Introduction to Robotics

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Overview:

1. Introduction
2. Definitions & Terminology
3. Rigid body motions and kinematics (inverse and forward kinematics)
4. Trajectory generation
5. Motion planning

Lecturer: Prof. Dr. Horsch

Email: thomas.horsch@h-da.de

Sources:

- Kevin M. Lynch, Frank C. Park: Modern Robotics – Mechanics, Planning and Control (http://modernrobotics.org)
- Wikipedia

Sources

Sources:

- Lynch, Park: Modern Robotics (available as PDF)
- Software: CoppeliaSim from Coppelia Robotics (EDU version)
  - Former name: Virtual Robot Experimental Platform (V-REP)

Introduction - Applications

Robotics is an **interdisciplinary branch** of engineering and science that includes

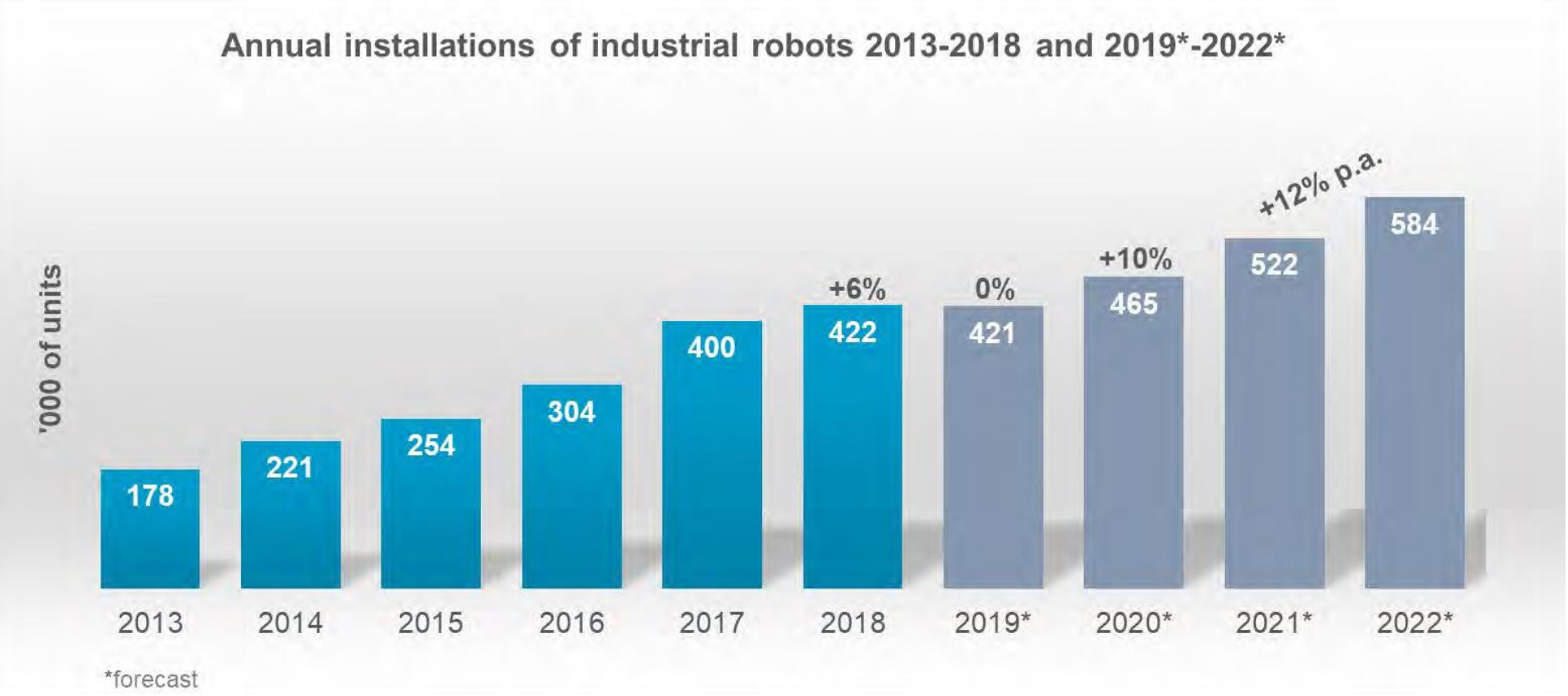
- mechanical engineering
- electrical engineering
- **computer science** and others...

Robotics deals with the design, construction, operation, and use of robots, as well as computer systems for their control, sensory feedback, and information processing.

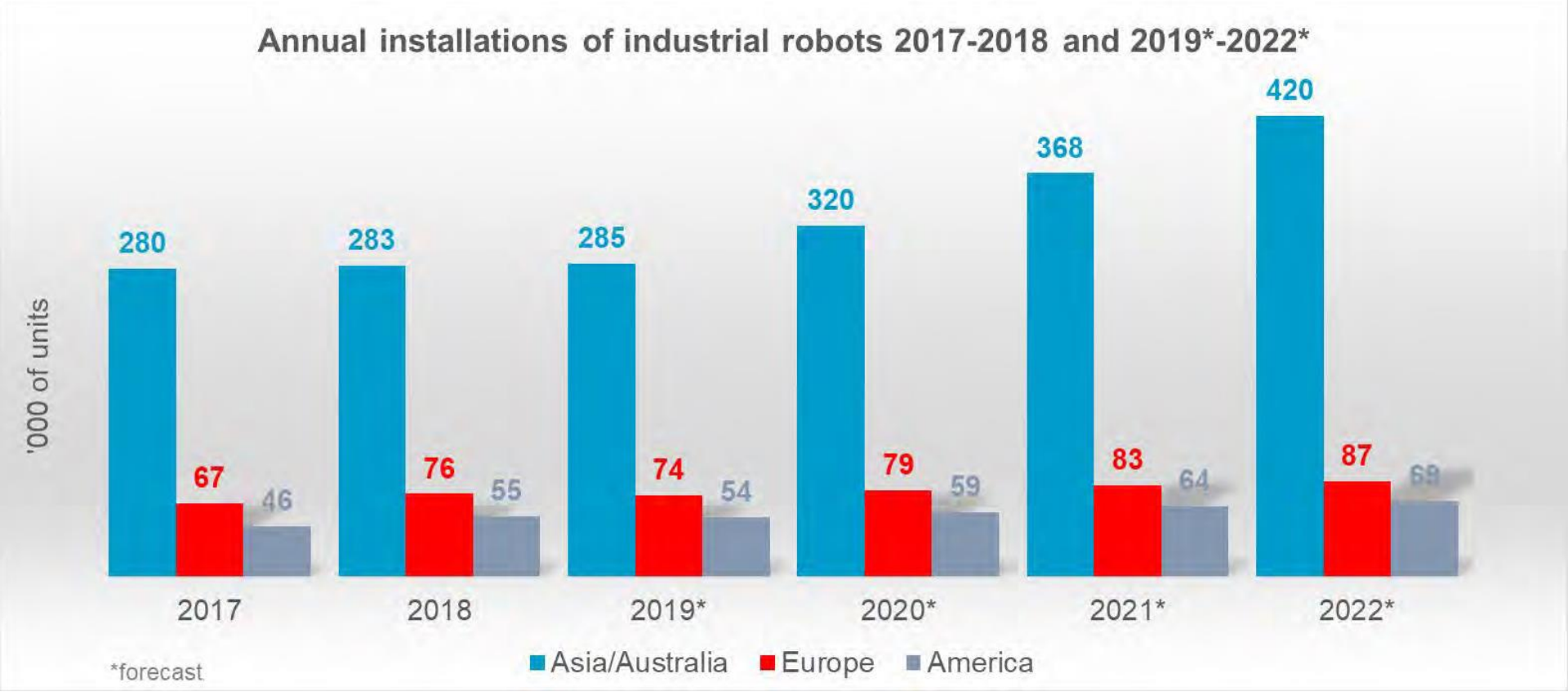
Typical **industrial robot** applications:

- spot welding
- arc welding
- machine handling
- pick & place
- coating and clueing
- deburring
- assembly / disassembly

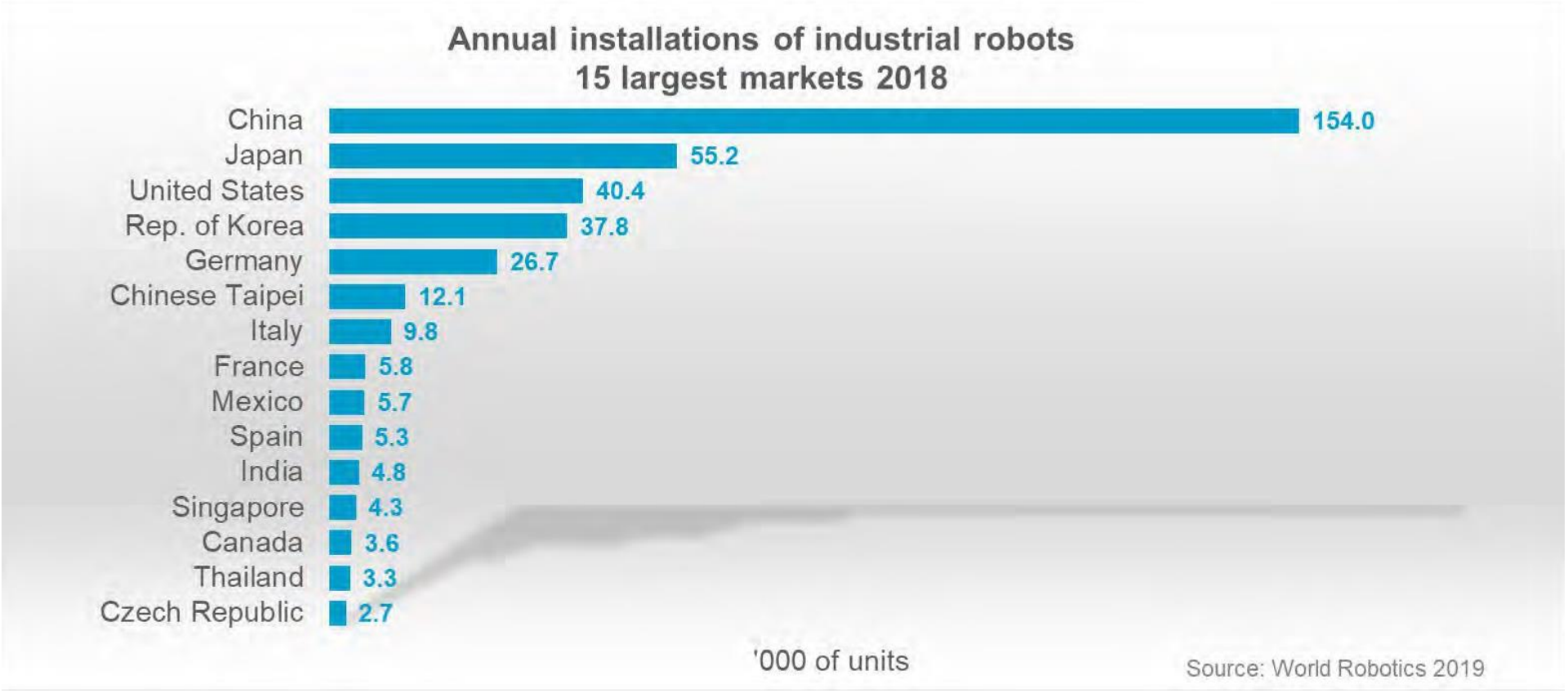
Introduction - market



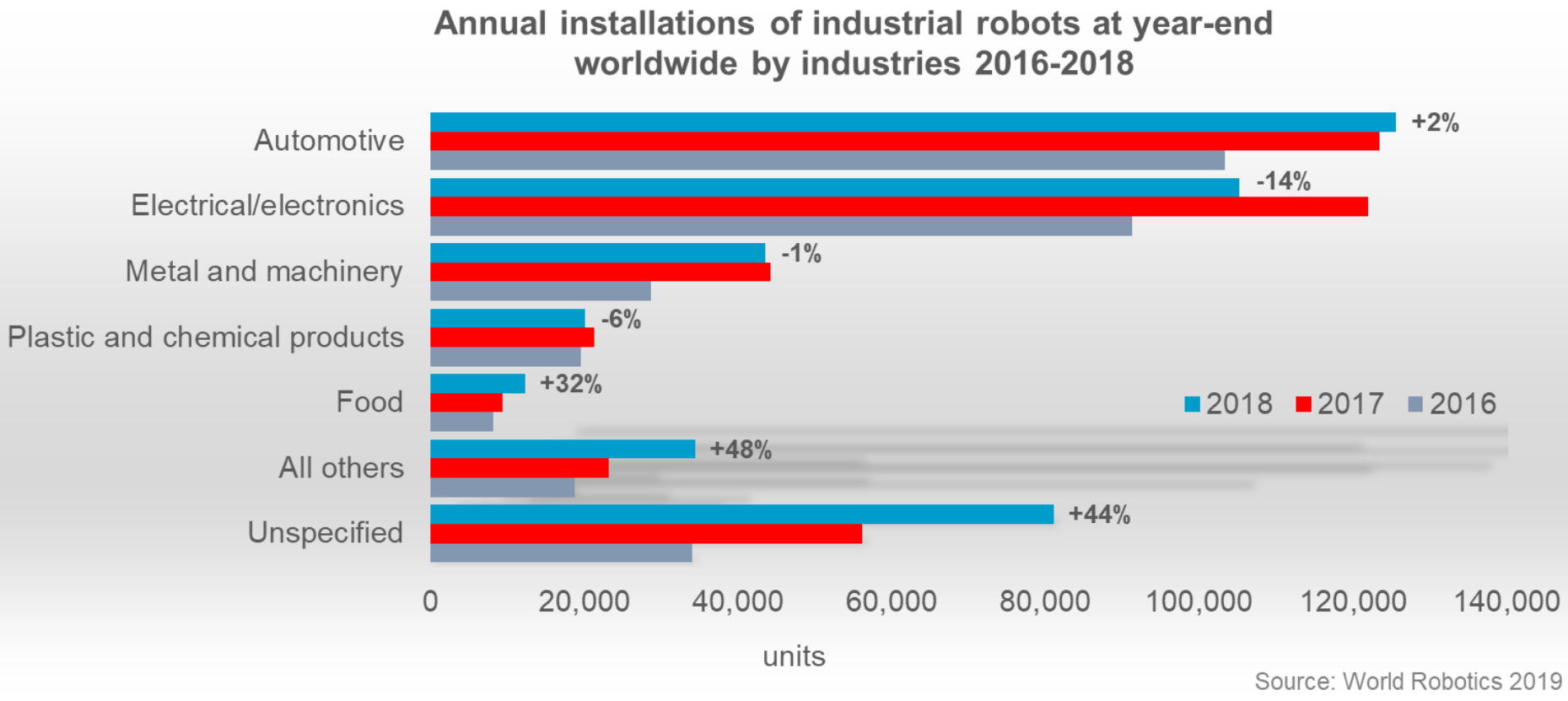
Introduction - market



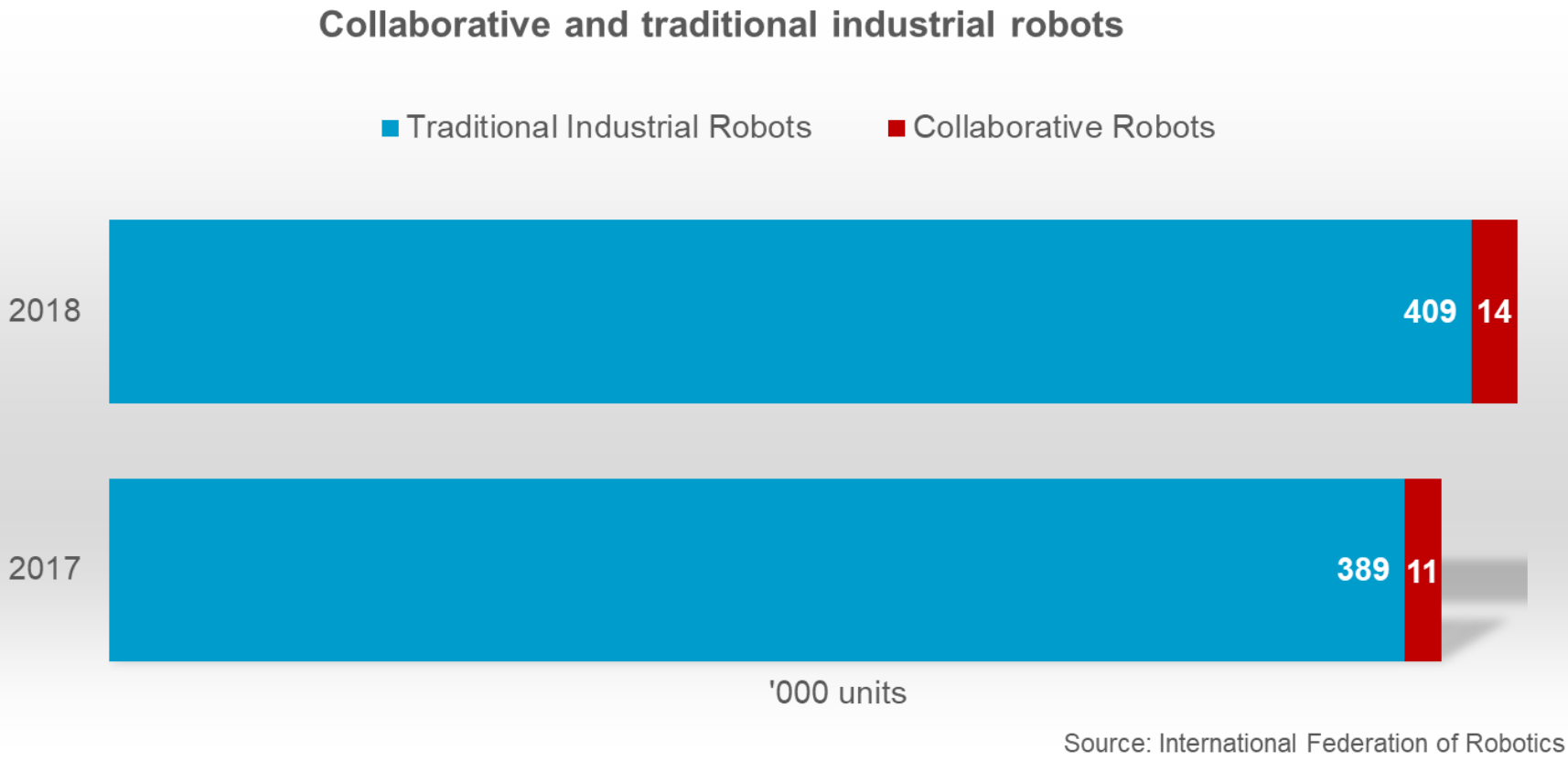
Introduction - market



Introduction - market



Introduction - market



Introduction – Robot simulation

Example **EASYROB**:

- arc welding (proj/arc_abb.cel)
- redundant system (proj/bremen02.cel)
- milling (proj_marin/marin03)

Example **CoppeliaSim**:

- open different example files
- ….
- ……

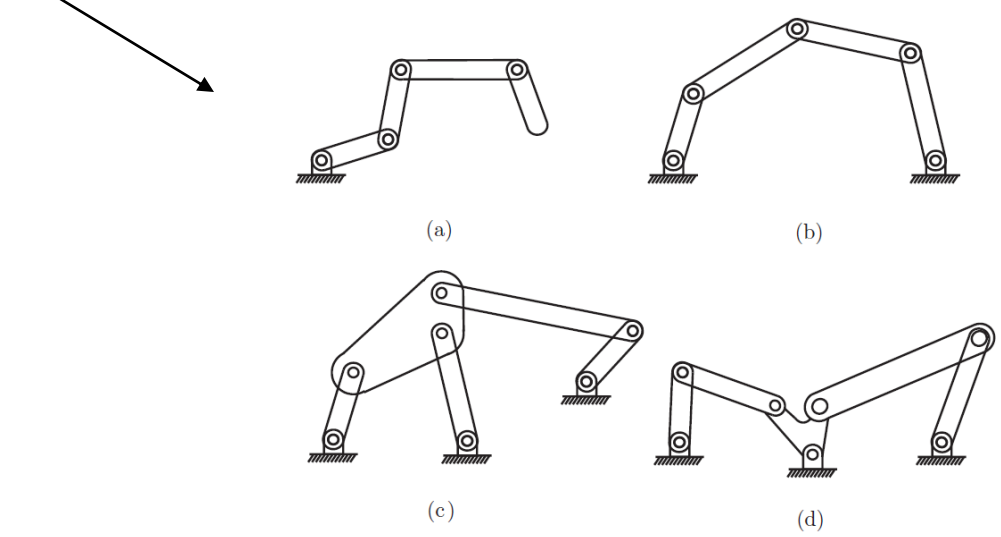
Overview:

1. Introduction
2. **Definitions & Terminology**
3. Rigid body motions and kinematics (inverse and forward kinematics)
4. Trajectory generation
5. Motion planning

Introduction – Definitions &Terminology

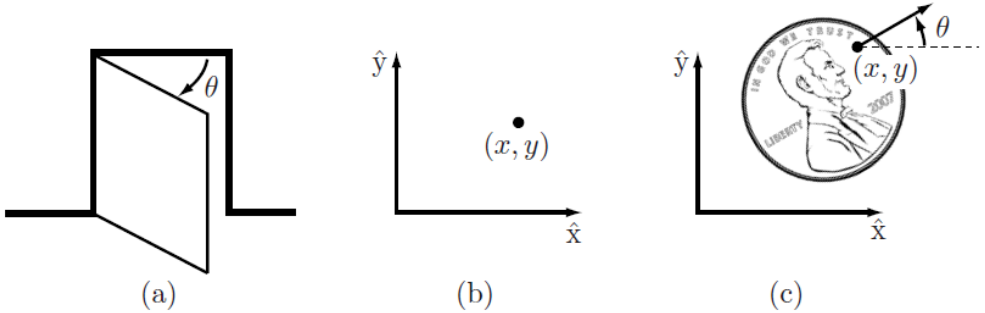
- Multi-body systems
  - Multibody system is the study of the dynamic behavior of interconnected rigid (or flexible) bodies, each of which may undergo translational and rotational displacements.
  - Basically, the motion of bodies is described by their kinematic behavior.

