- Define state $\begin{bmatrix} x \\ y \\ \theta \\ \gamma \end{bmatrix}$, generalized velocity $\boldsymbol{p} = \begin{bmatrix} v \\ \varsigma \end{bmatrix}$ (where ς is the steering velocity)
- Form the nonholonomic constraints by taking lateral slip constraints from the wheel model for both wheels, and find the nullspace of $\boldsymbol{H}(\boldsymbol{q})^T$ to get $\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\gamma} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ \frac{\tan\gamma}{d} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \varsigma \end{bmatrix} \iff \dot{\boldsymbol{q}} = \boldsymbol{G}(\boldsymbol{q})\boldsymbol{p}$
- Form the Lagrangian: $L = \frac{1}{2}(m_r + m_f)(\dot{x}^2 + \dot{y}^2) + \frac{1}{2}(I_r + m_f d^2)\dot{\theta}^2 + \frac{1}{2} I_f(\dot{\theta} + \dot{\gamma})^2$
$$= \frac{1}{2} m(\dot{x}^2 + \dot{y}^2) + \frac{1}{2} I\dot{\theta}^2 + \frac{1}{2} I_f(\dot{\theta} + \dot{\gamma})^2$$
  - Note this comes from the kinetic energy of the forward and rear wheels combined
- Now we have $\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial^T L}{\partial \dot{\boldsymbol{q}}}\right) = \begin{bmatrix} m\ddot{x} \\ m\ddot{y} \\ I\ddot{\theta} + I_f(\ddot{\theta} + \ddot{\gamma}) \\ I_f(\ddot{\theta} + \ddot{\gamma}) \end{bmatrix} = \boldsymbol{M}\ddot{\boldsymbol{q}}$, $\frac{\partial^T L}{\partial \boldsymbol{q}} = \boldsymbol{0}, \boldsymbol{\tau} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{\nu} = \boldsymbol{F}(\boldsymbol{q})\boldsymbol{\nu}$
  - $\boldsymbol{M} = \begin{bmatrix} m & 0 & 0 & 0 \\ 0 & m & 0 & 0 \\ 0 & 0 & I + I_f & I_f \\ 0 & 0 & I_f & I_f \end{bmatrix}$
  - Note $\boldsymbol{\tau}$ is somewhat arbitrary here; the matrix $\boldsymbol{F}(\boldsymbol{q})$ is not necessarily related to $\boldsymbol{G}(\boldsymbol{q})$, and instead is just based on what we choose
- Premultiply by $\boldsymbol{G}(\boldsymbol{q})^T$ to eliminate the Lagrange multiplier again to get $\dot{\boldsymbol{p}} = \boldsymbol{M}(\boldsymbol{q})^{-1}(-(\boldsymbol{G}(q)^T\boldsymbol{M}\dot{\boldsymbol{G}}(\boldsymbol{q}) + \boldsymbol{D})\boldsymbol{p} + \boldsymbol{u}) = \boldsymbol{M}(\boldsymbol{q})^{-1}(-\boldsymbol{D}(\boldsymbol{q}, \boldsymbol{p})\boldsymbol{p} + \boldsymbol{u})$
  - Note $\boldsymbol{M}(\boldsymbol{q}) = \boldsymbol{G}(\boldsymbol{q})^T\boldsymbol{M}\boldsymbol{G}(\boldsymbol{q})$
  - We again used damped forces $\boldsymbol{\nu} = -\boldsymbol{D}\boldsymbol{p} + \boldsymbol{u}$
- Complete system model: $\begin{bmatrix} \dot{\boldsymbol{q}} \\ \dot{\boldsymbol{p}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{G}(\boldsymbol{q}) \\ \boldsymbol{0} & -\boldsymbol{M}(\boldsymbol{q})^{-1}\boldsymbol{D}(\boldsymbol{q}, \boldsymbol{p}) \end{bmatrix} \begin{bmatrix} \boldsymbol{q} \\ \boldsymbol{p} \end{bmatrix} + \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{M}(\boldsymbol{q})^{-1} \end{bmatrix} \boldsymbol{u}$ In general all vehicle models can be written as a nonlinear differential equation $\dot{\boldsymbol{x}} = \boldsymbol{A}(\boldsymbol{x})\boldsymbol{x} + \boldsymbol{B}(\boldsymbol{x})\boldsymbol{u}$ where $\boldsymbol{x} = \begin{bmatrix} \boldsymbol{q} \\ \boldsymbol{p} \end{bmatrix}$ consisting of the configuration (pose, etc) and generalized velocity
- If we ignore dynamics, then $\boldsymbol{A} = 0$

- Additional constraints can be added to the model to limit $\{ \boldsymbol{x}, \boldsymbol{u} \}$ to an allowed set S_{allowed}, e.g. turning rates or obstacles
  - These constraints are in general non-convex and makes optimization much harder
- Example: imposing a curvature constraint on the unicycle model
  - Define the curvature $k = \frac{\omega}{v} = \frac{1}{b}\frac{\dot{\varphi}_r - \dot{\varphi}_l}{\dot{\varphi}_r + \dot{\varphi}_l}$ and radius of curvature $R = \frac{1}{|k|}$
  - A max curvature constraint would impose $|\frac{\omega}{v}| = |k| \leq k_{\max} = \frac{1}{R_{\min}}$

Lecture 6, Jan 14, 2026

Path-Tracking Control

- Given a reference path $\boldsymbol{x}_d$, we want to design controllers which output a vehicle command $\boldsymbol{u}$ input to the robot, which results in the actual state $\boldsymbol{x}$ subject to a disturbance $\boldsymbol{w}$; we measure the state with sensors which produce readings $\boldsymbol{y}$, corrupted by noise $\boldsymbol{n}$, and a state estimator produces the estimate $\hat{\boldsymbol{x}}$ used in the controller
  - For now we take sensing and state estimation for granted, i.e. $\hat{\boldsymbol{x}} = \boldsymbol{x}$
- This problem is complicated by the fact that our vehicle models are usually nonlinear (e.g. differential drive) and are working with a MIMO system

- Note that *trajectory control* involves commanding the robot to a specific state at a specific time, while *path-Tracking control* cares only about how the robot gets to the state and not the time
  - Trajectory control is more useful for e.g. synchronized drone swarms
  - When trajectory control falls behind it needs to cut corners to catch up, which is bad since it might hit hazards in the way