- Robot is a multi-body system, most of them can be characterized by a **open serial kinematic chain**, since the links are connected to each other one after the other



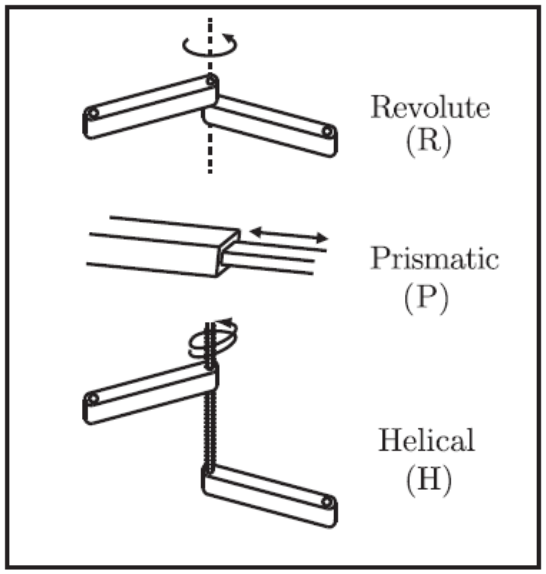
Introduction – Definitions &Terminology

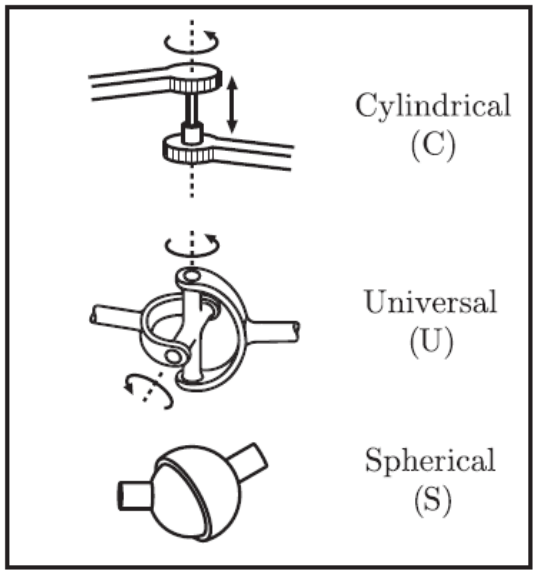
- How many dof ?



Robots links are connected by joints

- Typical robot joints

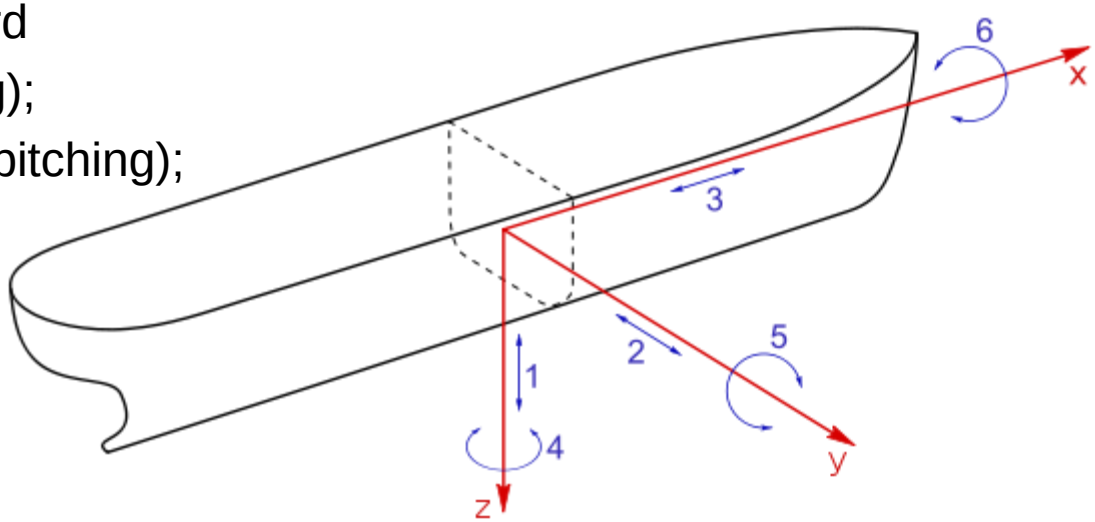




Introduction – Definitions &Terminology

- **Degrees of freedom (DOF)**
- The number DOF of a robot is the smallest number of coordinates needed to represent its **configuration**
- The configuration of a robot is a complete specification of the position of every point of the robot.

- The motion of a ship at sea has the six degrees of freedom of a rigid body, and is described as:
- Moving up and down
- Moving left and right
- Moving forward and backward
- Swivels left and right (yawing);
- Tilts forward and backward (pitching);
- Pivots side to side (rolling).



Introduction – Definitions &Terminology

- **Configuration space (C-space)**
- The set of all configurations a robot can achieve
- Shape of configuration space is important (topology)

system	topology	sample representation
point on a plane	 $\mathbb{E}^2$	$\mathbb{R}^2$
spherical pendulum	 S^2	$[-180°, 180°) \times [-90°, 90°]$
2R robot arm	 $T^2 = S^1 \times S^1$	$[0, 2\pi) \times [0, 2\pi)$
rotating sliding knob	 $\mathbb{E}^1 \times S^1$	$\mathbb{R}^1 \times [0, 2\pi)$

Introduction – Definitions &Terminology

- **Configuration space (C-space)**

Type of robot	Representation of Q
Mobile robot translating in the plane	$\mathbb{R}^2$
Mobile robot translating and rotating in the plane	$SE(2)$ or $\mathbb{R}^2 \times S^1$
Rigid body translating in the three-space	$\mathbb{R}^3$
A spacecraft	$SE(3)$ or $\mathbb{R}^3 \times SO(3)$
An n-joint revolute arm	T^n
A planar mobile robot with an attached n-joint arm	$SE(2) \times T^n$

Introduction – Definitions &Terminology

TCP, task space and work space

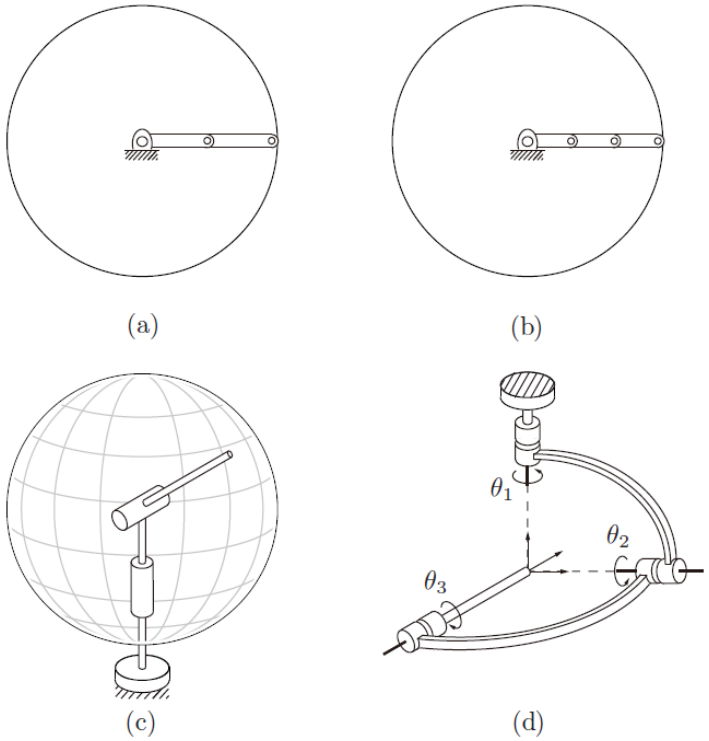
- The end-effector of a robot (e.g. a gripper) is used to interact with objects. Its coordinate system (frame) is called tool centre point (**TCP**)

- The **task space** is a space in which the robot's task can be naturally expressed.
  - If robot's task is to plot with a pen on a piece of paper, the task space would be R^2.
- The task space depends on the task; if the job is to point a laser pointer, then rotations about the axis of the laser beam are immaterial and the task space would be S^2, the set of directions in which the laser can point.

- The **workspace** is a specification of the configurations that the end-effector of the robot can reach. The definition of the workspace is primarily driven by the robot's structure, independently of the task.

Introduction – Definitions &Terminology

- Two mechanisms with different C-spaces may have the same workspace - figure (a) and (b).
- Two mechanisms with the same C-space may also have different workspaces - figure (a) and (c).

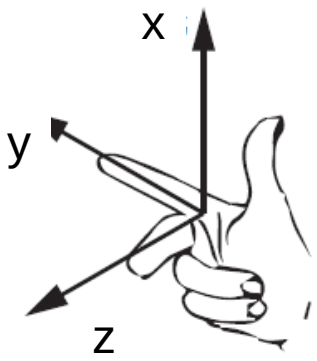


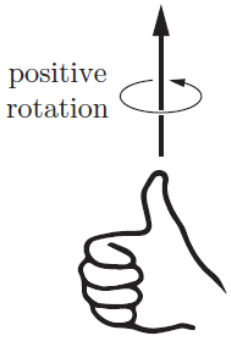
Overview:

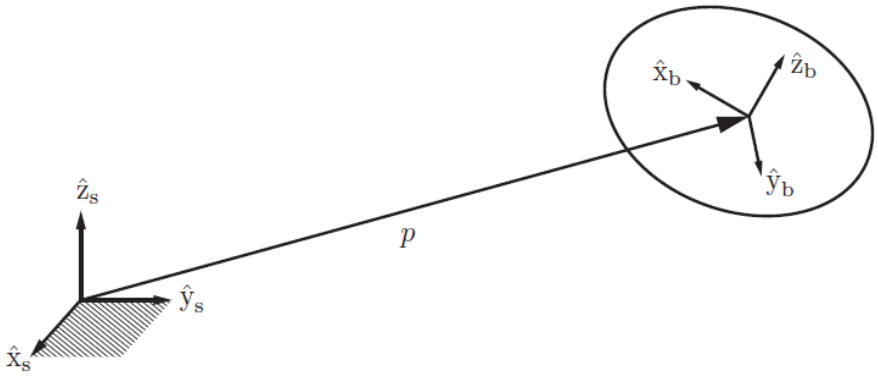
1. Introduction
2. Definitions & Terminology
3. **Rigid body motions and kinematics** (inverse and forward kinematics)
4. Trajectory generation
5. Motion planning

Rigid-body motions

Rotations







Rigid-body motions

Rotations representations:

- **three-parameter Euler angles:**
  - Z-Y-X rotations
  - Z-Y-Z rotations
  - X-Y-Z rotations
  - roll, pitch, yaw angles
- Angle-axis representation
- unit quaternions (four variables subject to one constraint)
- **3 by 3 rotation matrix**

Foundations – Rotations as 3x3 a matrix

$$R_X(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$R_Y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$$

$$R_Z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Multiplication is not commutative !

Question: What is the inverse of a rotation matrix ?

Kinematics - Foundations

Foundations – Rotations as 3x3 a matrix

Euler angle: Rot_x * Rot_y * Rot_z

$$\mathrm{Rot}_X(\alpha)\mathrm{Rot}_Y(\beta)\mathrm{Rot}_Z(\gamma) = \begin{pmatrix} C_\gamma \cdot C_\beta & -C_\beta \cdot S_\gamma & S_\beta \\ S_\alpha \cdot S_\beta \cdot C_\gamma + C_\alpha \cdot S_\gamma & -S_\alpha \cdot S_\beta \cdot S_\gamma + C_\alpha \cdot C_\gamma & -S_\alpha \cdot C_\beta \\ -C_\alpha \cdot S_\beta \cdot C_\gamma + S_\alpha \cdot S_\gamma & C_\alpha \cdot S_\beta \cdot S_\gamma + S_\alpha \cdot C_\gamma & C_\alpha \cdot C_\beta \end{pmatrix}$$

Example: $Rot_x(90)$*$Rot_y(0)$* $Rot_z(90)$

$$\mathrm{Rot}_X(90)\mathrm{Rot}_Y(0)\mathrm{Rot}_Z(90) = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix}$$

Kinematics - Foundations

homogeneous matrix combines rotation and translation p:

$$Tr = \begin{pmatrix} \cos(a) & -\sin(a) & 0 & p_x \\ \sin(a) & \cos(a) & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

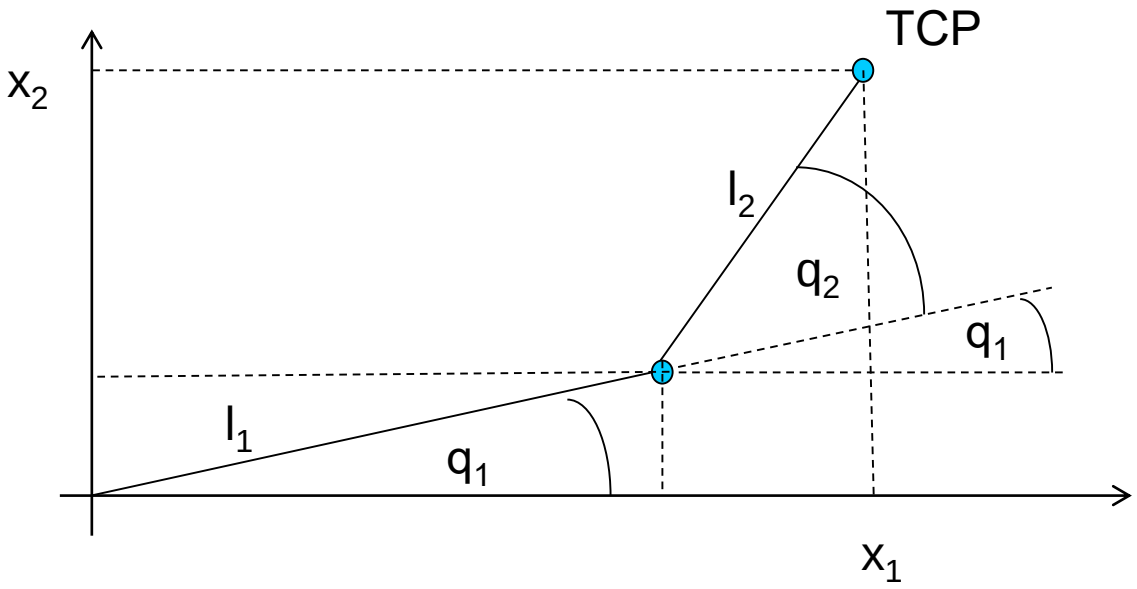
$$Transform1 = \begin{pmatrix} R & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} I & p \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R & Rp \\ 0 & 1 \end{pmatrix}$$

$$Transform2 = \begin{pmatrix} I & p \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} R & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R & p \\ 0 & 1 \end{pmatrix}$$

$$Tr = \begin{pmatrix} R & p \\ 0 & 1 \end{pmatrix} \Rightarrow inv(Tr) = \begin{pmatrix} R^T & -R^T p \\ 0 & 1 \end{pmatrix}$$

Kinematics - Foundations

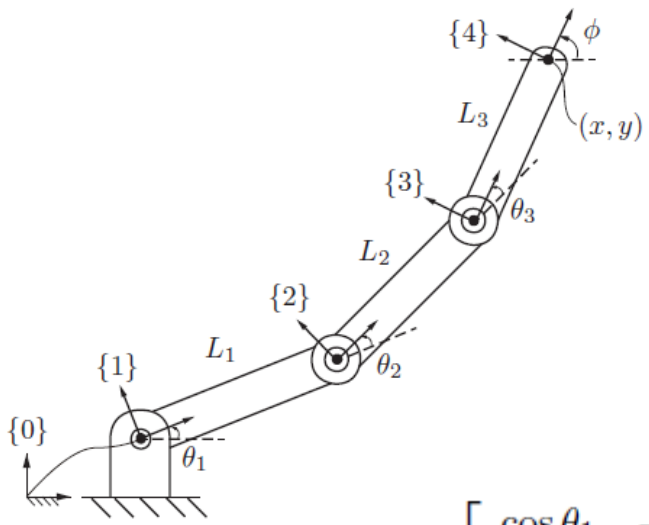
Example: forward kinematics of a 2 dof robot with rotational axes



$$f(q) = f(q_1, q_2) = \begin{pmatrix} f_1(q_1, q_2) \\ f_2(q_1, q_2) \end{pmatrix} = x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} l_1 \cos(q_1) + l_2 \cos(q_1 + q_2) \\ l_1 \sin(q_1) + l_2 \sin(q_1 + q_2) \end{pmatrix}$$

Kinematics - Foundations

Example: forward kinematics of a 3 dof robot with rotational axes with homogenous transforms



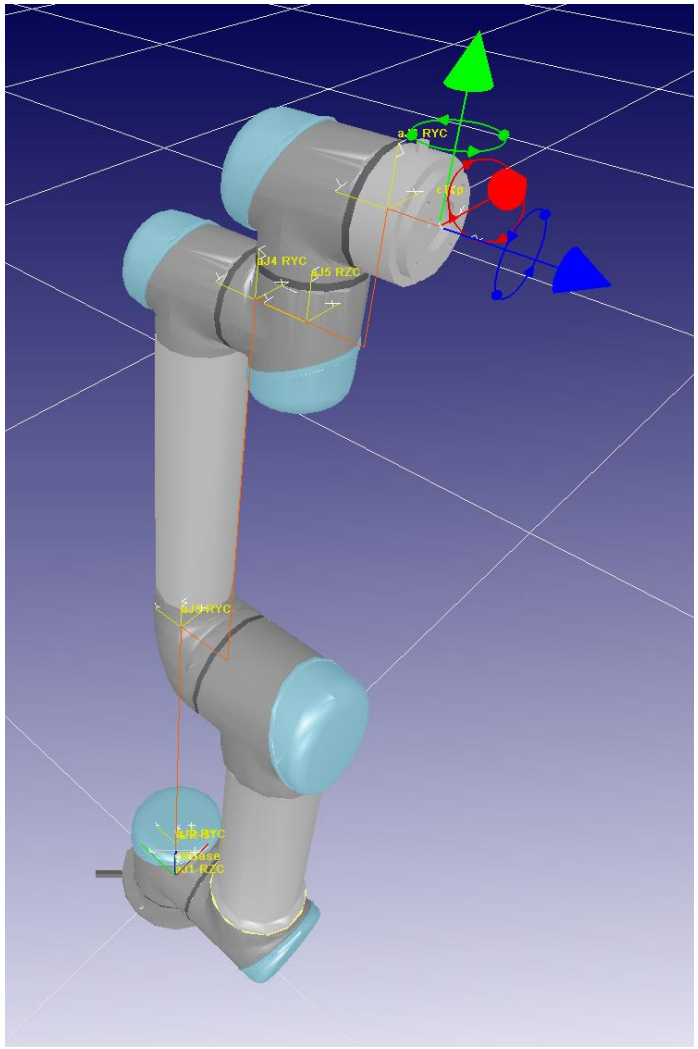
$$T_{04} = T_{01} * T_{12} * T_{23} * T_{34}$$

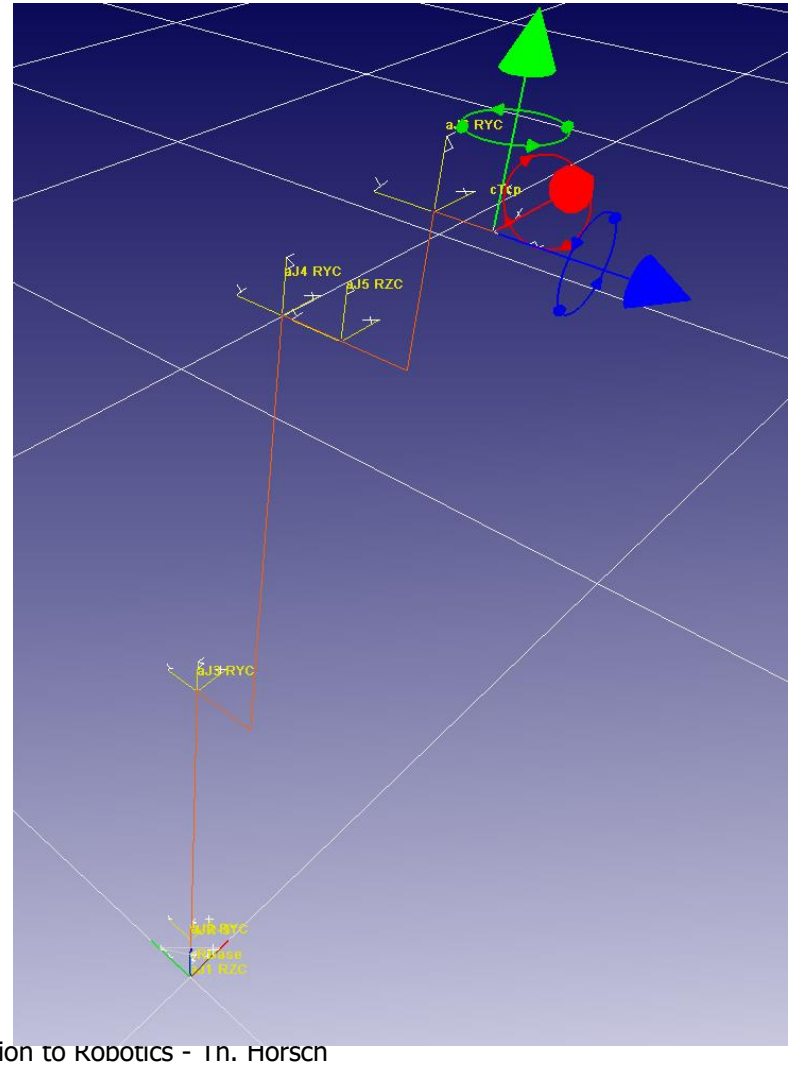
$$T_{01} = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T_{12} = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & L_1 \\ \sin\theta_2 & \cos\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{23} = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & L_2 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T_{34} = \begin{bmatrix} 1 & 0 & 0 & L_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Forward kinematics

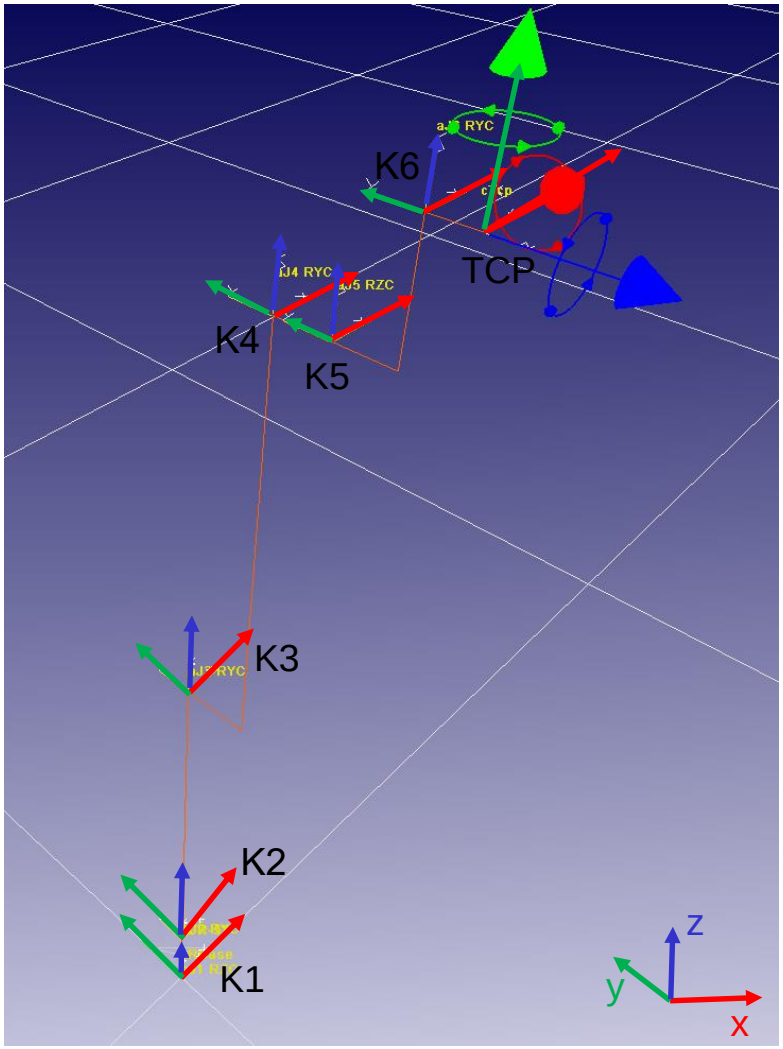
Example: 6 axes robot Universal Robot UR5 (axis 4-6 do not intersect in one point, so no decoupling of position and orientation possible





Forward kinematics

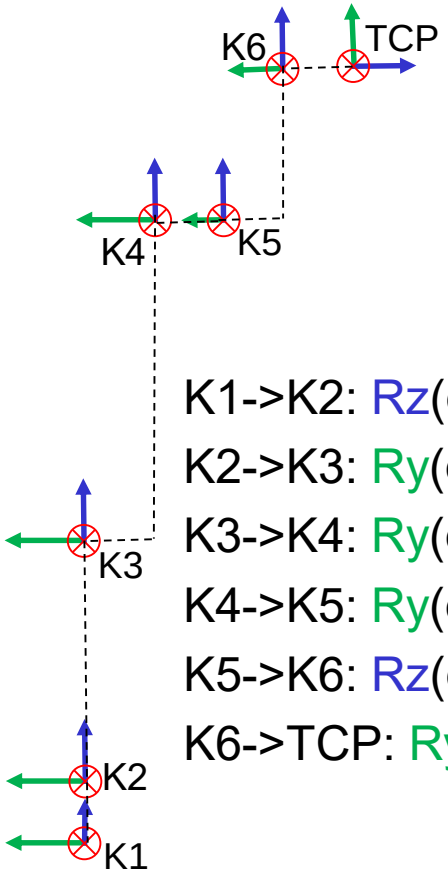
Sketch of UR5 model (perspective)



Sketch of UR5 model (in y-z plane)

⊗ axis points into the drawing plane (arrow from back)

⊙ axis points out of the drawing plane (arrow from front)



K1->K2: Rz(q1)*Trans(0, 0, 86.2)

K2->K3: Ry(q2)*Trans(0, 0, 425.)

K3->K4: Ry(q3)*Trans(0, -63.5, 392.4)

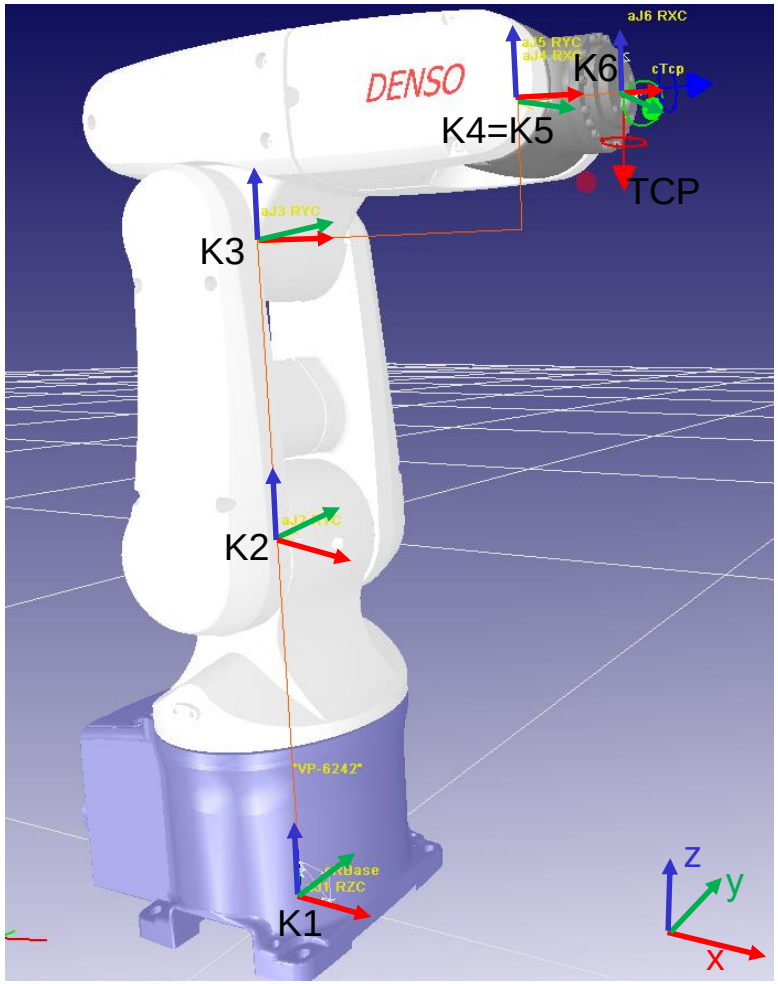
K4->K5: Ry(q4)* Trans(0, -45., 0)

K5->K6: Rz(q6)* Trans(0, -47.5, 93)

K6->TCP: Ry(q6)*T(0,-34.5,0)*Rx(90)

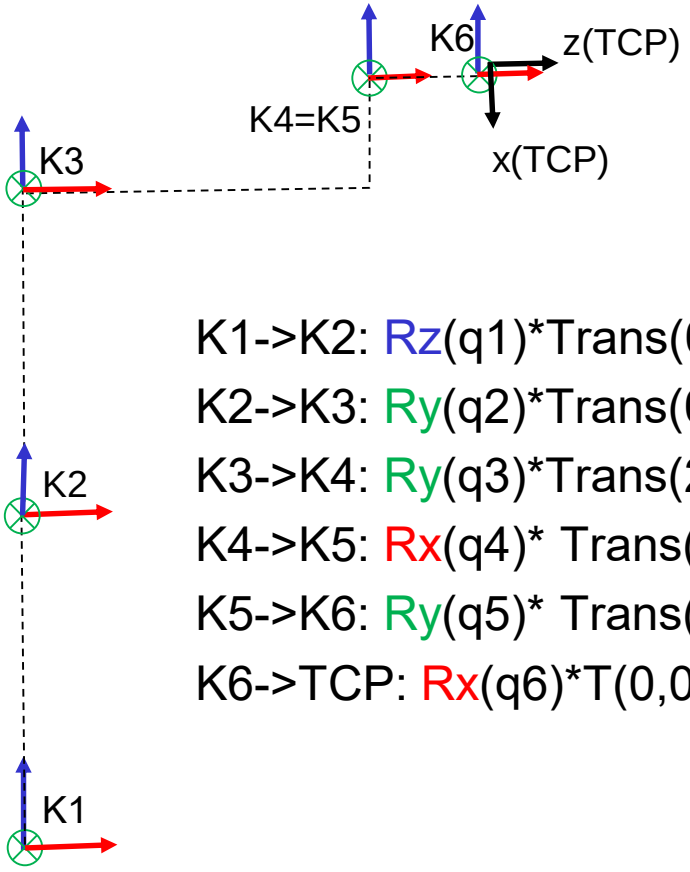
Forward kinematics

Example: 6 DOF Denso robot



⊗ axis points „into“ drawing plane (arrow from behind)

⊙ axis points „out of“ drawing plane (arrow from top)



K1->K2: Rz(q1)*Trans(0, 0, 280)

K2->K3: Ry(q2)*Trans(0, 0, 210)

K3->K4: Ry(q3)*Trans(210, 0, 75)

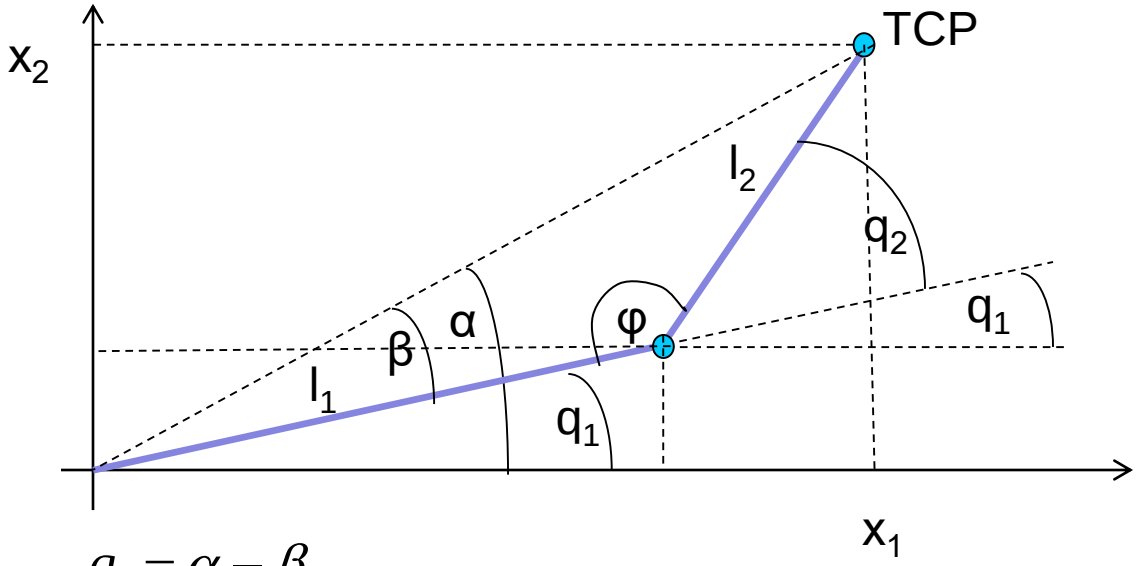
K4->K5: Rx(q4)* Trans(0, 0, 0)

K5->K6: Ry(q5)* Trans(70, 0, 0)

K6->TCP: Rx(q6)*T(0,0,0)*Ry(90)

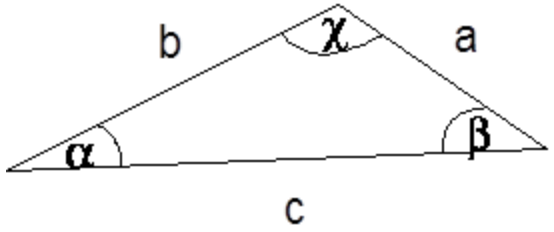
Kinematics - Foundations

Example: inverse kinematics of a 2 dof robot with rotational axes



law of cosine:

$$a^2 = b^2 + c^2 - 2 \cdot b \cdot c \cdot \cos\alpha$$



$$q_1 = \alpha - \beta$$

$$\alpha = \arctan\left(\frac{x_2}{x_1}\right), \beta = \arccos\left(\frac{l_1^2 + x_1^2 + x_2^2 - l_2^2}{2l_1\sqrt{x_1^2 + x_2^2}}\right)$$

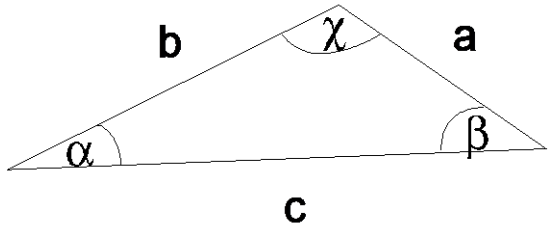
$$q_2 = 180 - \varphi$$

$$\varphi = \arccos\left(\frac{l_1^2 + l_2^2 - (x_1^2 + x_2^2)}{2l_1 l_2}\right)$$

Inverse kinematics

Law of cosine:

$$a^2 = b^2 + c^2 - 2 \cdot b \cdot c \cdot \cos\alpha$$



Inverse kinematics of a 6-dof robot with rotational joints, where the last 3 axes intersect in one point, called the hand wrist point (HWP)

Enables decoupling of position and orientation of end-effector (TCP)

Enables geometric solution approach

1. step: calculation of the hand wrist point (HWP) based on the given TCP frame
2. step: calculation of axes q1, q2, q3
3. step: calculation of axes q4, q5, q6

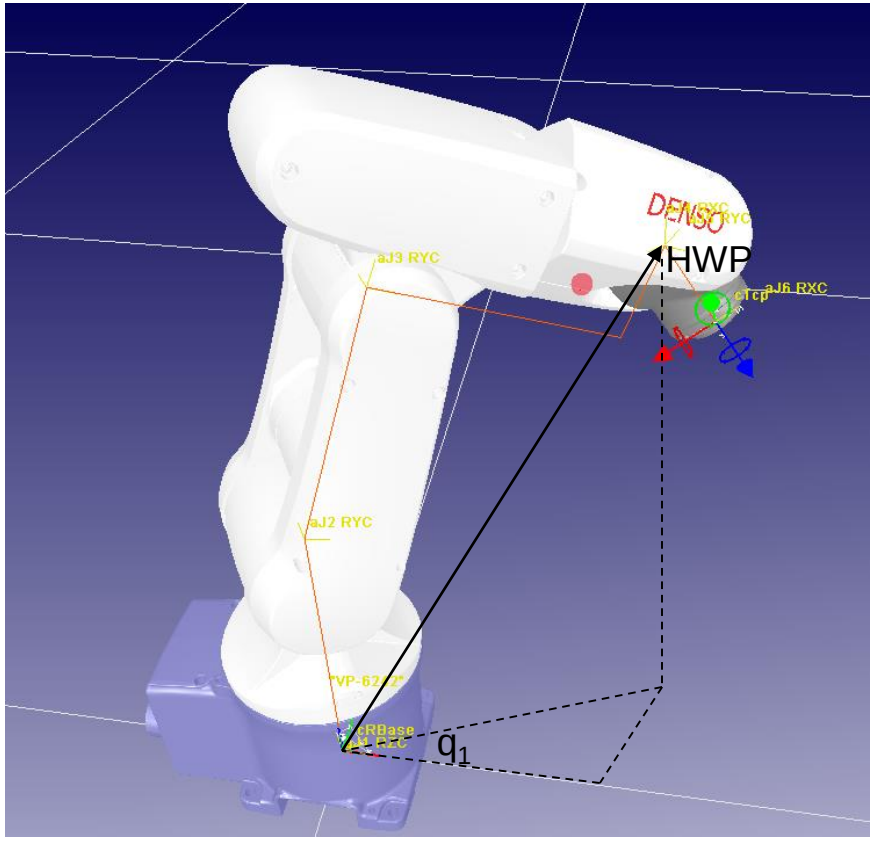
given: 4x4 transformation matrix T of the TCP

$$T(q) = {}^{6}_{1}A(q) = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Inverse Kinematics of a KUKA KR60 robot as a closed form solution: https://www.youtube.com/watch?v=3s2x4QsD3uM

Inverse Kinematics

Axis q1: is only dependant on the projection of the HWP into the x-y plane



Vector K1->K4 = p_{14} = HWP

Expressed in coordinate system 1

$${}^{1}\underline{p}_{14}$$

to coordinate system 4

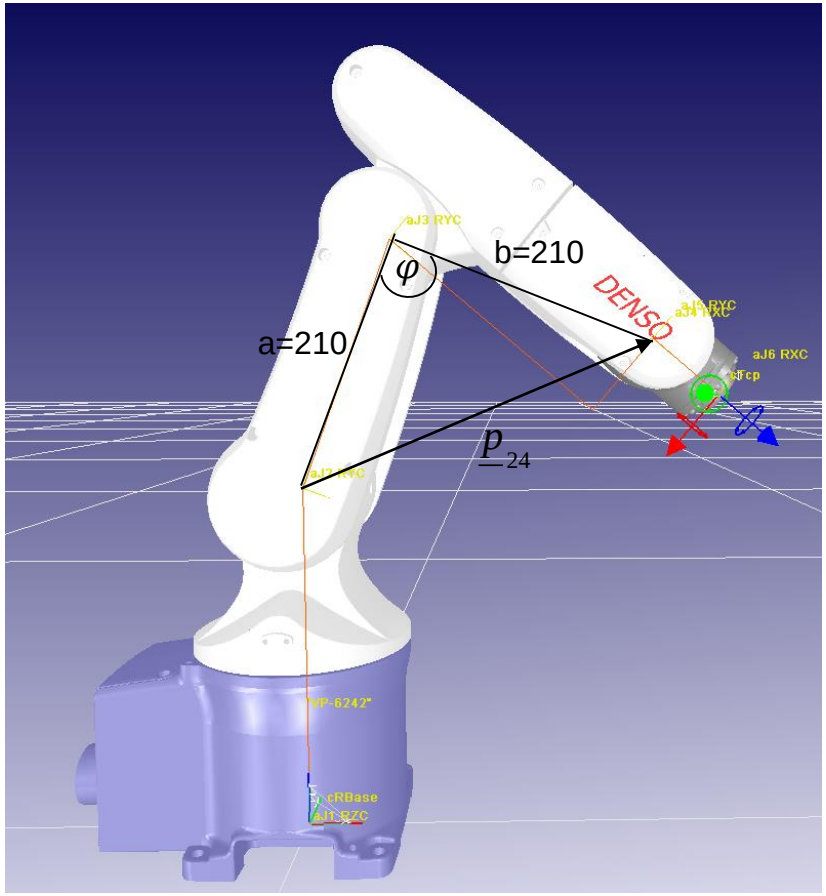
from coordinate system 1

$${}^{1}\underline{p}_{14} = {}^{1}\underline{p}_{16} - {}^{1}\underline{p}_{46} = {}^{1}\underline{p}_{16} - {}^{1}\underline{a}_{16} \cdot l_6$$

$$q_1 = \arctan 2(p_{14,y}, p_{14,x})$$

Inverse Kinematik

Axis q3



$$ {}^1\underline{p}_{24} = {}^1\underline{p}_{14} - {}^1\underline{p}_{12} = {}^1\underline{p}_{14} - \begin{pmatrix} 0 \\ 0 \\ 280 \end{pmatrix} $$

law of cosine wrt. ϕ:

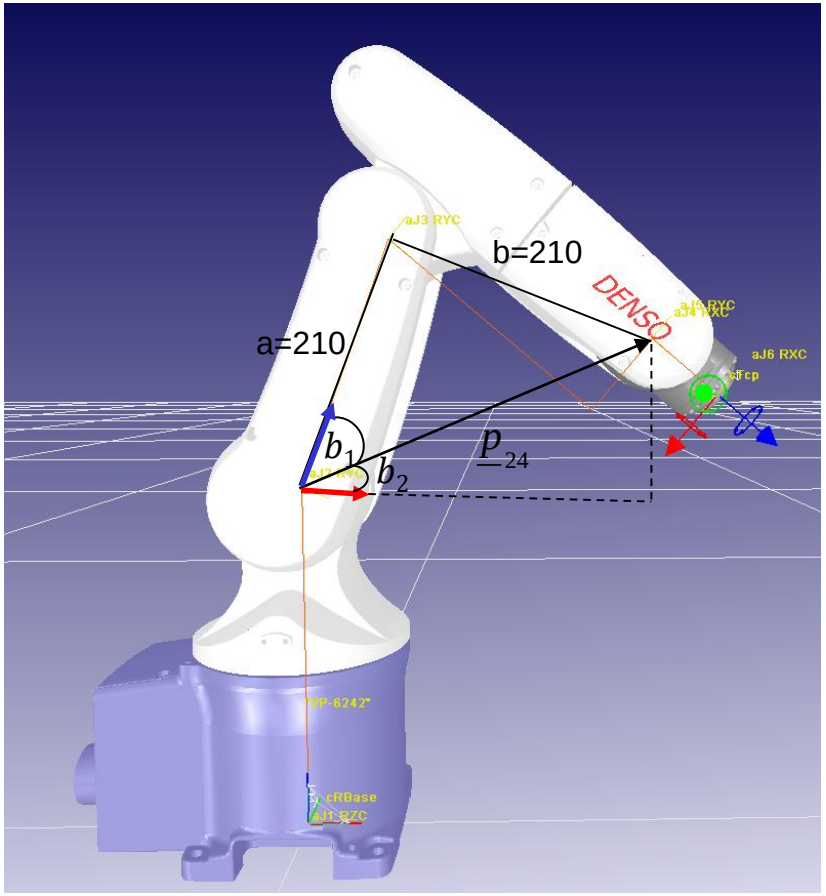
$$ |p_{24}|^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\phi $$

$$ \phi = \arccos \frac{a^2 + b^2 - |p_{24}|^2}{2 \cdot a \cdot b} $$

$$ q_3 = \frac{\pi}{2} - (\phi - \arctan(75/210)) $$

Inverse kinematics

Axis q2



$$ {}^2p_{24} = {}^1_2A\,{}^1p_{24} = Rot_Z(-q_1).{}^1p_{24} = \begin{pmatrix} \cos q_1 & \sin q_1 & 0 \\ -\sin q_1 & \cos q_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} {}^1p_{24} $$

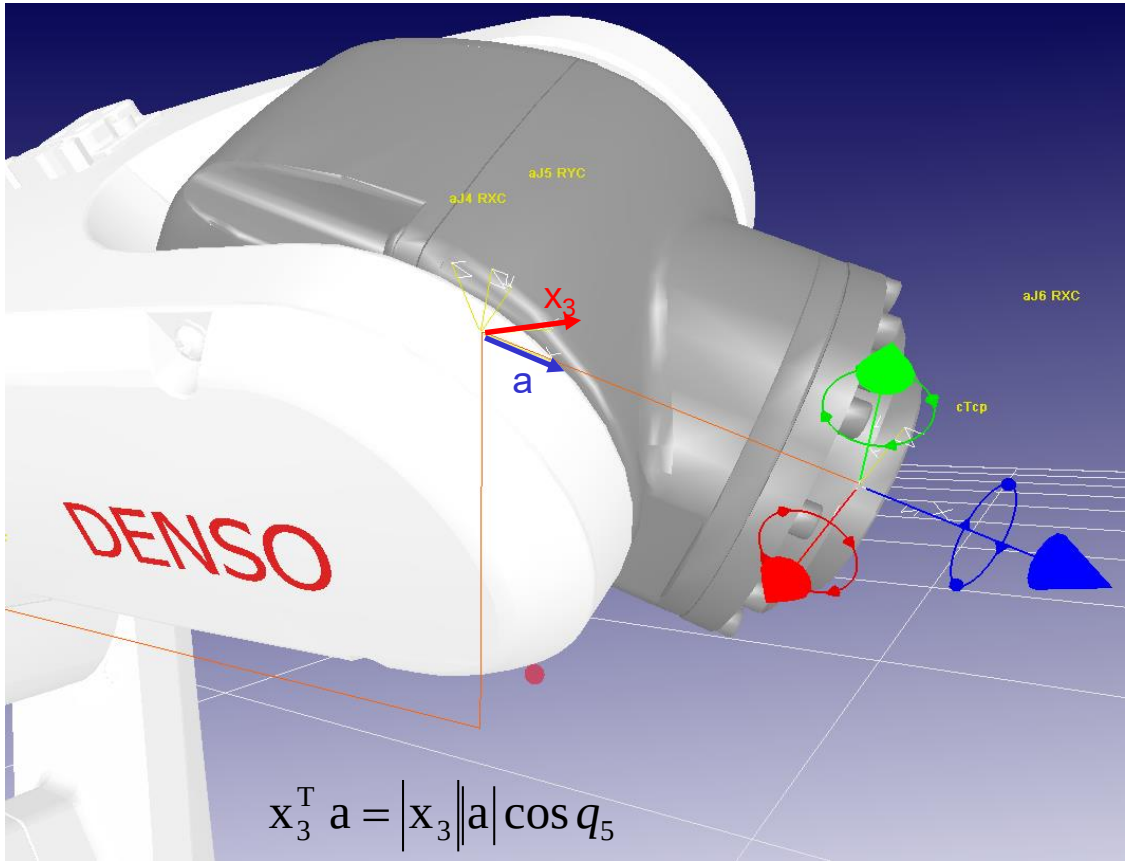
$$ b_1 = \arccos \frac{a^2 + \left|{}^2p_{24}\right|^2 - b^2}{2 \cdot a \cdot \left|{}^2p_{24}\right|} $$

$$ b_2 = \arctan({}^2p_{24z} / {}^2p_{24x}) $$

$$ q_2 = \frac{\pi}{2} - (b_1 + b_2) $$

Inverse kinematics

Axis q5



$$x_3^T\, a = \left|x_3\right|\left|a\right|\cos q_5$$

$$q_5 = \arccos\left(x_3^T\, a\right)$$

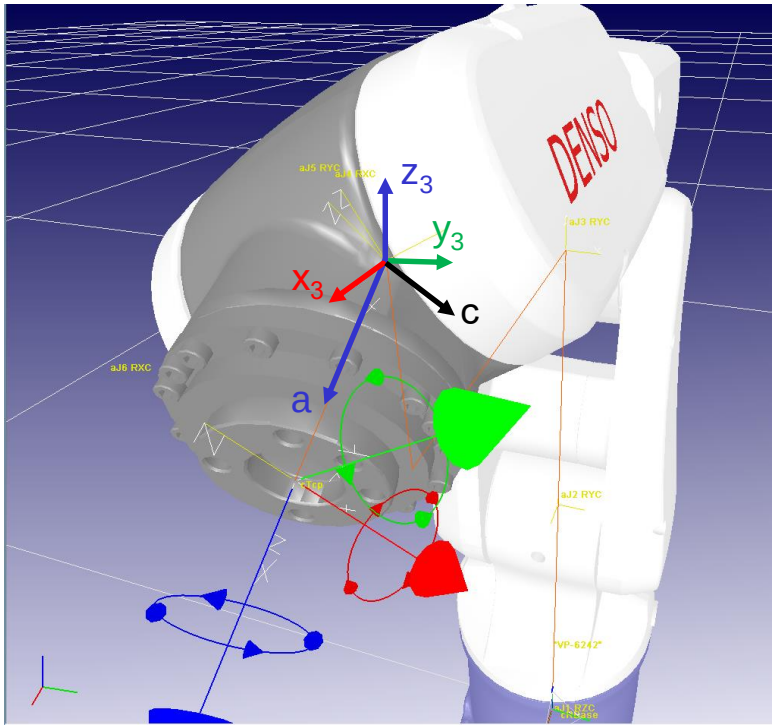
x_3 is first column of ${}^2_1A(q_1)\,{}^3_2A(q_2)\,{}^4_3A(q_3) = {}^4_1A(q_1, q_2, q_3)$

Inverse kinematics

Axis q4

$$c = \frac{x_3 \times a}{\left| x_3 \times a \right|}, \ \left| q_4 \right| = \arccos(\mathrm{y}_3^{\mathrm{T}} \mathrm{c})$$

$$\alpha = \arccos(\mathrm{z}_3^{\mathrm{T}} \mathrm{c}), \ \alpha > \pi/2 \Rightarrow q_4 = -\left| q_4 \right|, \ \alpha \le \pi/2 \Rightarrow q_4 = \left| q_4 \right|$$



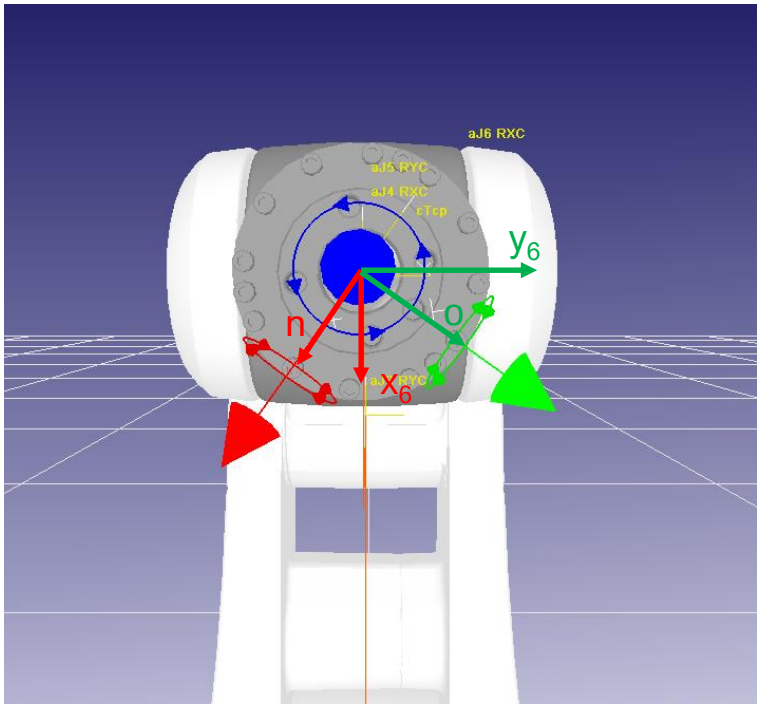
Inverse kinematics

Axis q6

y_6 is 2nd column of ${}^{2}_{1}A(q_1)\,{}^{3}_{2}A(q_2)\,{}^{4}_{3}A(q_3)\,{}^{5}_{4}A(q_4)\,{}^{6}_{5}A(q_5) = {}^{6}_{1}A = \begin{pmatrix} {}^{1}\underline{x}_6 & {}^{1}\underline{y}_6 & {}^{1}\underline{z}_6 & {}^{1}\underline{p}_6 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

$$q_6 = \arccos\left(y_6^T\ o\right)$$

$$\alpha = \arccos(y_6^T\ n),\ \ \alpha > \pi/2 \Rightarrow q_6 = -|q_6|,\ \ \alpha \le \pi/2 \Rightarrow q_6 = |q_6|$$



Youtube:
Inverse Kinematics for the KUKA KR60 robot with the same principle

https://www.youtube.com/watch?v=3s2x4QsD3uM

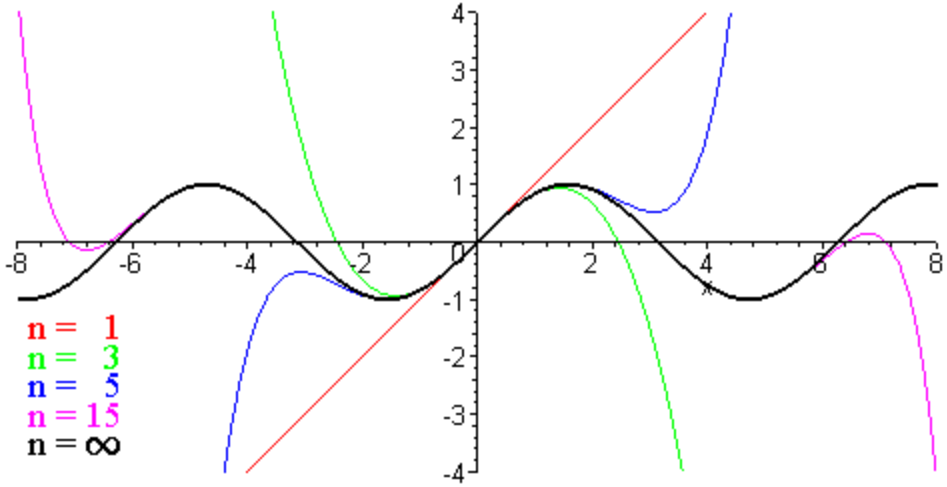
Inverse kinematics via jacobian

Taylor series of 1D-function at point a

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

$$f(x) = T_n(x) + R_n(x)$$

$$T_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k$$



Example: sinus function at a=0

Numerical solution via jacobian matrix

$$x + \Delta x = f(\theta + \Delta\theta) = f(\theta) + \left(\frac{\partial f_i(\theta)}{\partial \theta_j}\right)\Delta\theta + \ldots.$$

$$x + \Delta x \approx f(\theta) + \left(\frac{\partial f_i(\theta)}{\partial \theta_j}\right)\Delta\theta$$

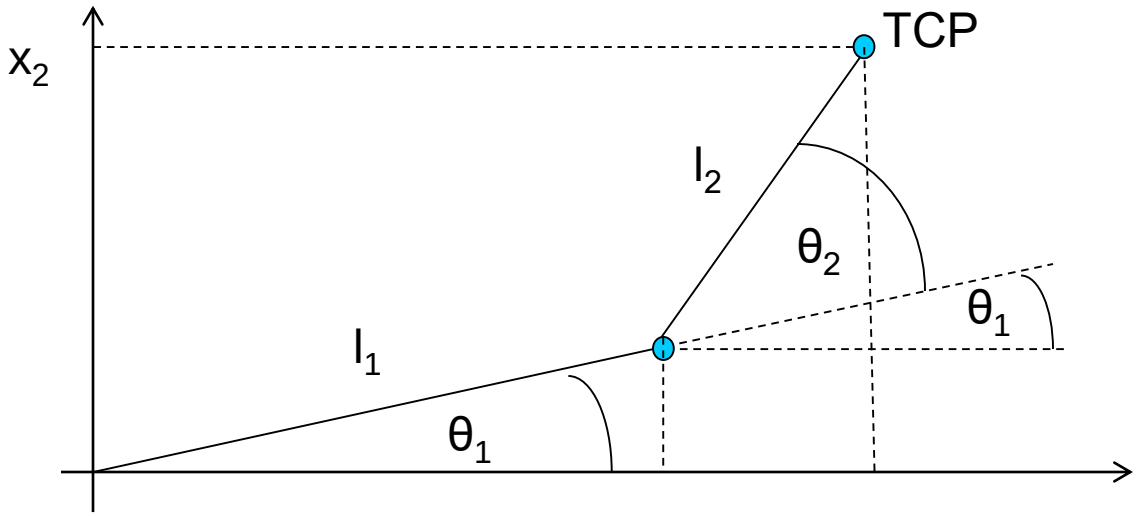
$$\Delta x \approx \left(\frac{\partial f_i(\theta)}{\partial \theta_j}\right)\Delta\theta = J(\theta)\Delta\theta$$

it follows:

$$\Delta\theta \approx J^{-1}(\theta)\Delta x$$

Inverse kinematics via jacobian

Example: Direct kinematics for 2 DOF robot with rotational DOF



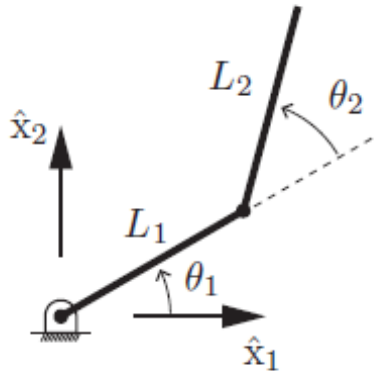
$$f(\theta) = f(\theta_1, \theta_2) = \begin{pmatrix} f_1(\theta_1, \theta_2) \\ f_2(\theta_1, \theta_2) \end{pmatrix} = x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) \\ l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) \end{pmatrix}$$

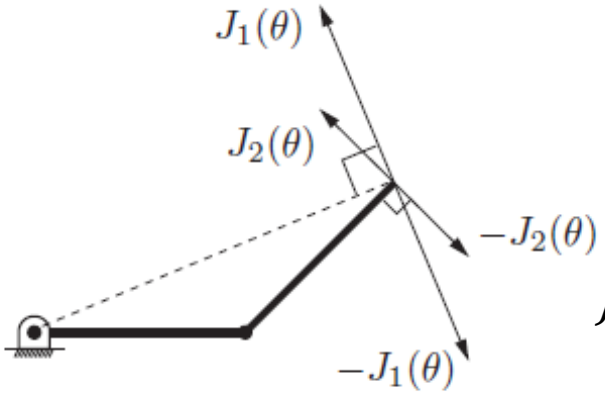
a) Partial derivatives:

$$J(\theta) = J(\theta_1, \theta_2) = \begin{pmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} \\ \frac{\partial f_2}{\partial \theta_1} & \frac{\partial f_2}{\partial \theta_2} \end{pmatrix} = \begin{pmatrix} -l_1 \sin(\theta_1) - l_2 \sin(\theta_1 + \theta_2) & -l_2 \sin(\theta_1 + \theta_2) \\ l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) & l_2 \cos(\theta_1 + \theta_2) \end{pmatrix}$$

Inverse kinematics via jacobian

Geometric interpretation of columns of jacobian





$$\theta_1 = 0, \theta_2 = 45, l_1 = l_2 = 1$$

$$J(\theta_1, \theta_2) = \begin{pmatrix} -0.71 & -0.71 \\ 1.71 & 0.71 \end{pmatrix}$$

$$f(\theta) = f(\theta_1, \theta_2) = x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) \\ l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) \end{pmatrix}$$

a) Partial derivatives:

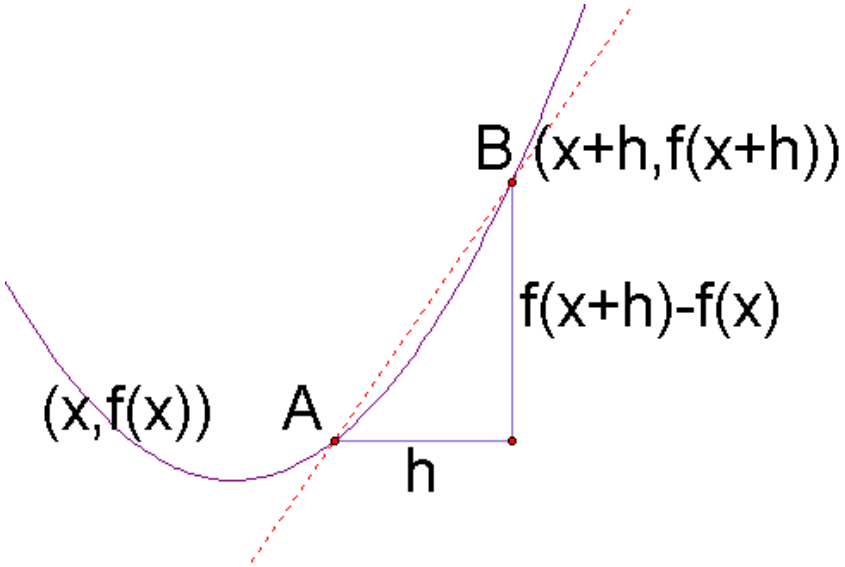
$$J(\theta) = J(\theta_1, \theta_2) = \begin{pmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} \\ \frac{\partial f_2}{\partial \theta_1} & \frac{\partial f_2}{\partial \theta_2} \end{pmatrix} = \begin{pmatrix} -l_1 \sin\theta_1 - l_2 \sin(\theta_1 + \theta_2) & -l_2 \sin(\theta_1 + \theta_2) \\ l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) & l_2 \cos(\theta_1 + \theta_2) \end{pmatrix}$$

One dimensional function

Example: f(x) is a nonlinear function.

Knowing A=(x,f(x)) and B=(x+h,f(x+h)) we want to find an estimate of the derivative curve.