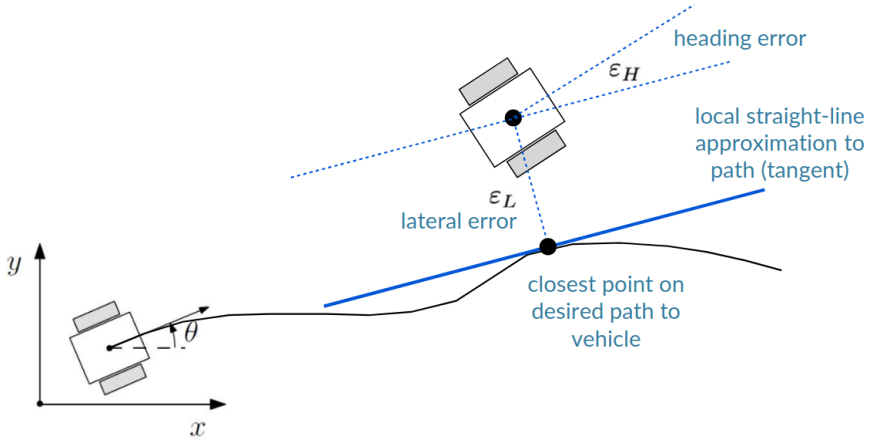

Figure 9: Lateral and heading error.

- The path can be approximated as locally straight, allowing us to define a *crosstrack error* $\varepsilon_L = y_d - y$ (i.e. lateral deviation) and a heading error $\varepsilon_H = \theta_d - \theta$ for the robot, which we assume can be measured (e.g. through odometry)
  - We define x to point along the path and y perpendicular to it

Feedback Linearization

- The first approach is *feedback linearization*, where we design a controller to cancel out the nonlinear dynamics, which is only possible if we know the dynamics of the system exactly
  - Assuming the motion is of the form $\dot{x} = f(x) + g(x)u$, we want to find an input mapping which maps a new input η to u, so that the overall plant with η as an input is linear
  - We want to find $u = a(x) + b(x)\eta$ such that $\dot{x} = f(x) + g(x)(a(x) + b(x)\eta) = Ax + B\eta$, i.e. find A and B such that $f(x) + g(x)a(x) = Ax, g(x)b(x)\eta = B\eta$
  - This can be difficult and might not be possible for some systems
- Applied to the path tracking controller using the unicycle model $\dot{\boldsymbol{q}} = \begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos\theta & 0\\ \sin\theta & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}v\\ \omega\end{bmatrix}$
  - Assuming a constant velocity, $\begin{bmatrix}\dot{\varepsilon}_L\\ \dot{\varepsilon}_H\end{bmatrix} = \begin{bmatrix}v\sin\varepsilon_H\\ \omega\end{bmatrix}$
    - Note that $y_d, \theta_d = 0$ since we are working in the path frame, so the desired heading and lateral position are always zero
    - This can be interpreted as an implicit assumption that we're following a straight path (since the path is approximated as straight based on the closest point)
  - Apply a change of variables $\begin{bmatrix}z_1\\ z_2\end{bmatrix} = \begin{bmatrix}\varepsilon_l\\ v\sin\varepsilon_H\end{bmatrix}$, computing the new derivatives gives us the vehicle model $\begin{bmatrix}\dot{z}_1\\ \dot{z}_2\end{bmatrix} = \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}z_1\\ z_2\end{bmatrix} + \begin{bmatrix}0\\ 1\end{bmatrix}\eta$ where $\eta = v\omega\cos\varepsilon_H$, which is now linear
    - Note when performing the change of variables we need to ensure that the new variables go to zero when the original path errors go to zero
  - If we add an output $y = \begin{bmatrix}\alpha & 1\end{bmatrix}\begin{bmatrix}z_1\\ z_2\end{bmatrix}$, we now have a SISO system which we can design a controller for using classical methods; we can pick α as the relative importance of the heading and crosstrack

errors
    * This has transfer function $\frac{s+\alpha}{s^2}$ (which we converted from state-space form)
    * The overall closed-loop transfer function is $\frac{y(s)}{y_d(s)} = \frac{P(s)C(s)}{1+P(s)C(s)}$ where $C(s)$ is the controller transfer function, $P(s)$ is the plant transfer function
    * Using a simple proportional controller, $C(s) = K, \eta = Ke(t)$, giving us poles at $1+P(s)C(s) = \frac{s^2+Ks+\alpha K}{s^2} = 0$
    * To get e.g. two identical negative real poles, we can choose $K = 4\alpha$, which results in poles at -2α
  - The final controller has 2 tuning parameters, α and the constant speed v
  - To actually implement the controller, we need to compute z_1, z_2 from the errors, then y, then η according to our controller, and finally back to ω
  - When we make all substitutions, we get
$$\begin{aligned}\omega &= \frac{1}{v\cos\varepsilon_H}\eta \\ &= \frac{1}{v\cos\varepsilon_H}K(y_d - y) \\ &= -\frac{1}{v\cos\varepsilon_H}4\alpha(\alpha z_1 + z_2) \\ &= -\frac{1}{v\cos\varepsilon_H}4\alpha(\alpha\varepsilon_L + v\sin\varepsilon H) \\ &= -\frac{4\alpha^2}{v}\frac{\varepsilon_L}{\cos\varepsilon_H} - 4\alpha\tan\varepsilon_H\end{aligned}$$
    * This reveals that we need to ensure $v \neq 0$ and $\cos\varepsilon_H \neq 0$, which means that the controller cannot deal with pure rotations or 90 degree heading errors
    * The singularity also means that if we start with a heading error greater than 90 degrees, we can never recover
- Another issue with this controller is that it reacts to only the current lateral heading errors (since it only looks at the errors at the closest point on the path), so it has no way of looking forwards to correct for future errors
  - This results in overshoots at every turn
  - This can be partially addressed by considering path curvature or computing the error based on a point ahead of the vehicle

Important

In general, feedback linearization can hide singularities inside the change of variables as we've seen in the unicycle model example, which we need to watch out for.

Geometric Path Following Control

- For specific vehicle kinematic models, we can come up with specialized controllers for path control, e.g. the *pure pursuit* controller for unicycle and bicycle models, which simply chases a point at a fixed distance ahead along the desired path
- The *Stanley controller* for bicycle models can drive heading and crosstrack errors to zero without singularities (i.e. global convergence)

- For the Stanley controller, we calculate the tracking errors relative to the centre point of the front axle, which is more stable (as opposed to the kinematic centre at the rear axle)
- We have two inputs, steering rate δ and vehicle velocity v_f in the direction of front wheels
  - The heading error is computed about the instantaneous centre of rotation, which is defined by the steering angle δ
  - $\dot{\varepsilon}_H(t) = \frac{-v_f(t)\sin(\delta(t))}{l}$

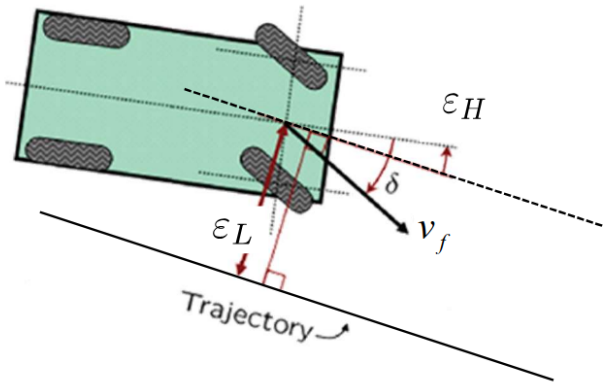

Figure 10: Stanley controller errors.

  - $\dot{\varepsilon}_L(t) = v_f(t)\sin(\varepsilon_H(t) - \delta(t))$
- The Stanley controller combines 3 requirements for steering for the combined law $\delta(t) = \varepsilon_H(t) + \tan^{-1}\left(\frac{k\varepsilon_L(t)}{v_f(t)}\right)$
  - Steer to align heading with desired heading, proportional to heading error: $\delta(t) = \varepsilon_H(t)$
  - Steer to eliminate crosstrack error, inversely proportional to speed and soft-capped with inverse tangent: $\delta(t) = \tan^{-1}\left(\frac{k\varepsilon_L(t)}{v_f(t)}\right)$
  - Keep steering angle between a maximum and minimum $\delta(T) \in [\delta_{\min}, \delta_{\max}]$
- The error dynamics can be derive as $\dot{\varepsilon}_L(t) = \dfrac{-k\varepsilon_L(t)}{\sqrt{1 + \left(\frac{k\varepsilon_L(t)}{v_f}\right)^2}}$
  - Notice that when the crosstrack error is small, this approximately results in exponential decay
- In practice, there are several improvements we can make:
  - Add a softening constant k_s to $v_f(t)$ to prevent aggressive steering when the vehicle is moving slowly
  - Extra damping can be applied to the heading term helps counteract effect of noise at higher speeds
  - We can also steer into constant radius curves using a feedforward term (i.e. have a term to steer to a specific angle if we know we're following a known curvature)

Model Predictive control

- Model predictive control (MPC) uses the full nonlinear model of the plant and a parametrized controller to predict the errors within a future window (the horizon)
  - At each step, we predict all future errors for some horizon, and find the optimal control input within the horizon that minimizes this error, then we apply only first step (or a short time), and iterate again
    * Hence MPC is also known as *receding horizon control* since we have this time horizon that keeps moving
  - The optimization is usually numerical, and minimizes a weighted sum of squared errors, with current errors often weighted higher than future errors
- In this case, the "controller" we're parametrizing is a series of control inputs $v(t), \omega(t)$
  - A simple version is the cubic polynomial: $\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} v_0 \\ \omega_0 \end{bmatrix} + \begin{bmatrix} v_1 \\ \omega_1 \end{bmatrix} t + \begin{bmatrix} v_2 \\ \omega_2 \end{bmatrix} t^2 + \begin{bmatrix} v_3 \\ \omega_3 \end{bmatrix} t^3$
  - This gives $\boldsymbol{u} = (v_0, \omega_0, v_1, \omega_1, v_2, \omega_2, v_3, \omega_3)$
- The full nonlinear model is used to forward simulate the system according to the sequence of inputs, which allows us to calculate the lateral error

- We can define a cost function $J(\boldsymbol{u}) = \frac{1}{2}\sum_i w_i \varepsilon_{L,i}^2 + \frac{1}{2}\sum_k \left(w_{v,k} v_k^2 + w_{\omega,k}\omega_k^2\right)$
  - The first term is the path tracking cost; we can choose the weights w_i to weigh current errors more than future ones
  - The second term acts like regularization which penalizes large control inputs (ridge regression)
- This can be solved for the optimal control inputs $\boldsymbol{u}^* = \underset{\boldsymbol{u}}{\operatorname{argmin}}\, J(\boldsymbol{u})$, which we apply for a short time and rerun the optimization

Lecture 7, Jan 19, 2026

Introduction to Planning and Graphs

- In general planning involves finding a path between two configuration states, i.e. answering queries about the connectivity of some space, subject to constraints and optimality criteria
- A classic manipulator planning approach was to find the entire allowed configuration space by mapping all the obstacles into the configuration space, and finding a path in space through only allowed regions
  - This is expensive and only feasible for static situations such as factory robots, which could be engineered in advance
  - Mobile robots cannot use this approach since their environments are much larger and typically not known in advance
  - The physical space is in general much easier to define, but the configuration space is much easier to plan in
- For simplicity we assume that our localization and mapping are perfect; planning under uncertainty is much more computationally expensive
- Planning often utilizes a two-tiered approach: a strategic/high-level (global) planner, planning an optimal route based on a priori knowledge of the environment (e.g. satellite image and GPS), and a tactical/low-level planner that handles real-time local planning for obstacle avoidance (reactive methods)
  - This leads to a standard workflow where we first compute a simple path, smooth the path to account for kinematics, design a trajectory to account for dynamics (i.e. adding a velocity profile), and designing a controller to track the trajectory