A is the point (x, f(x)) and B is a point on the function a short distance **h** from A.
This gives B the coordinates (x+h, f(x+h))



$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$f(x + dx) - f(x) \approx f'(x) * dx$$

Inverse kinematics via jacobian

b) Numerical difference quotient:

$$J(\theta) = J(\theta_1, \theta_2) = \begin{pmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} \\ \frac{\partial f_2}{\partial \theta_1} & \frac{\partial f_2}{\partial \theta_2} \end{pmatrix} = \begin{pmatrix} \frac{f_1(\theta_1+d\theta_1)-f_1(\theta_1)}{d\theta_1} & \frac{f_1(\theta_2+d\theta_2)-f_1(\theta_2)}{d\theta_2} \\ \frac{f_2(\theta_1+h)-f_2(\theta_1)}{d\theta_1} & \frac{f_2(\theta_2+d\theta_2)-f_2(\theta_2)}{d\theta_2} \end{pmatrix}$$

$$= \left(\frac{f(\theta_1+d\theta_1)-f(\theta_1)}{d\theta_1} \quad \frac{f(\theta_2+d\theta_2)-f(\theta_2)}{d\theta_2}\right)$$

Inverse kinematics via jacobian

Differential relationship between changes of angles $d\theta$ and changes of TCP pose dx

$$J(\theta)d\theta \approx dx$$

We are interested on the inverse relationship: Which changes in $d\theta$ corresponds to TCP changes in dx ?

$$d\theta \approx J^{-1}(\theta)dx$$

$$\theta_{new} = \theta_{old} + d\theta$$

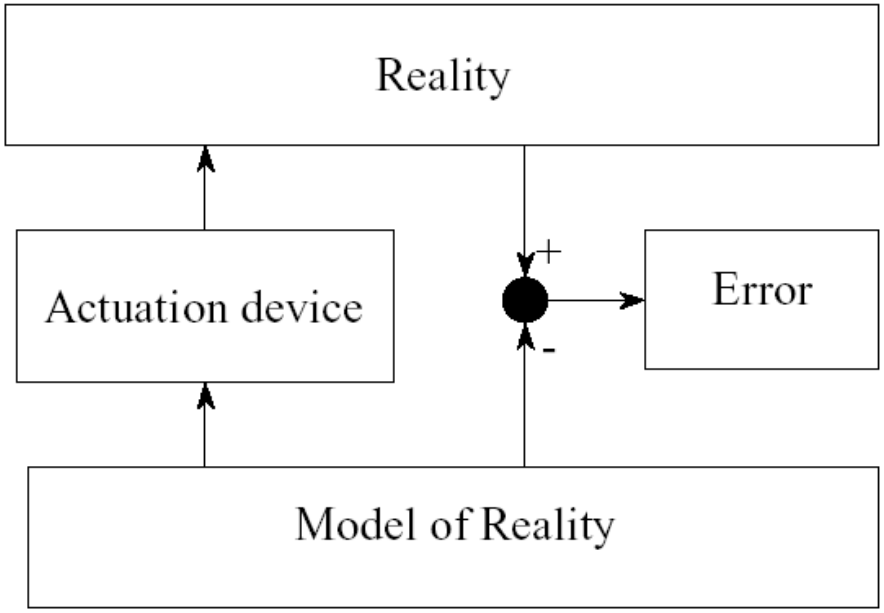
- Typically the system is underdetermined (especially for redundant systems)
- You have more degees of freedom in C-space (number of axes) axes than in task space (cartesian degrees of freedom)
- There are an infinite number of solutions for $d\theta$
- You can incorporate additional conditions to the system
- For example: collision avoidance in scene *ikAndObstacleAvoidance.ttt*

Robot calibration

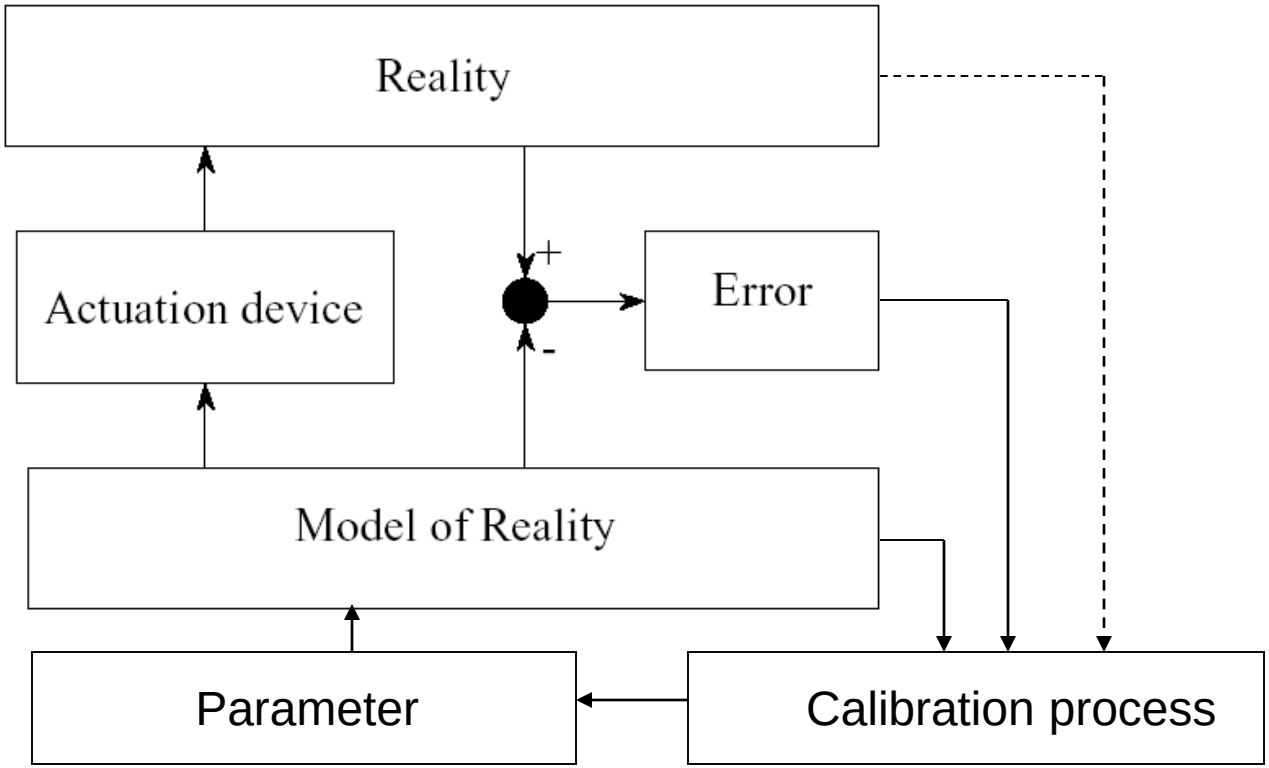
What is robot calibration?

Robot calibration is a process used to improve the accuracy of robots, particularly industrial robots which are highly repeatable but not accurate.

Robot calibration is the process of identifying certain parameters in the kinematic structure of an industrial robot, such as the relative position of robot links.



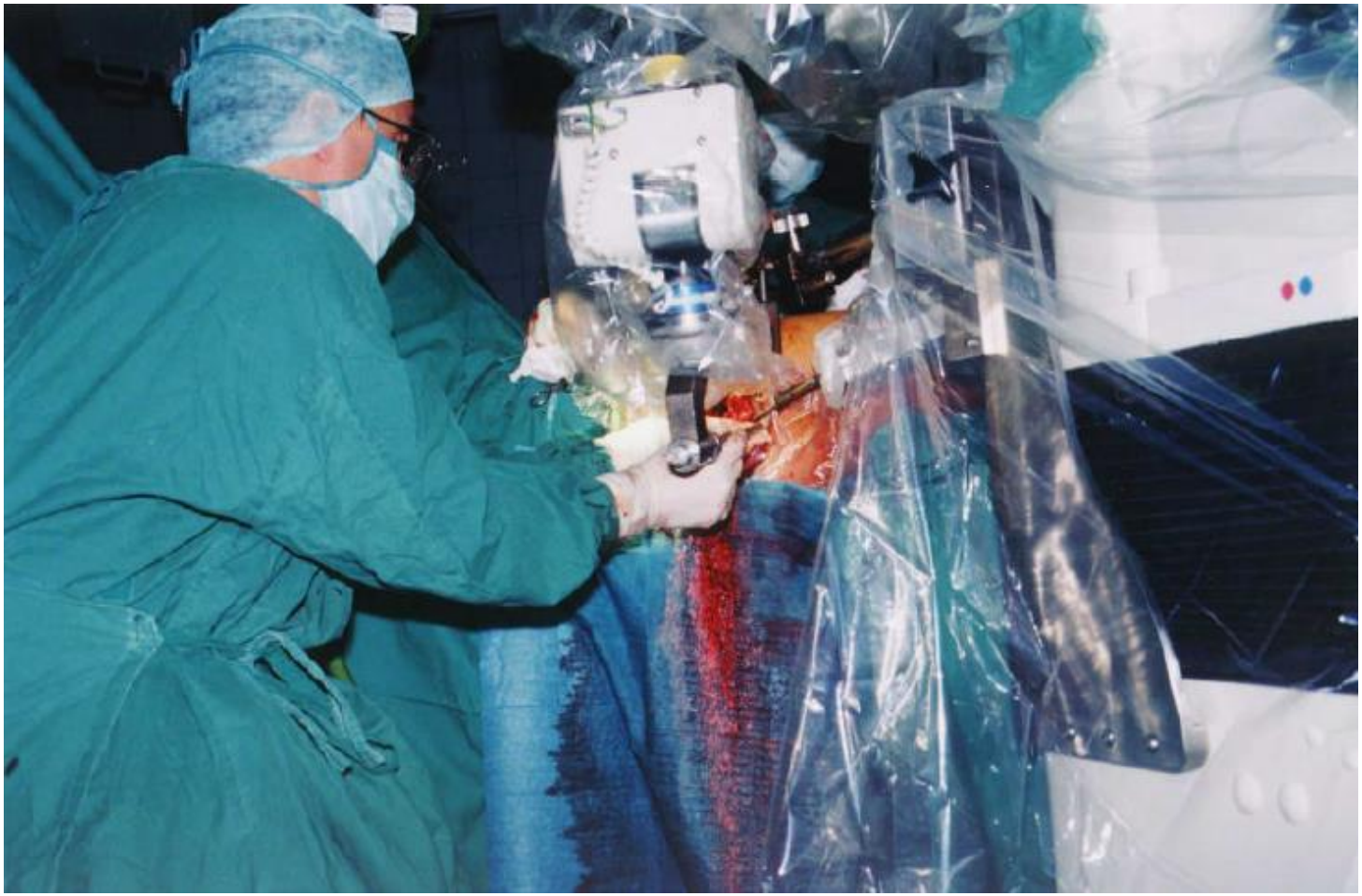
Robot calibration



Solution approach: Estimation of parameters

Robot calibration

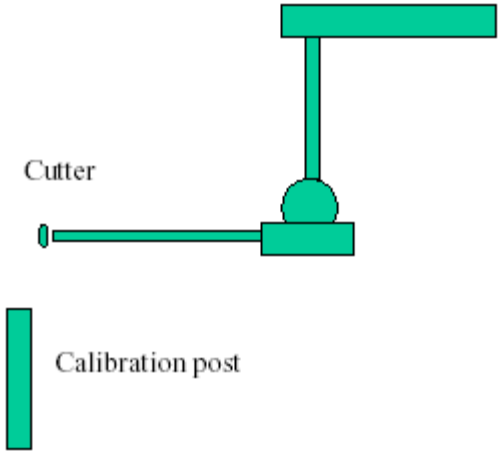
Example: Robodoc – robot assisted surgery



Robot calibration

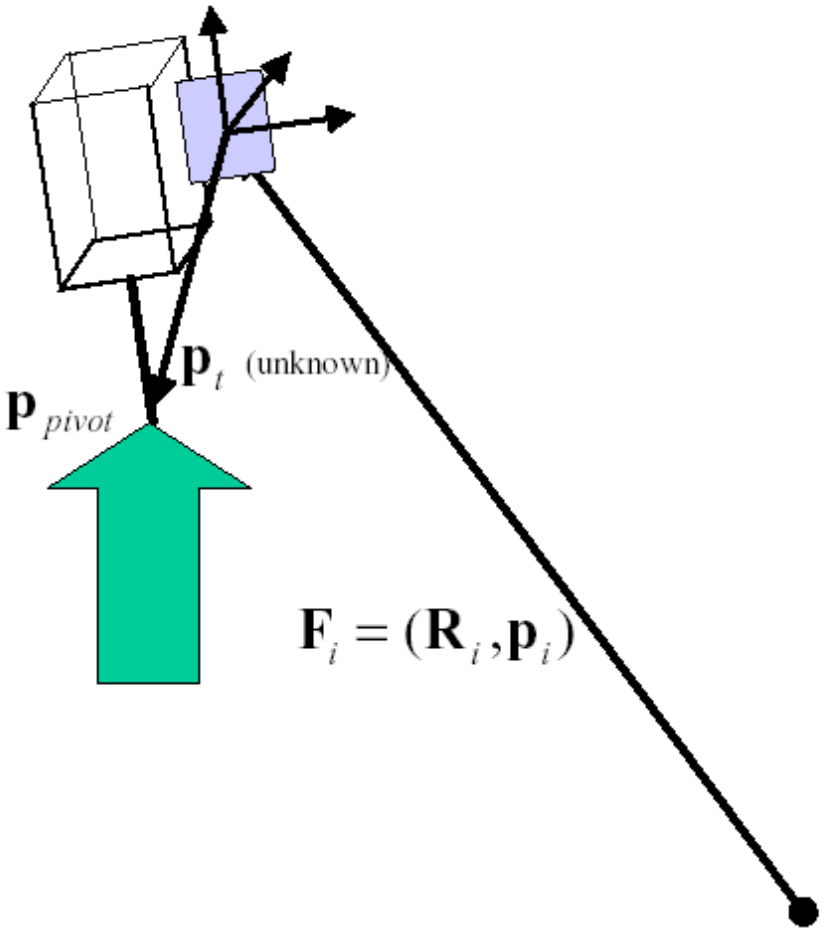
Beispiel: Robodoc – robot assisted surgery

- Robot itself is highly accurate
- custom specific tool is attached to the robots mechanical interface
- Goal: exact determination of the pose (position and orientation) of the cutter‘s tip



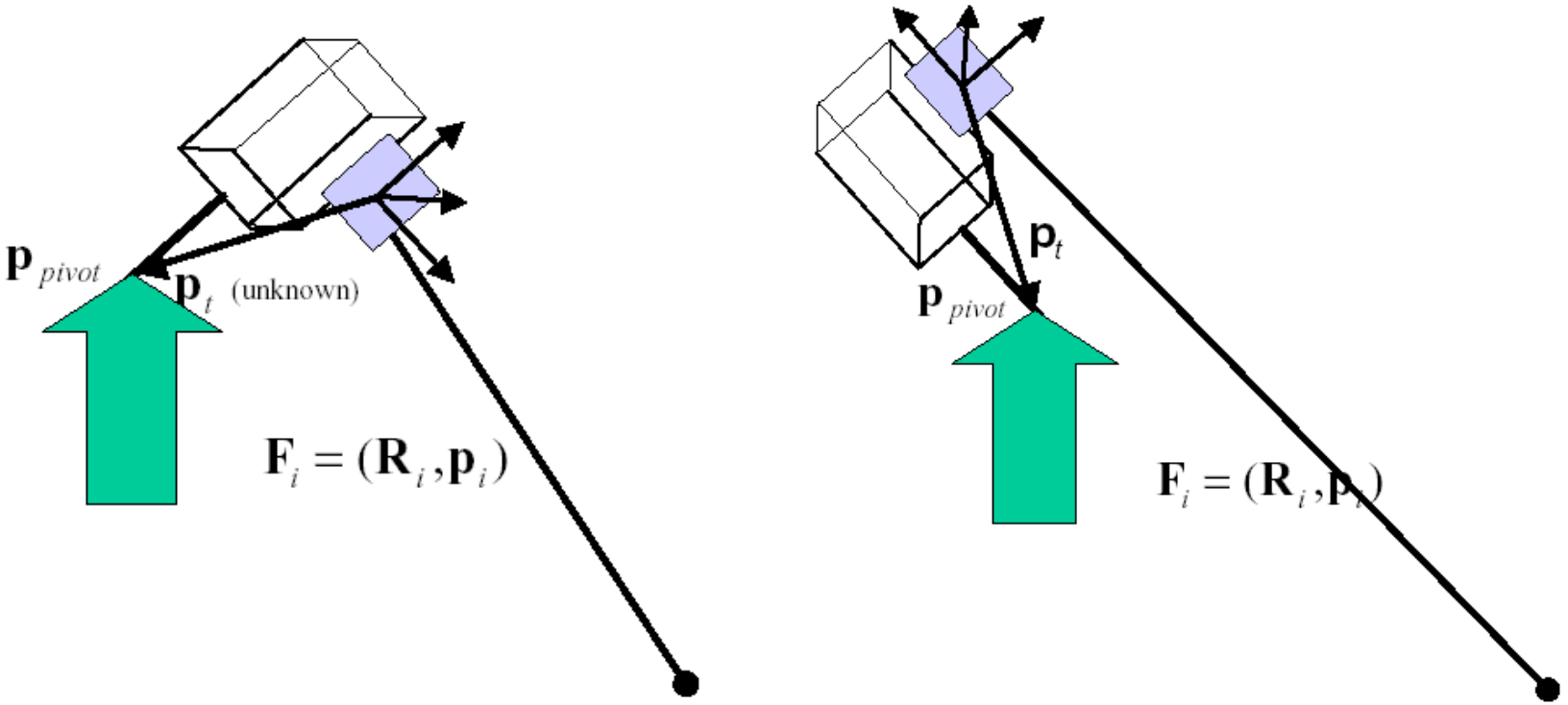
Robot calibration

Calibration of a pointing device:



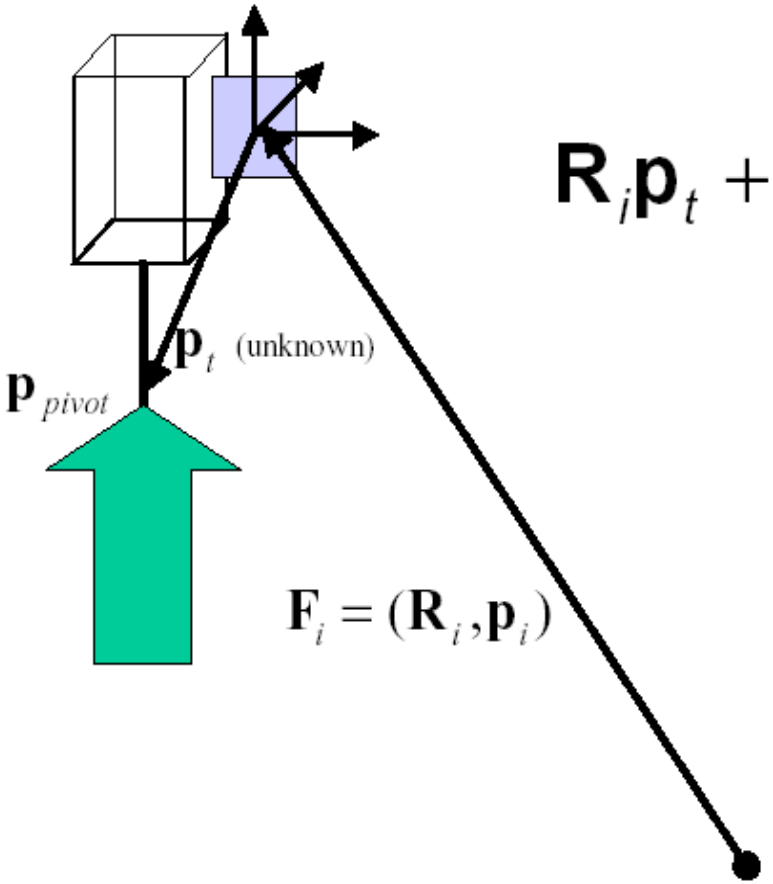
Robot calibration

Calibration of a pointing device:



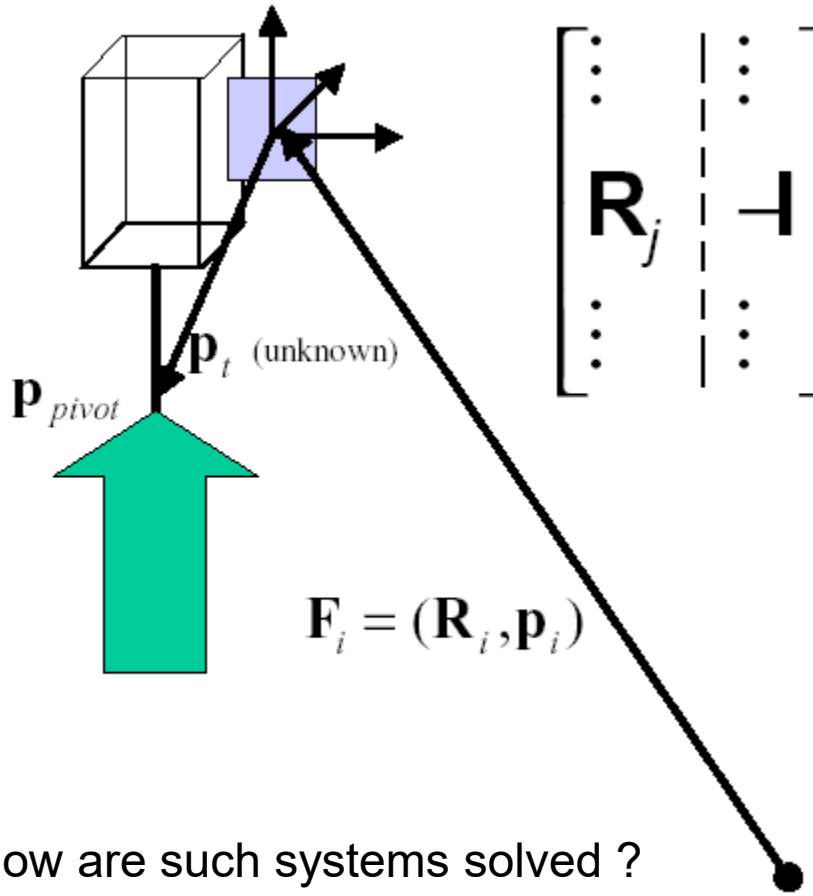
Robot calibration

Calibration of a pointing device:



$$\vdots$$
$$\mathbf{R}_i\mathbf{p}_t + \mathbf{p}_i = \mathbf{p}_{pivot}$$
$$\vdots$$

Robot calibration



$$\begin{bmatrix} \vdots & \vdots \\ \mathbf{R}_j & -\mathbf{I} \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} \mathbf{p}_t \\ \hline \mathbf{p}_{pivot} \end{bmatrix} \cong \begin{bmatrix} \vdots \\ -\mathbf{p}_j \\ \vdots \end{bmatrix}$$

How are such systems solved ?