Reactive Planning

- *Potential fields* is one of the simplest approaches for planning, based on finding the minimum of a potential field
  - The target potential attracts towards the goal: $V_{att}(q) = K_{att}\rho(q, q^g)^2$
    - q is the current configuration, q^g is the goal configuration
    - $\rho(q, q^g)^2$ measures the distance between the current configuration and the goal, often in physical space
  - Obstacle potentials repel the vehicle: $V_{rep}(q) = K_{rep}\sum_{i=1}^{n} \frac{1}{\rho(q, O^i)^2}$
    - $\rho(q, O^i)$ computes the shortest distance between q and obstacle i (more precisely the closest point on the obstacle surface)
    - We often impose a maximum range of influence so that outside of this distance the obstacle potential is zero
  - The total potential is the combination of the two fields: $V(x_t) = V_{att}(x_t) + V_{rep}(x_t)$
  - To find the path, use a gradient descent approach; at each step we move in the direction of steepest descent of the potential field, which can be found by taking the gradient $-\nabla V$
  - The path often looks like a straight movement towards the goal until the robot gets within the region of influence of an obstacle, and then it's pushed around the obstacle
  - The major weakness of potential fields is the existence of local minima in the potential function in complex scenarios, which the robot can get stuck in and cannot recover from
  - Potential fields are also non-optimal and do not consider dynamic constraints

- *Extended potential fields* adapt the method specifically for driving robots by incorporating the vehicle heading
  - This adds a rotation potential, with a dependence on bearing to obstacles so that the robot turns away from obstacles and also diminishes the influence of obstacles behind the robot
- *Trajectory rollout* is a very basic form of MPC that considers a discrete number of possible trajectories in a short time window
  - Consider n different inputs to apply, e.g. holding velocity constant and considering n different rotation rates
  - For each of the inputs, propagate the trajectory for some time T using a vehicle model, and check each trajectory for collisions
  - Out of the valid trajectories, score on criteria such as progress to goal, distance to obstacles, similarity to previous choice, etc.
  - If computing trajectories takes time, we need to predict where the starting point is by the time we're done planning, using the current inputs
  - Kinematic and dynamic constraints can be considered by constraining the input choices based on the current state at each step, e.g. restricting the possible changes to angular and translational velocity in one step
  - As a primarily local planner, trajectory rollout can still get stuck due to the very limited planning window

Global Planning

- The environment can be represented as a regular grid of traversable vs. non-traversable locations, producing an *occupancy map*, or irregularly sampled locations connected by traversibility, producing a *probabilistic roadmap*
  - PRMs have the advantage of being able to operate in physical space or configuration space, as long as we have a way to check whether two configurations are connected
- Many of these methods lead to graphs, with nodes representing allowable configurations and edges connecting them; once the graph is built, the problem simply becomes a shortest path search on a graph
- Graph-based planners should satisfy some properties:
  1. Completeness: In finite time, the planner should either produce a path or conclude that no path exists, i.e. a complete planner never misses a valid path
     - For grid-based planning, a *resolution complete* planner is guaranteed to find a path, if one exists, if the resolution of the grid is fine enough
     - For probabilistic planning, a *probabilistically complete* planner has a failure probability asymptotically approaching zero as more work is performed
  2. Optimality: The planner should report the best (lowest cost) path
     - A planner is *anytime optimal* if, as more work is performed, progressively better paths are found, i.e. we can cut off the planning early to find a feasible but less optimal path
  3. Efficiency: The planner should find the solution in a timely manner
- The planning task involves finding a sequence of actions that take us from the initial state x_I to some final state within the goal set X_G
  - We can use a generic forwards-search algorithm to find the path
    * At each step, take the next state x from the queue, mark it as visited, discover all unvisited nodes reachable from x, and add each reachable unvisited node to the queue
  - States that have been encountered but not visited are kept in a queue, and the prioritization of this queue leads to different search algorithms:
    * LIFO (stack): depth-first search
    * FIFO (queue): breadth-first search
    * Cost-to-come (i.e. cost to get to the node): Dijkstra's algorithm
    * Heuristic-guided cost function: A*
    * Estimated cost function: suboptimal best-first

Lecture 8, Jan 19, 2026

Optimal Graph-Based Planning

- We are now interested in finding the optimal path, given nonuniform edge costs
- *Bellman's principle of optimality*: An optimal policy has the property that regardless of the initial states and decisions, the remaining decisions must constitute an optimal policy with regard to the state resulting from the initial decisions, i.e. every point along the path is the start of another optimal path
  - This means that an optimization problem in discrete time can be stated in a recursive, step-by-step form where we start from the final state and work backwards
- Define a cost of a sequence π_K as $L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F)$ where $l(x_k, u_k)$ is the cost of taking action u_k in state x_k, and $l_F(x_F)$ is the cost of the final state
- Let $G^*(x)$ denote the cost-to-go (i.e. optimal cost of a path to the goal from x), then we can work backwards from the final state and get $G^*(x) = \min_u \{ l(x, u) + G^*(f(x, u)) \}$ (*Bellman's equation*) where $f(x, u)$ transitions state x by applying input u
  - Using this we can define a *value iteration* approach
- *Dijkstra's algorithm* prioritizes the queue based on cost-to-come, so that at each step we expand the node with the smallest path-to-come, so that at each step we expand the node with the smallest path-to-come, so that at each step we expand the node with the smallest path-to-come, so that at each step we expand the node with the smallest path-to-come
- A* prioritizes states based on the sum of the cost-to-come and a lower bound heuristic on the cost-to-go (e.g. Manhattan or Euclidean distance)
  - This prevents us from expanding states that would not get us closer to the goal, since these states will have a higher cost-to-come but not a lower heuristic estimate of the cost-to-go
  - Since the heuristic is a lower bound, the algorithm is still optimal
  - Note that when we first find a path, we're not done, but we can skip processing all remaining nodes in the queue with a priority equal or higher than that of the goal since those nodes cannot possibly lead to a better path
- Lifelong planning A* (LPA*) can reuse results from previous A* runs when only a few edge costs have changed
  - All cells that have an incoming edge cost change are checked for inconsistency, by recomputing the cost-to-come based on predecessor's values
  - This progresses outwards until all cells have consistent costs, and the search is restarted using this information
- Focused Dynamic A* (or D*) is an improved version of LPA* that replans while moving towards the goal, making them even more efficient than A^*

Lecture 9, Jan 21, 2026

Sampling-Based Planning – Probabilistic Roadmaps (PRMs)