Robot calibration

Write all the measurements in a compact notation:

$$P_i + R_i P_t - P_{pivot} \cong 0, i = 1..n$$

$$\rightarrow \underbrace{\begin{pmatrix} R_1 & -I \\ \vdots & \vdots \\ R_n & -I \end{pmatrix}}_{\substack{coefficient\ matrix \\ A}} \underbrace{\begin{pmatrix} P_t \\ P_{pivot} \end{pmatrix}}_{\substack{Parameter \\ vector \\ x}} \cong \underbrace{\begin{pmatrix} -P_1 \\ \vdots \\ -P_n \end{pmatrix}}_{\substack{observations \\ b}}$$

Leading to:

$$Ax \approx b$$

- Typically, such a system is overdetermined
- You have less variables than equations to fulfill

- Try to minimize length of error: $e^T e$
- x is the unique vector of smallest magnitude which minimizes $|Ax - b|^2 = e^T e$

- How are such systems solved?

Least square solution

In general, given a overdetermined linear system $Ax - b = e$:

$$\begin{aligned} a_1^T x - b_1 &= e_1 \\ &\vdots \\ a_i^T x - b_i &= e_i \\ &\vdots \\ a_m^T x - b_m &= e_m \end{aligned}$$

Where a_i is the i -th row of A

sometimes denoted by $Ax \approx b$

We are going to find a solution: $\min \sum_i^m e_i^2 = e^T e \ = (Ax - b)^T (Ax - b)$

which minimizes the sum of the squares

Least square solution

- There exist several methods for solving the overdetermined system

$$Ax = b$$

- A simple method uses the transpose of the matrix A

$$\begin{aligned} Ax &= b \\ A^T A\, x &= A^T b \\ x &= (A^T A)^{-1} A^T b \\ x &= A^+ b \end{aligned}$$

- A^+ is called the pseudo inverse of A, which is not necessarily quadratric
- $A^T A$ is symmetric
- The inverse of $A^T A$ must exist, if so $A^T A$ is positive definite (has positive eigenvalues)
- Better (more robust) methods exist, which are based on orthogonal transformations

Least square solution

Why does this solution minimizes the error:

We are searching the solution vector x of the overdetermined system:

$Ax = b + e$, such that $e^T e \rightarrow min$:

$$
\begin{aligned}
e^T e &= (Ax - b)^T (Ax - b) \\
&= x^T A^T Ax - x^T A^T b - b^T Ax + b^T b \\
&= x^T A^T Ax - x^T A^T b - \left((b^T Ax)^T\right)^T + b^T b \\
&= \underbrace{x^T A^T Ax}_{1x1} - \underbrace{x^T A^T b}_{1x1} - \underbrace{(x^T A^T b)^T}_{1x1} + \underbrace{b^T b}_{1x1} \\
&= \underbrace{x^T A^T Ax}_{1x1} - \underbrace{x^T A^T b}_{1x1} - \underbrace{x^T A^T b}_{1x1} + \underbrace{b^T b}_{1x1} \\
&= \underbrace{x^T A^T Ax}_{1x1} - \underbrace{2\, x^T A^T b}_{1x1} + \underbrace{b^T b}_{1x1}
\end{aligned}
$$

$$\text{min: } \frac{d}{dx}\left(x^T A^T Ax - 2\, x^T A^T b + b^T b \right) = 0$$

$$2\, A^T A\, x - 2\, A^T b = 0$$

$$A^T A\, x = A^T b$$

$$x = (A^T A)^{-1} A^T b \quad \text{with} \quad \det(A^T A) \neq 0$$

Orthogonal transformations

important property of an orthogonal matrix: $Q^{-1} = Q^T$

conclusions from this property:

- the inverse of Q is equal to the transpose of Q
- different rows of Q are mutually perpendicular to each other
- Same rows have unit length
- mapping of Q does not change length of vector x

$$Q^{-1} = Q^T$$

$$Q = [q_1\, q_2\, \dots\, q_n]$$

$$q_i^T q_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

$$|Qx| = \sqrt{(Qx)^T (Qx)} = \sqrt{x^T Q^T Q x} = \sqrt{x^T x} = |x|$$

Singular value decomposition (SVD)

Given an arbitrary *m×n* matrix **A**, then there exist orthogonal matrices **U, V** and a diagonal matrix **S** with the following decomposition:

$$A_{m\times n} = U_{m\times m} \begin{vmatrix} S_{n\times n} \\ 0_{(m-n)\times n} \end{vmatrix} V^T, \quad \text{if } m \geq n$$

$$A_{m\times n} = U_{m\times m} \left| S_{n\times n} \quad 0_{m\times(n-m)} \right| V^T, \quad \text{if } m < n$$

Singular value decomposition (SVD)

$$A_{m\times n}x = b$$

$$U_{m\times m}\begin{vmatrix} S_{n\times n} \\ 0_{(m-n)\times n} \end{vmatrix} V^T x = b, \quad \text{with } y = V^T x$$

$$\begin{vmatrix} S_{n\times n} \\ 0_{(m-n)\times n} \end{vmatrix} y = U^T_{m\times m} b, \quad \text{with } y = V^T x$$

Solve this for **y** (trivial, since **S** is a diagonal matrix),
Then calculate:

$$Vy = VV^T x = x$$

Overview:

1. Introduction
2. Definitions & Terminology
3. Rigid body motions and kinematics (inverse and forward kinematics)
4. **Trajectory generation**
5. Motion planning

Trajectory generation

Path:

- purely geometric description of the sequence of configurations achieved by the robot

Trajectory:

- specification of the robot position as a function of time is called a trajectory
- combination of a path and a time scaling, which specifies the times when those configurations are reached

Several types:

- point-to-point straight-line trajectories in joint space (PTP)
- trajectories in task space
- trajectories passing through a sequence of timed via points
- minimum-time trajectories along specified paths taking actuator limits into consideration

- finding paths that avoid obstacles (motion planning)

Trajectory generation

Path:

- $\theta(s)$ maps a geometric path parameter s, assumed to be 0 at the start of the path and 1 at the end, to a point in the robot's configuration space

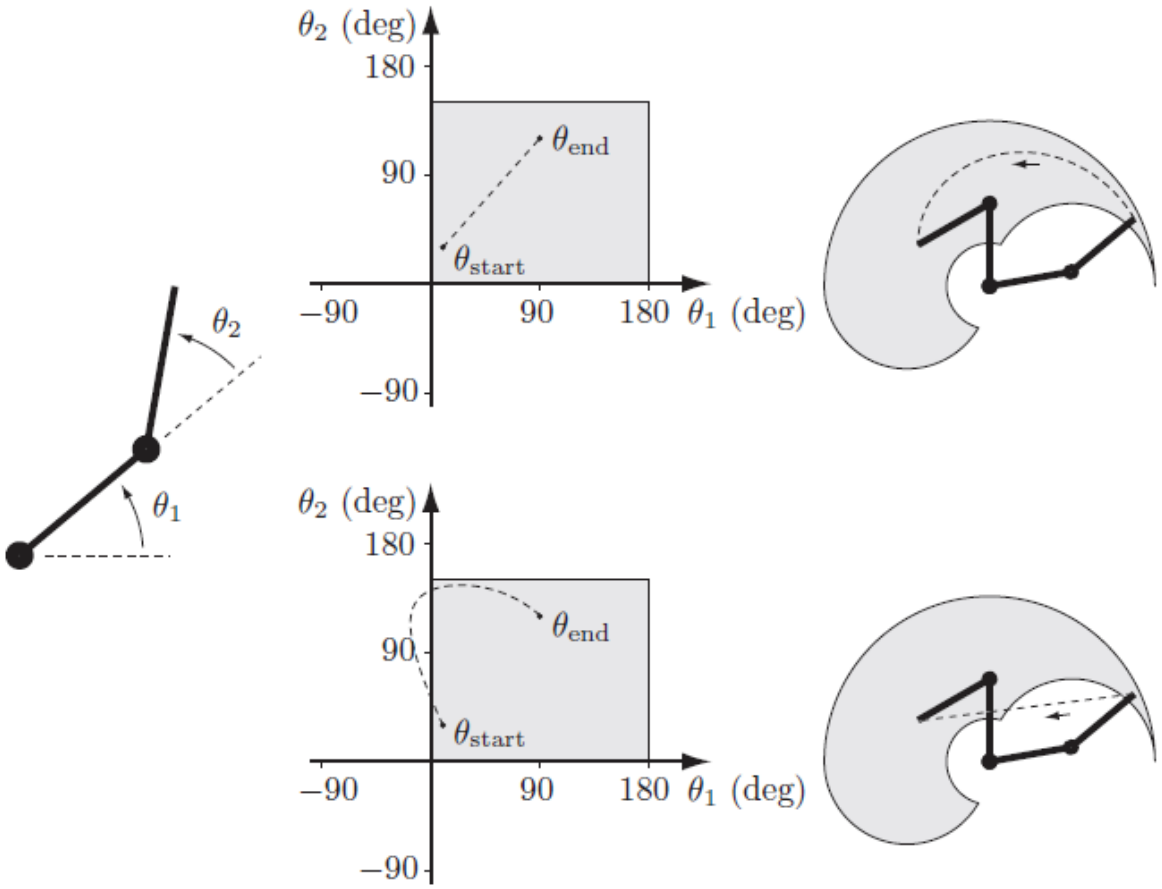
Trajectory:

- A time scaling $s(t)$ assigns a value s to each time $t \in [0, T], s: [0, T] \rightarrow [0,1]$
- Separate the role of geometric path parameter s from the time parameter t
- Together, a path and a time scaling define a trajectory $\theta(s(t))$

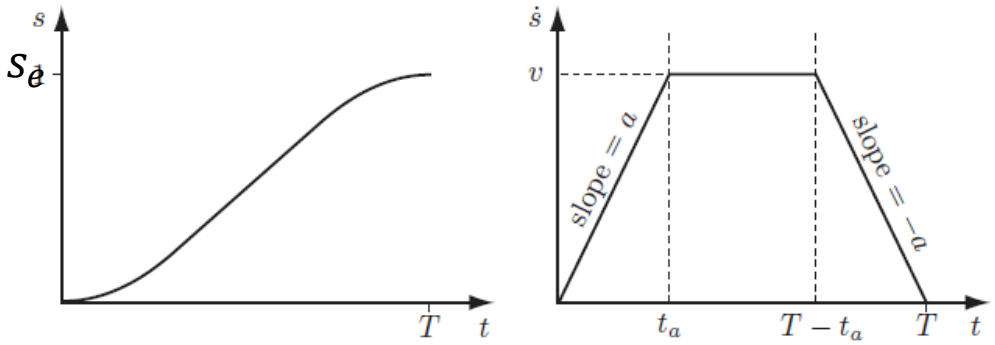
Simple trajectory: point-to-point straight-line trajectories in joint space (PTP) from start configuration θ_s to end configuration θ_e

$$\theta(s) = \theta_s + s(\theta_e - \theta_s), s \in [0,1]$$

Trajectory generation



Trajectory generation



Trapezoidal velocity profile consists of 3 phases:

$$s_e = |q_f - q_0|$$
$$t_a = \frac{v}{a}$$
$$T = \frac{s_e}{v} + t_a$$
$$t_v = T - t_a$$

$$0 \le t \le t_a: \quad \ddot{s}(t) = a, \dot{s}(t) = at, s(t) = \frac{1}{2}at^2$$

$$t_a \le t \le t_v: \quad \ddot{s}(t) = 0, \dot{s}(t) = v, s(t) = vt - \frac{1}{2}\frac{v^2}{a}$$

$$t_v \le t \le T: \quad \ddot{s}(t) = -a, \dot{s}(t) = v - a(t - t_v), s(t) = vt_v - \frac{a}{2}(T - t)^2$$

If s_e is small, *v* can not be reached: triangle profile (2 phases):

$$s_e = t_a v = \frac{v^2}{a} \Rightarrow v = \sqrt{a s_e}$$

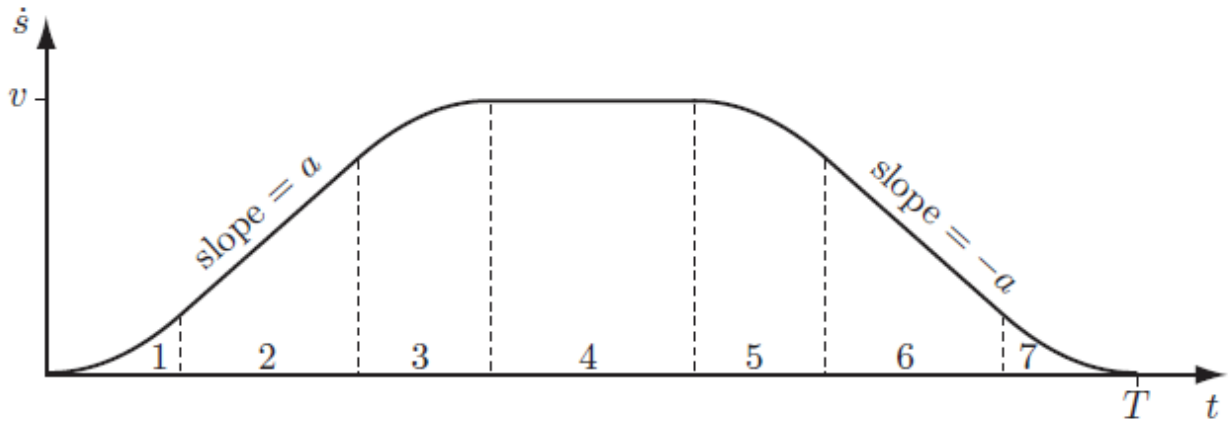
Trajectory generation

trapezoidal motions cause discontinuous jumps in acceleration at

$$t \in \{0, t_a, T - t_a, T\}$$

Jerk $J = \dddot{s}(t)$ is unbounded

- A solution is a smoother S-curve time scaling
- 7 phase velocity profile
  - a popular motion profile in motor control
  - avoids vibrations or oscillations induced by step changes in acceleration
  - Jerk is bounded



Trajectory generation

A robot has typically more than one axis

All axes need to **synchronized**, so that the axes begin and end their motion at the same time

Sketch of the algorithm:

- for each joint θ_i use its maximal velocity and acceleration to calculate its T_i
- Define the axis with the largest $T_{max} = \max\{T_i\}$ as the “leading” axis
- Scale the velocity of each other axes, so that they get the same T_{max}

Trajectory generation

robot user would like to control the motion between start and end point
several possibilities:

- straight line motion (TCP follows a straight line)
- circular motion (TCP follows a circle segment)
- spline motion

- inverse kinematics needs to be calculated at run time
- computational expensive (depending on the robot)

issues:

- interpolation of TCP position (linear change of coordinates)
- interpolation of orientation (depends on representation)

Trajectory generation - Bezier curves

Splines (Bezier splines) to describe the path for the TCP:

- Bezier curves are interesting for pathes with certain “smoothness” properties
- Defined by a series of control points (sometimes with associated weights)
- Weights pulls the curve towards the control points
- Bezier curves are based on Bernstein polynoms
- (we are looking at cubic polynoms: n=3)

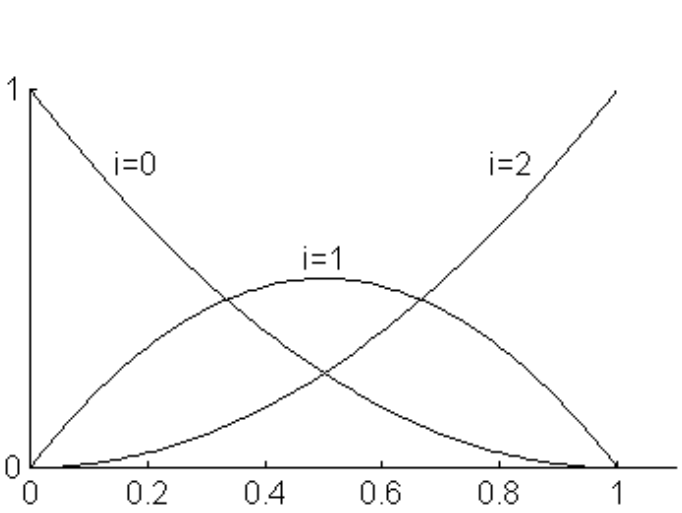
$$B_k^n(u) := \binom{n}{k}(1-u)^{n-k}u^k$$

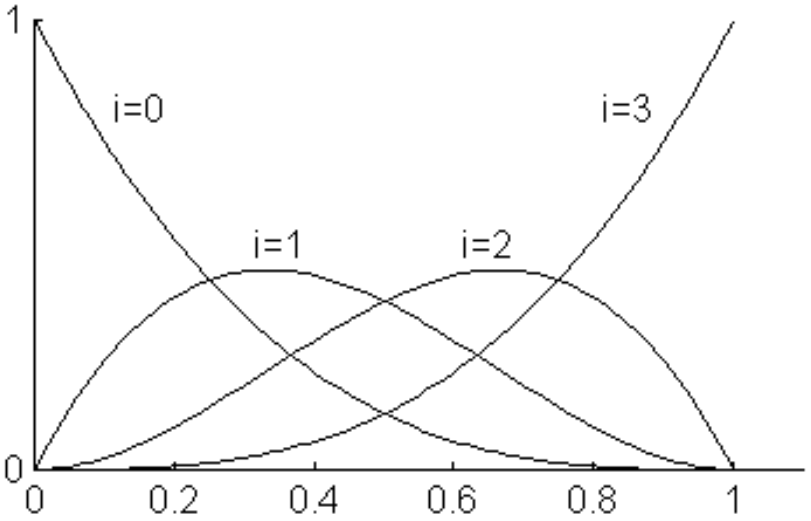
With normalised parameter interval [0,1]
Bernstein polynoms can motivated with the following equation:

$$1 = ((1-u)+u)^n = \sum_{k=0}^{n}\binom{n}{k}(1-u)^{n-k}u^k$$

Trajectory generation - Bezier curves

Bernstein polynoms of order n=2 (quadratic) or n=3 (cubic)





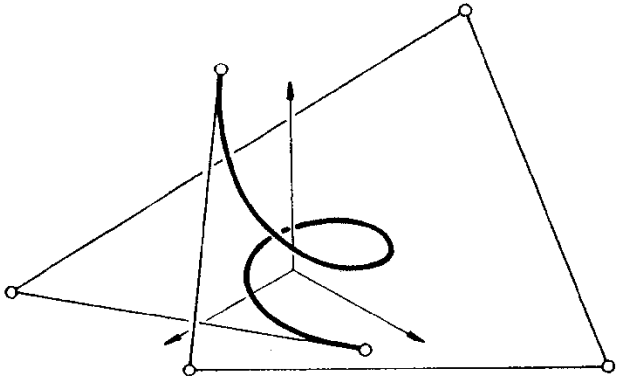
$$B_i^2(u), B_i^3(u)$$

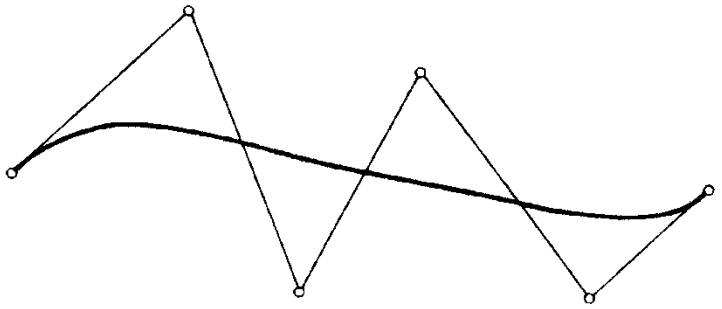
Trajectory generation - Bezier curves

Definition of Bezier curves are based on Bernstein polynoms:

$$X(u) = \sum_{i=0}^{n} b_i B_i^n(u)$$

- b_i are Bézier points or control points.
- The polygon connecting the control points are called control polygon
- Convex hull property: the Bezier curve is inside the convex hull of the control points
- Two examples of Bezier curves with associated control polygon





Trajectory generation - Bezier curves

de Casteljau-Algorithm for calculating points on the Bezier curve:
Geometrical interpretation: repetitive division of the control polygon
Calculation of: $X_n(u_0)$