- Consider the problem of planning a path in configuration space, where some states are free and some others are forbidden; we have the ability to sample some states to do a collision check, but it would be too expensive to check every single state
- We can randomly sample a large number of points and check each one for collisions, discarding the forbidden states
  - The remaining states are the known as the *milestones*
- For each milestone, try to link it to its nearest neighbours using a straight line path, and check each path for collisions, keeping the collision-free ones
  - Since the path is continuous, we will need some kind of approximation
  - The remaining edges are retained as the *local paths*
- *Multi-query roadmaps* build the PRM once, and reuse it for answering queries; *single-query roadmaps*

compute the PRM from scratch for each new query, which is useful for dynamic environments

- More precisely, given starting state s and goal state g, configuration space C and free space $F \subseteq C$:
  1. Initialize PRM R with nodes s, g
  2. Repeat until s and g are in the same connected component of R, or after a maximum of N iterations:
     1. Sample a configuration q from the configuration space C with probability p
     2. If $q \in F$ then add it as a new milestone of R
     3. For each milestone $v \in R$ such that $v \neq q$, if path $(q, v) \in F$, then add it as a new edge of R
        - Note here we can apply any technique we want to select v, and control the number of nearest neighbours that we consider
  3. If s and g are connected, return a path between them, otherwise return no path

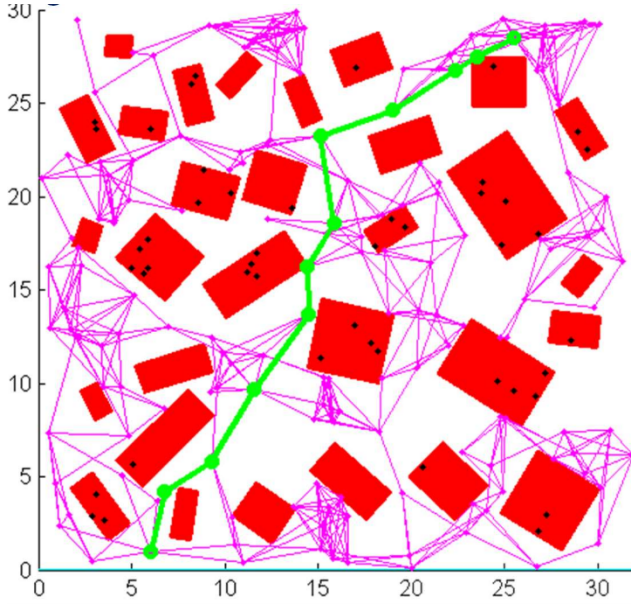

Figure 11: PRM planning example.

- Checking sampled configurations and connections between them for collisions is the most important part of the algorithm, which can be done with *hierarchical collision detection*
- Instead of sampling randomly, we can apply non-uniform sampling strategies to select more informative states, so we can capture the connectivity of free space with relatively few nodes and local paths
  - PRM is bottlenecked by the ability to find milestones in narrow passages and connect them to the rest of the graph
- The *visibility set* of a configuration q is $V(q) = \{ q' \mid (q, q') \in F \}$, i.e. the set of all configurations that can be connected to q by a straight line edge
  - Intuitively, the larger the visibility set of a configuration, the more of free space it captures
  - The size of $V(q)$ is the *expansiveness* of the configuration q
  - We can sum the expansiveness of all the configurations in C to get a sense of the expansiveness of the entire configuration space
    * High expansiveness environments are generally open, with wide passages
    * Spaces with narrow passages, especially non-straight passages, have low expansiveness
    * A convex set (i.e. a set in which we can connect every point with every other point) is maximally expansive
- In practice, most planning problems result in relatively expansive configuration spaces even with complex constraints, so PRM tends to work pretty well
- It's possible to prove that with probability converging exponentially in the number of milestones, a feasible path will be found, if one exists
  - This means PRM is probabilistically complete, and we can also find paths relatively quickly due

to the exponential convergence

- We can do this in a batched manner where we first find a set of milestones, then the edges, and then search the graph hoping for connectivity, or we can do it in an online manner, adding one milestone and paths each time and doing a search to check for connectivity
  - The online search can lead to faster convergence, but sacrifices optimality
  - We can mix these two approaches for a balance between anytime convergence and optimality

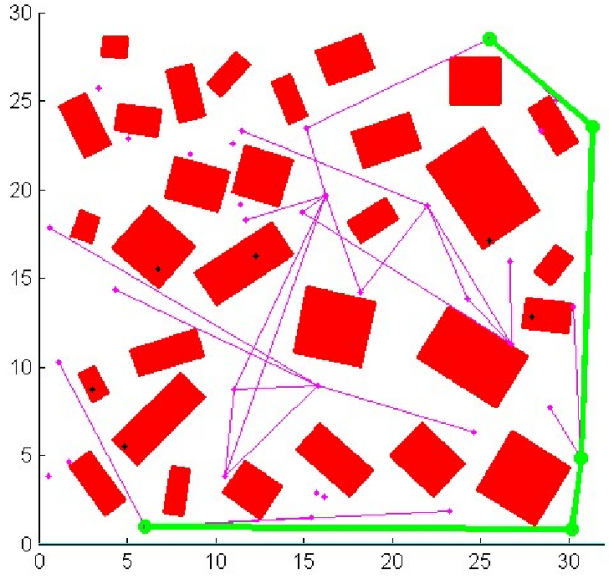

Figure 12: Result of online PRM planning for the same problem.