$(1-u_0):u_0$

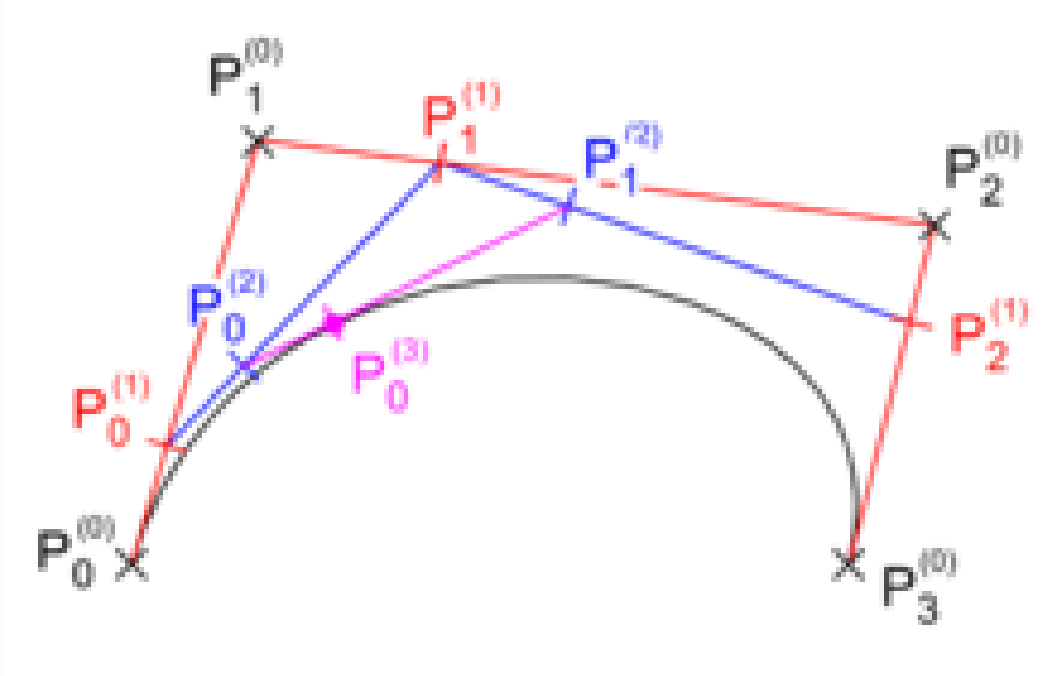
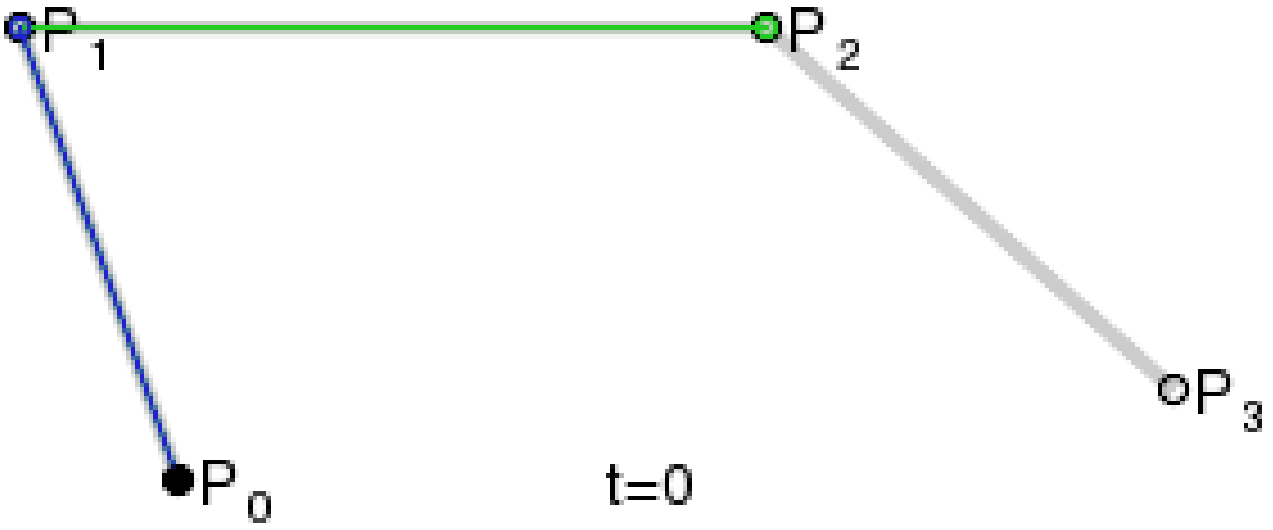


Figure: source Wikipedia

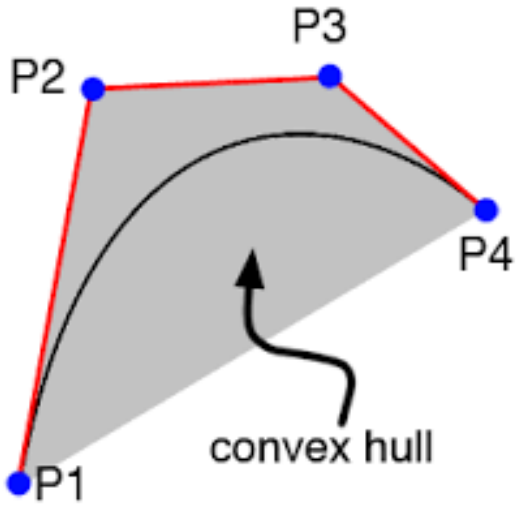
Trajectory generation - Bezier curves

Animation of the de Casteljau-algorithm:



Trajectory generation - Bezier curves

Further examples (convex hull property):



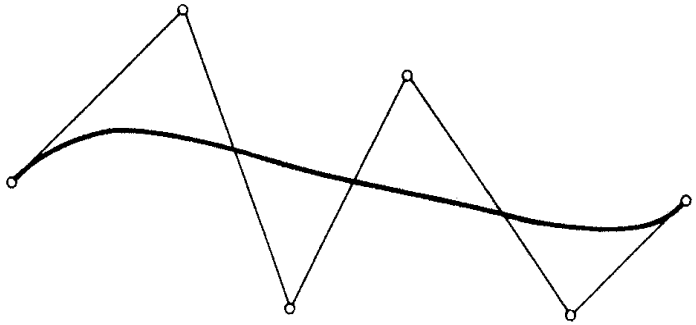
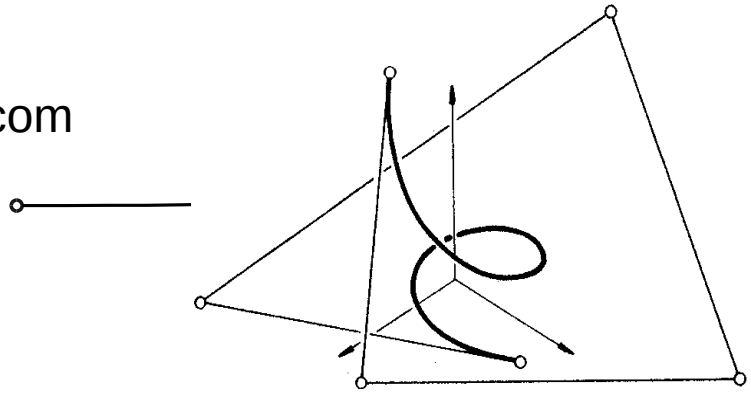


Figure: source www.scratchapixel.com



Trajectory generation - Bezier curves

Exercise:

Given are the (two-dimensional) control points b0, b1, b2 and b3 for a cubic Bezier curve:

b0 $=(0,0)^T$, b1 $=(1,0)^T$, b2 $=(1,2)^T$, b3 $=(0,2)^T$

Calculate the point on the curve X(0.5) via

$$X(u) = \sum_{i=0}^{n} b_i B_i^n(u)$$

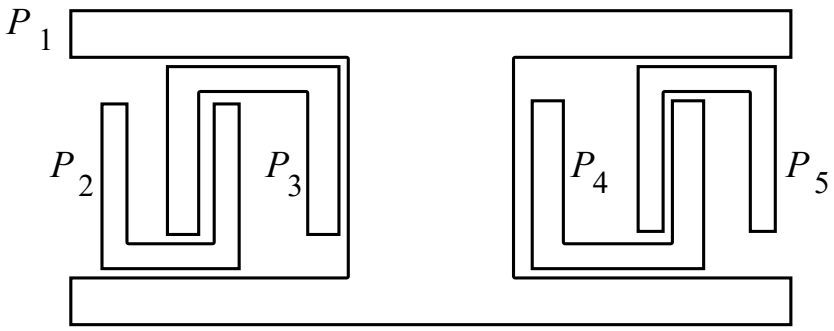
and geometrically with the de Casteljau algorithm

Overview:

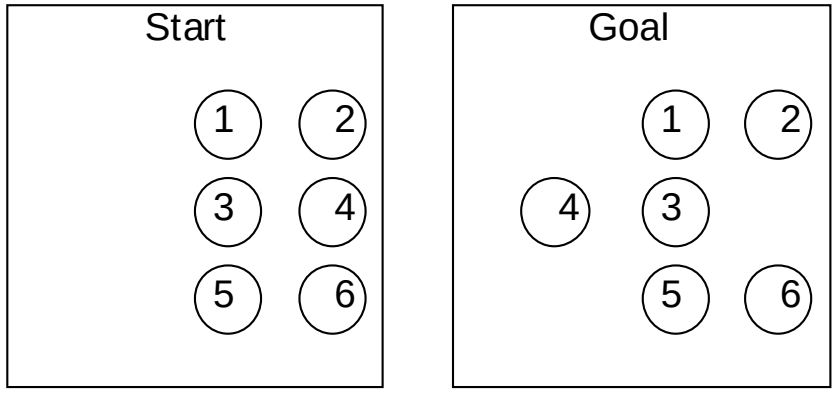
1. Introduction
2. Definitions & Terminology
3. Rigid body motions and kinematics (inverse and forward kinematics)
4. Trajectory generation
5. **Motion planning**

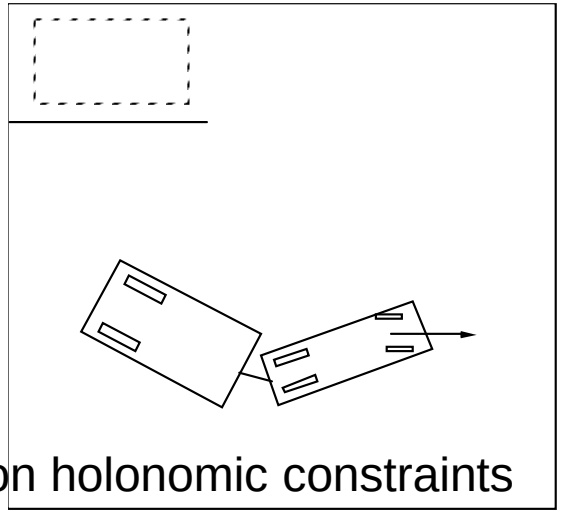
Motion Planning

assembly planning

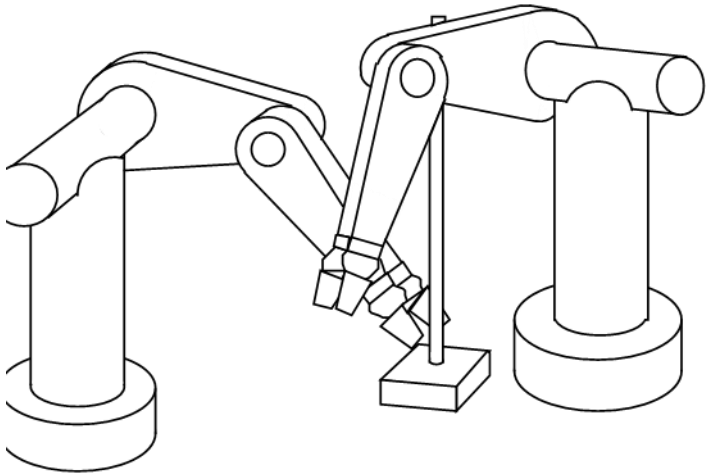


Mobile Robots

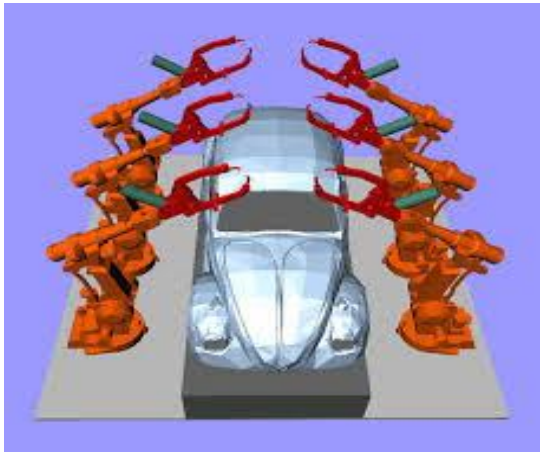


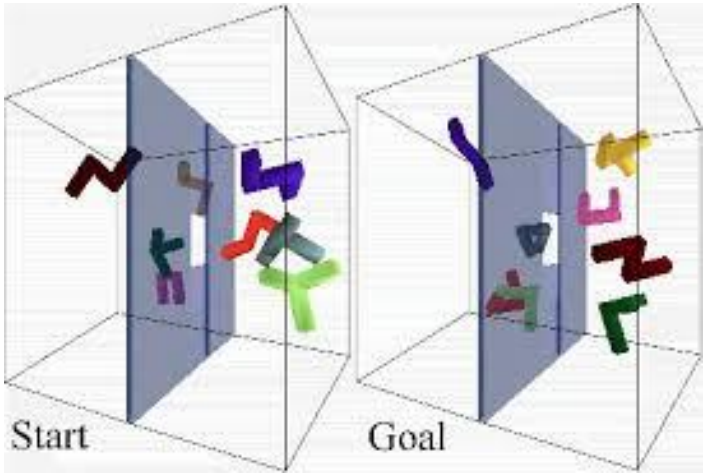


Non holonomic constraints

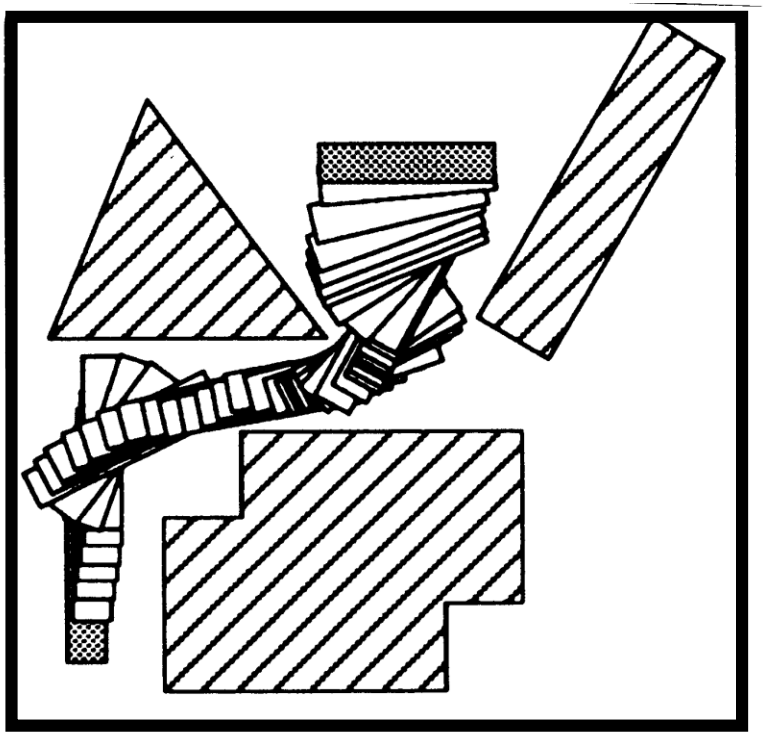


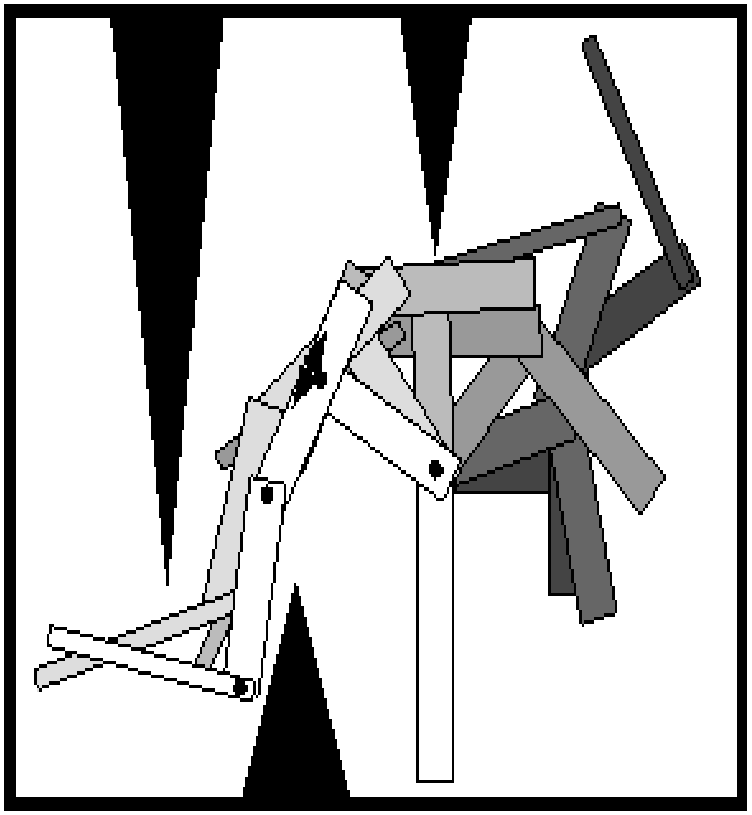
Motion Planning





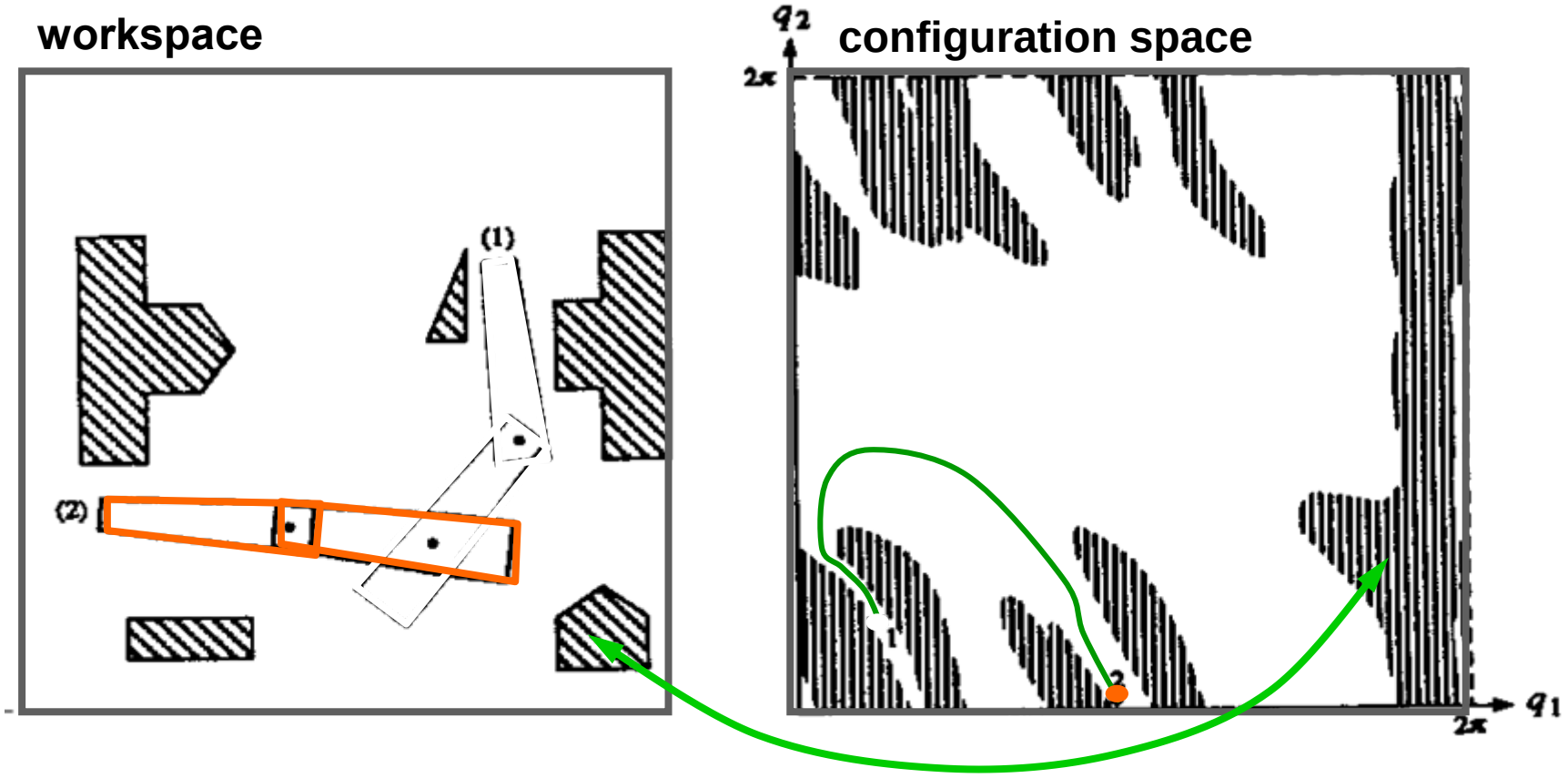
What is a path?