- In 2D spaces, we can obtain the true optimal path by using a *visibility graph*, where each vertex on each obstacle is a milestone and connected to every other possible vertex
  - However this only works in 2D and without any motion constraints, so it's not very useful in practice
  - This also takes a very long time to build the graph, so PRM may still be preferable since it can generate a non-optimal path in shorter time

Lecture 10, Jan 26, 2026

Efficient Sampling-Based Planning

- Goal: minimize the PRM size we need to find a feasible path to the end configuration
- Instead of uniform sampling we try to identify samples with good visibility; regions with poorer visibility (e.g. narrow passages) should be more densely sampled than open space
- 4 strategies exist for milestone sampling:
  - Workspace-guided strategies: find narrow passages in the workspace and map them into the configuration space to sample those areas more
    - This is simple but limited
  - Filtering strategies: sample many configurations to find interesting patterns, and only retain promising configurations, since checking configurations is cheap but connecting them is much more expensive
    - *Gaussian sampling*: sample a configuration uniformly at random, then another configuration with a Gaussian distribution centered at the first; if one sample is in free space and the other is in collision, retain the one in free space, otherwise skip
      - The idea is that we bias retained samples to those that are near the boundaries of free space and occupied space

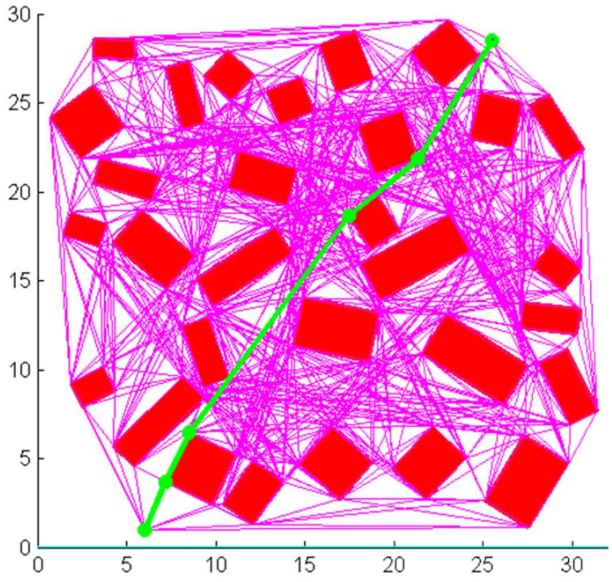

Figure 13: Planning on a visibility graph.

- Less likely to get samples in the middle of free space
- One variation is to sample uniformly within a ball centred at the first sample, with the size of the ball determined by sampling a 1D Gaussian, which may be faster

* *Bridge sampling*: sample 2 configurations using the Gaussian technique; if both are in collision, find the midpoint between the samples, and retain that sample if it's in free space
  - This biases the samples even further to narrow passages
* Gaussian sampling covers all surfaces while bridge sampling focuses on narrow passages, which may scale better in higher dimensions

– Adaptive strategies: adjusting the sampling distribution on the fly by considering collisions
– Deformation strategies: deforming the free space and morphing the resulting path

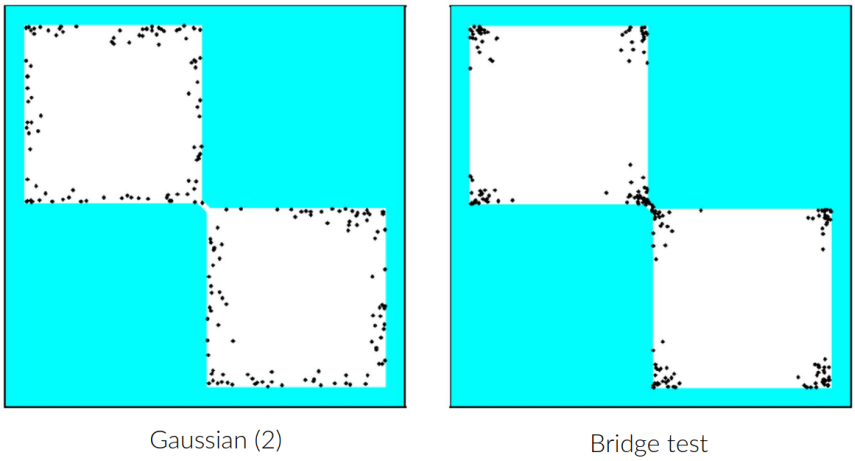

Figure 14: Comparison of Gaussian and bridge sampling.

- For connection sampling, nearest neighbour checking is usually the simplest and most successful
- Collision checking is usually the most computationally expensive; a few techniques can be used to optimize

- *Bounding volume hierarchy*: instead of checking for collisions of the entire complex shape, enclose objects into successively tighter bounding volumes and checking starting from the largest bounding volume
  * This can be done with e.g. circular or rectangular bounding volumes; lower level bounding volumes can be created by breaking up into subregions and creating a bounding volume for each one
    - Considerations include bounding volume accuracy and ease of computation
  * If we detect a collision with a bounding volume, we then check again with all the sub-volumes enclosed within it until we get to the lowest level with the desired accuracy
  * Most obstacles will be far away, so the first checks can eliminate lots of obstacles with very little work
  * This is particularly helpful in static environments since bounding volumes can be computed in advance
- To check a path for collisions, we can first check the distance between endpoints and obstacles to eliminate large parts of the path, then bisect any remaining path length successively

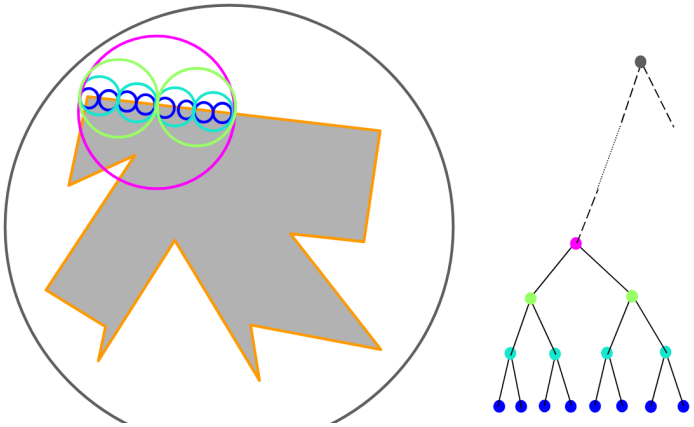

Figure 15: Bounding volume hierarchy illustration for a complex shape.

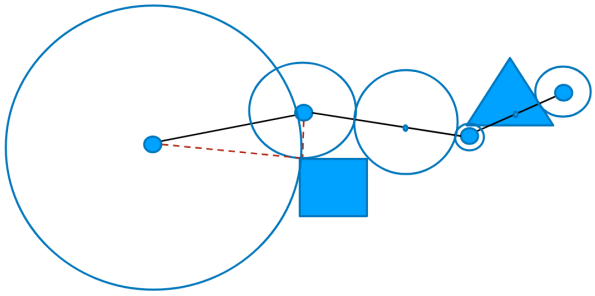

Figure 16: Adaptive collision checking.

- Most collision-free connections will not be a part of the final path, so we can do lazy collision checking to only check edges when they are being searched
  - Find a shortest path without checks, then collision check every edge on the path; if any have collisions redo the search without those edges

Lecture 11/12, Jan 28, 2026

Using Vehicle Models in Planning – Rapidly-exploring Random Trees (RRTs)

- Grid-based planning methods we previously discussed do not account for vehicle kinematics; this can work when the grid size is large so the robot can stay within the cells while respecting kinematic constraints, but when the grid is small it can fail
- To account for kinematics, we can define a state lattice where nodes are connected with motion primitives, so each node is an inherently feasible state
  - However, the grid cannot be arbitrarily fine, so the solution may not be resolution complete
  - This can be problematic especially in highly cluttered environments

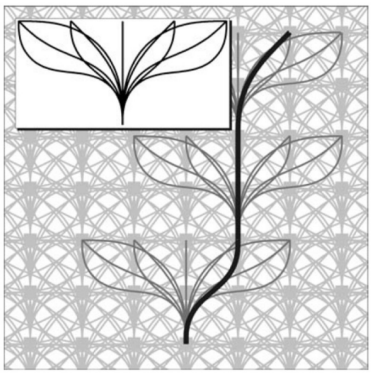

Figure 17: State lattice search-based planning.

- When vehicle constraints are involved, we often cannot execute an action backwards, so the structure we build will be a tree with directed edges instead of a graph as generated by PRM
- RRTs can be used to produce paths that are feasible by the vehicle model by building a tree, where every branch in the tree is a sequence of inputs we can apply, guaranteeing each state is feasible
- To construct the RRT:
  1. Randomly pick some point in space
     - We can work in physical or configuration space
  2. Identify the closest node in the tree to that point
  3. Select the vehicle input that would move us towards the point we sampled
     - Note that with constraints it's often impossible to determine the exact input that will take us to the sampled point, so we just go a small amount towards the sampled point
     - This is a critical difference from PRM
     - This can be done approximately, or through trajectory rollout, or through random sampling, or solving the boundary value problem
  4. Simulate the vehicle forward to determine the expected result, checking for obstacles, creating a new node if feasible
  5. Repeat until we reach the goal state
- RRT covers the space quickly as it explores, but it may take a long time to find a specific goal location, and the longer it takes to find the goal the more convoluted the final path will be
  - Therefore it's advisable to have a large goal region
- On each random sample, we can either get a refinement of the regions we've already explored (connecting to a vertex that already has connections), or an expansion into an unknown region (connecting to a new vertex)
  - The probability of extending each vertex is proportional to the area of its Voronoi region, with heavily favours expansion over refinement since the Voronoi cells of outer vertices are much larger
  - We can control the probability of refinement vs. expansion by restricting samples to within a bounding box of the root node
    * Start with a smaller box and gradually expand outwards; the larger the bounding box, the

more expansion dominates
* This favours early refinement to fully explore the space

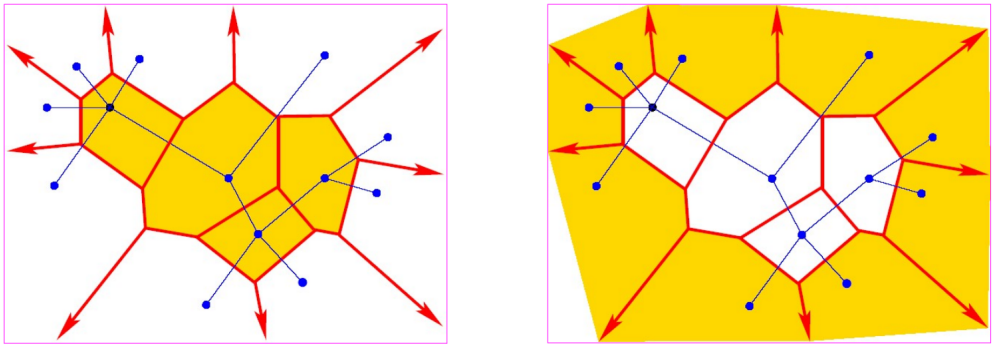

Figure 18: RRT expansion vs. refinement.

- Important drawbacks of RRT:
  - Solutions are sub-optimal and not smooth (further refinement often needed)
  - Difficulty handling small openings similar to PRM
  - Non-deterministic behaviour
- RRTs are probabilistically complete, but not optimal, because once an edge is added between nodes it never changes
- RRT* is an improvement which considers more than just the closest node when sampling and allows edges to be rewired to make path improvements, making it asymptotically optimal
  - After connecting a new node, RRT* further considers a ball in the neighbourhood of the new node; if we can connect to any nodes within the neighbourhood that results in a shorter total path, we connect to that node instead
  - After rewiring, we check all other nodes in the ball to see if the new node we added results in an improvement
  - For leaf nodes, we can allow heading variations for additional flexibility

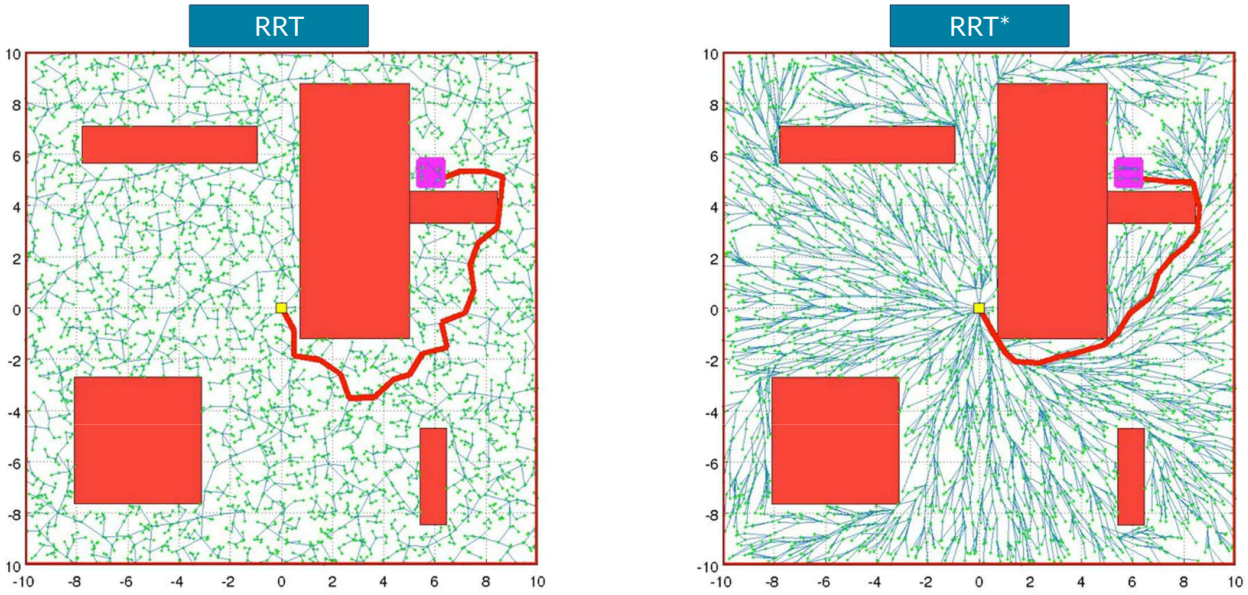

Figure 19: RRT vs. RRT* generated trees.

- Informed RRT* uses a heuristic to bound the search space instead of expanding randomly in all

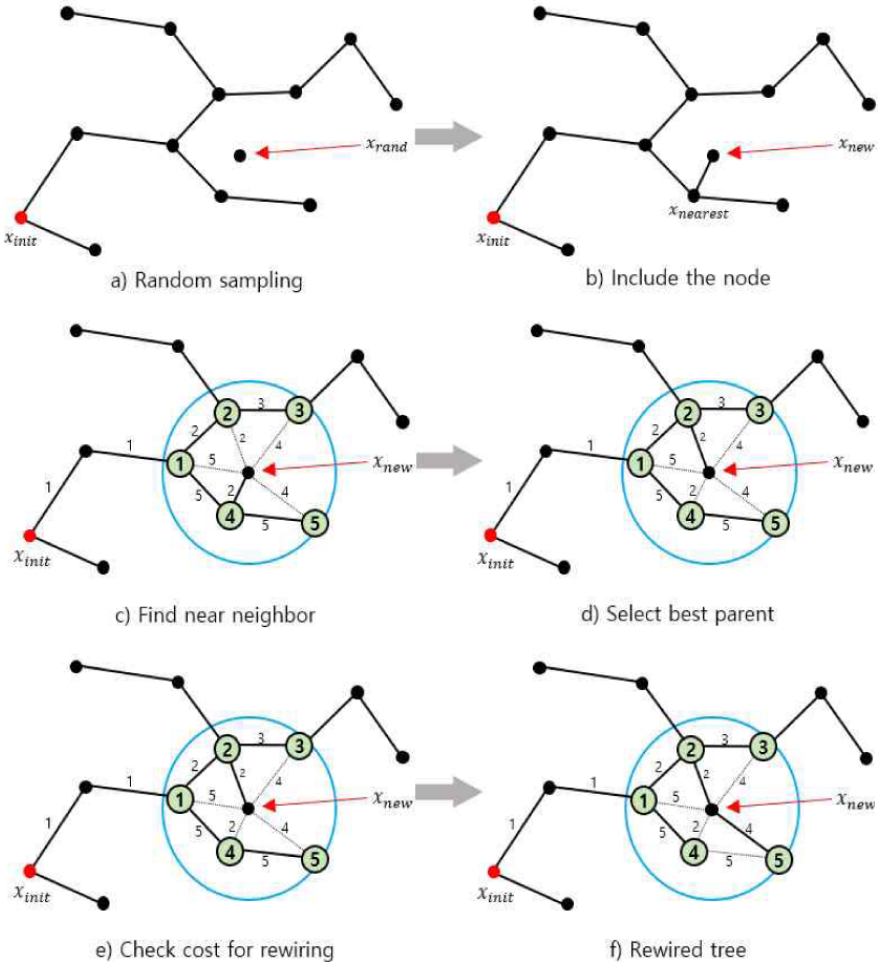

Figure 20: RRT* algorithm summary.

directions, similar to the idea of A*

- After finding an initial path, we can form an ellipse where possible better paths can be in, using the path length and starting and ending nodes as foci, and restricting the sampling to only within this space
- Batch Informed RRT* (BIT*) also biases the initial goal search to an ellipse

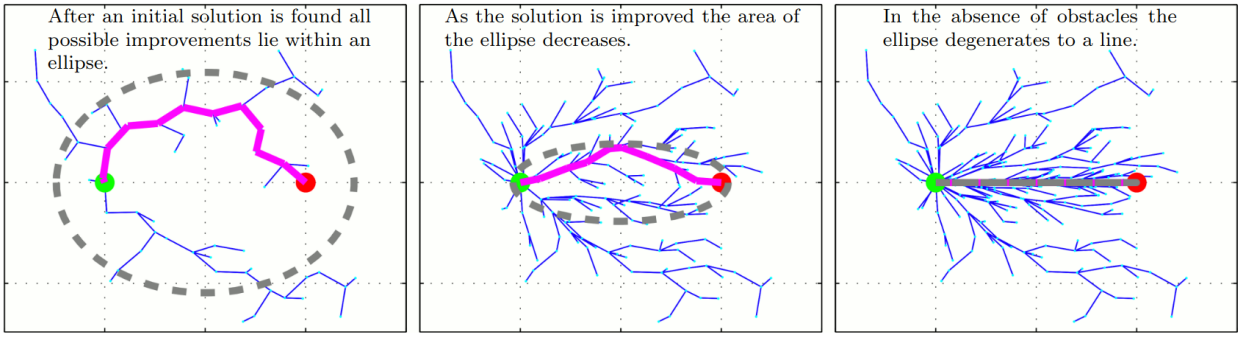

Figure 21: Informed RRT* ellipse bound.

	Graph Planners (configuration space no model)	Tree Planners (input space with model)
Complete planners (feasible; finds a solution)	• Breadth-First Search • Depth-First Search	• RRT • State Lattice
Optimal Planners (finds the best solution)	• Value Iteration • Dijkstra's / A* • Lifelong Planning A* • PRM	• RRT* • Informed RRT* • FMT* • BIT*

Figure 22: Summary of planning methods.