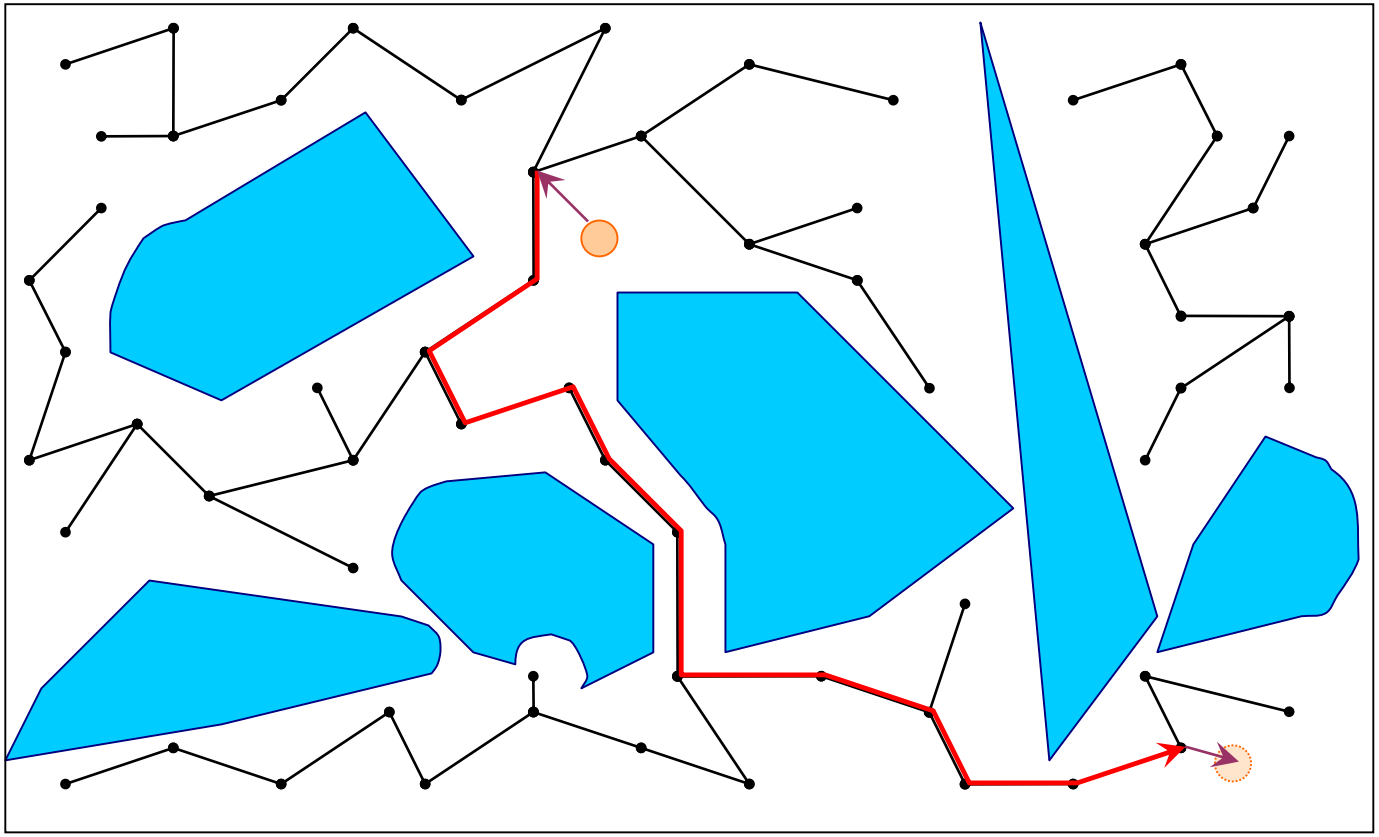


Rough idea of configuration space (c-space)

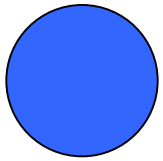
- Convert rigid robots, articulated robots, *etc.* into points
- Apply algorithms for moving points
- Mapping from the workspace to the configuration space



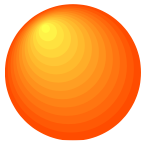
What is a Configuration Space ?



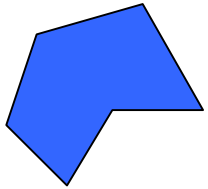
Degrees of Freedom



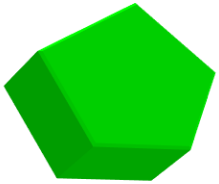
Disc robot:
2 dofs: (x, y)



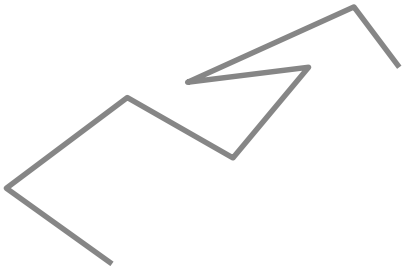
Ball robot (flyer):
3 dofs: (x, y, z)



Polygonal robot:
3 dofs: (x, y, θ)



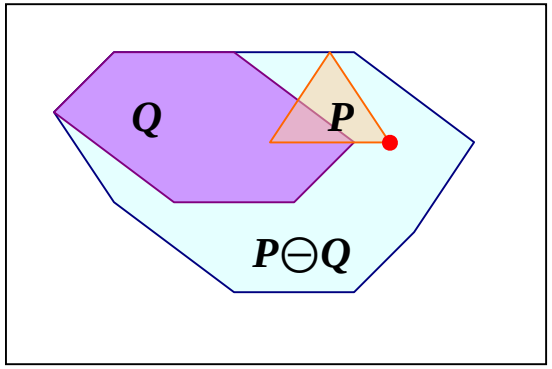
Polyhedral robot (flyer):
6 dofs: $(x, y, z, \theta_x, \theta_y, \theta_z)$

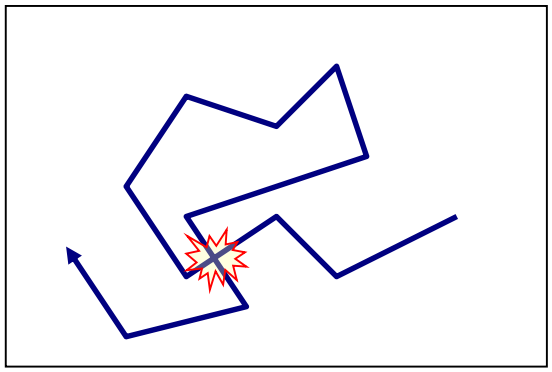


Poly-line robot:
n+2 dofs: $(x, y, \theta_1, \theta_2, \ldots, \theta_n)$

Forbidden Configurations

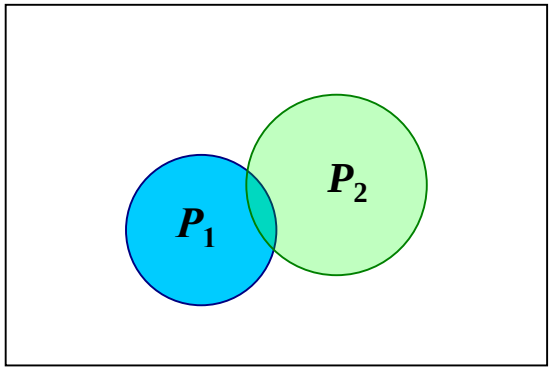
Due to robot–obstacle collisions
(configuration within a C-obstacle)





Due to self-collisions

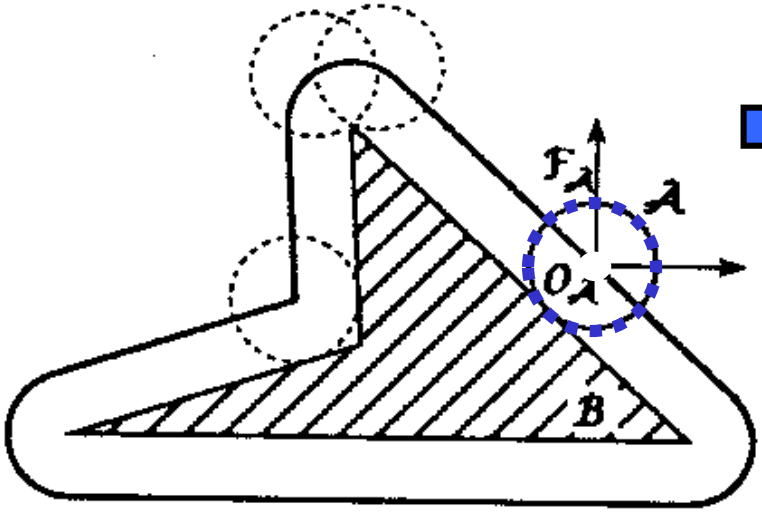
Due to robot–robot collisions
(in case of multiple robots)



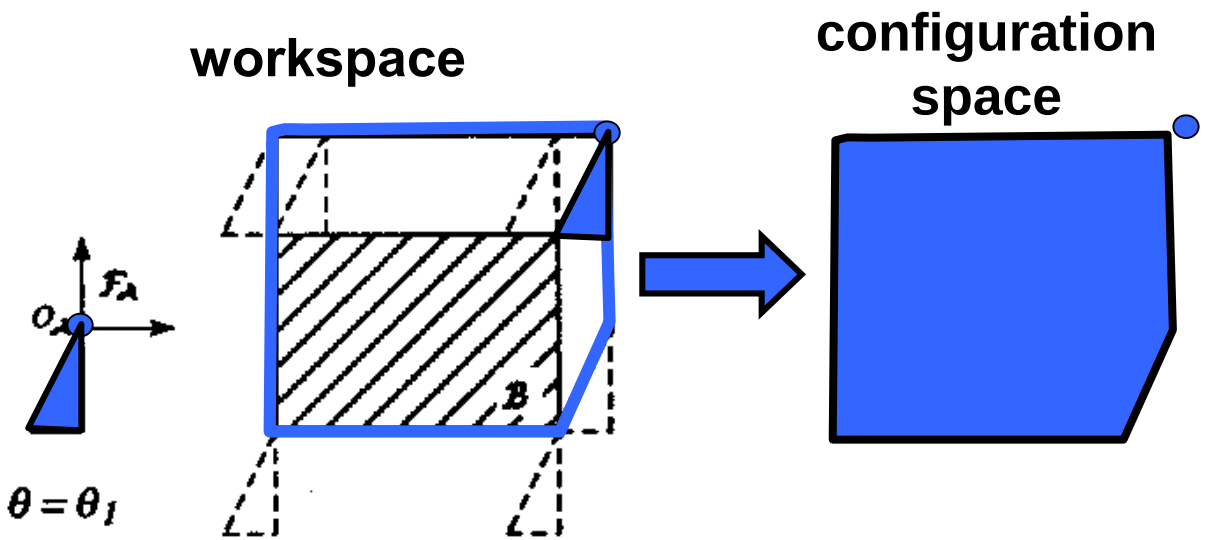
Disc in 2-D workspace

workspace

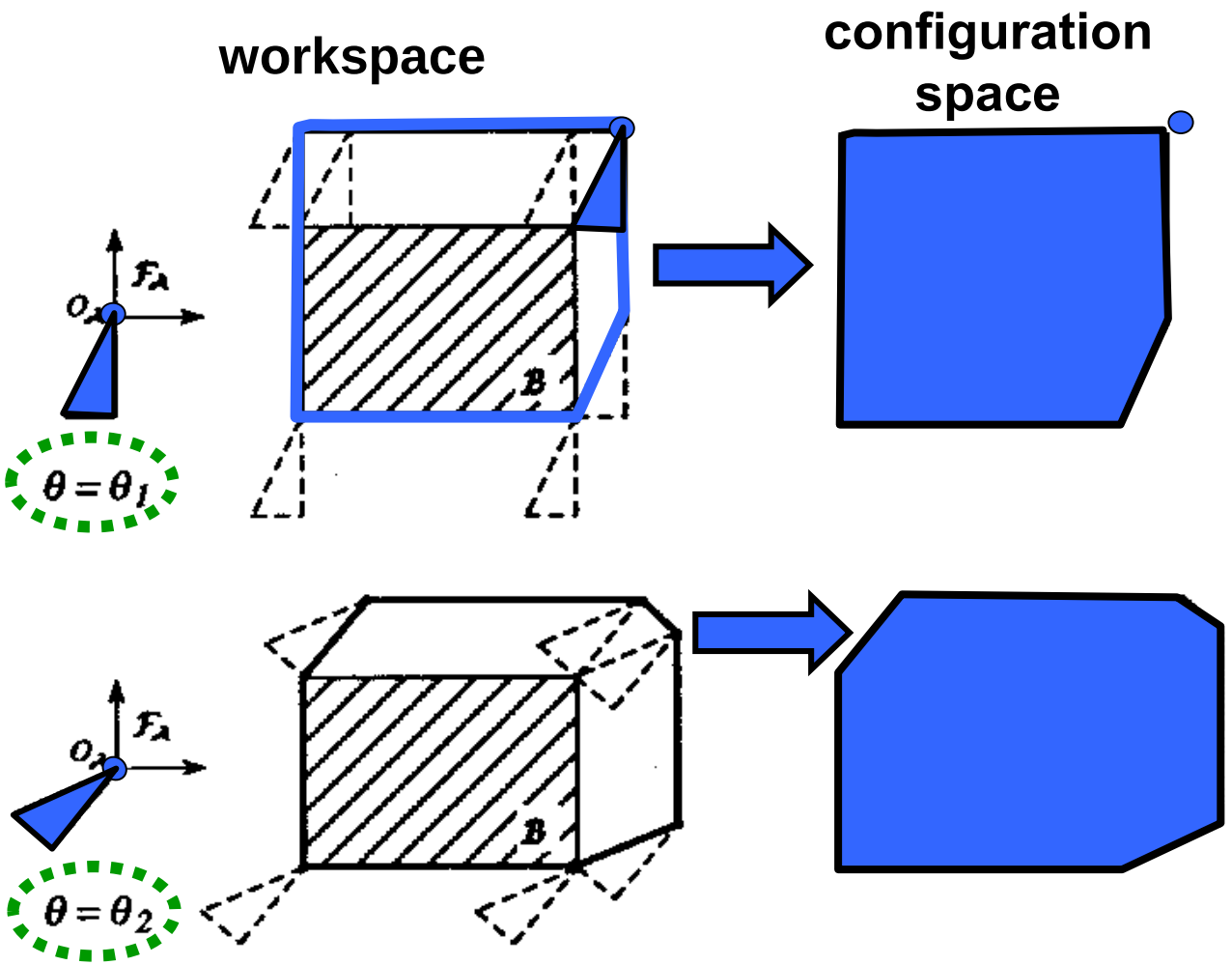
configuration space



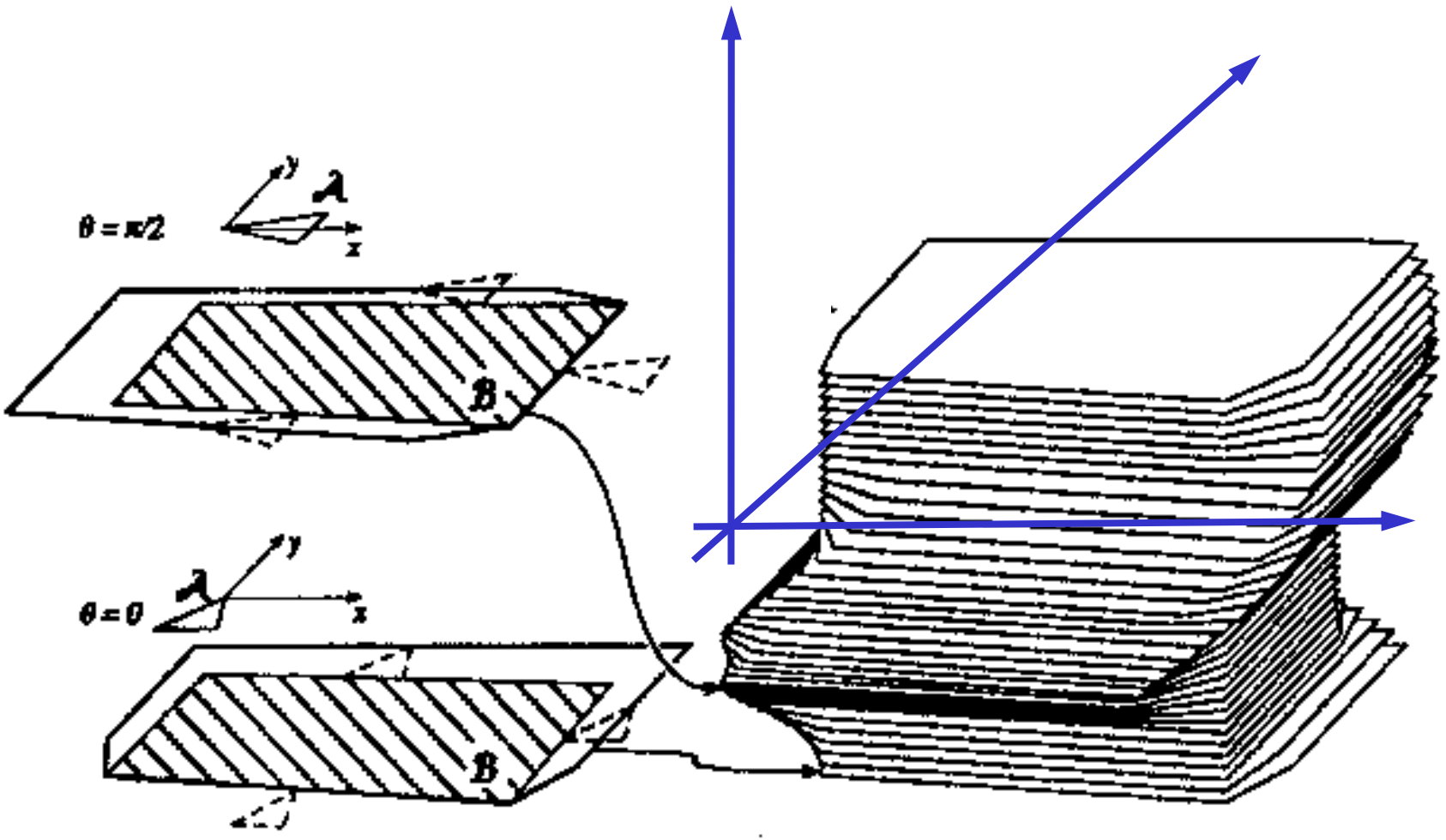
Polygonal robot translating in 2-D workspace



Polygonal robot translating & rotating in 2-D workspace



Polygonal robot translating & rotating in 2-D workspace



Free space topology

- A **free** path lies entirely in the free space F.
- The moving object and the obstacles are modeled as closed subsets, meaning that they contain their boundaries.
- One can show that the C-obstacles are closed subsets of the configuration space C as well.
- Consequently, the free space F is an open subset of C. Hence, each free configuration is the center of a ball of non-zero radius entirely contained in F.

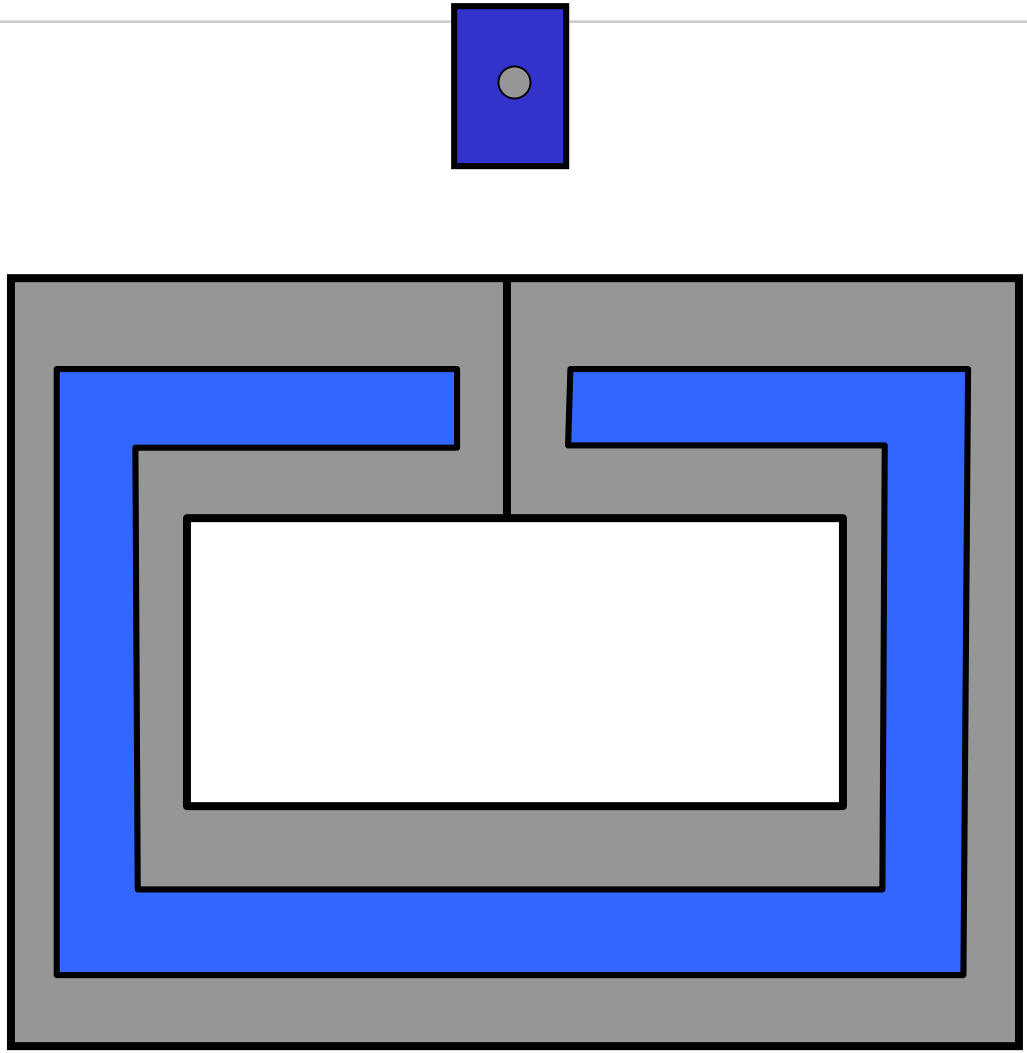
Semi-free space

A configuration q is semi-free if the moving object placed at q touches the boundary, but is not in the interior of obstacles:

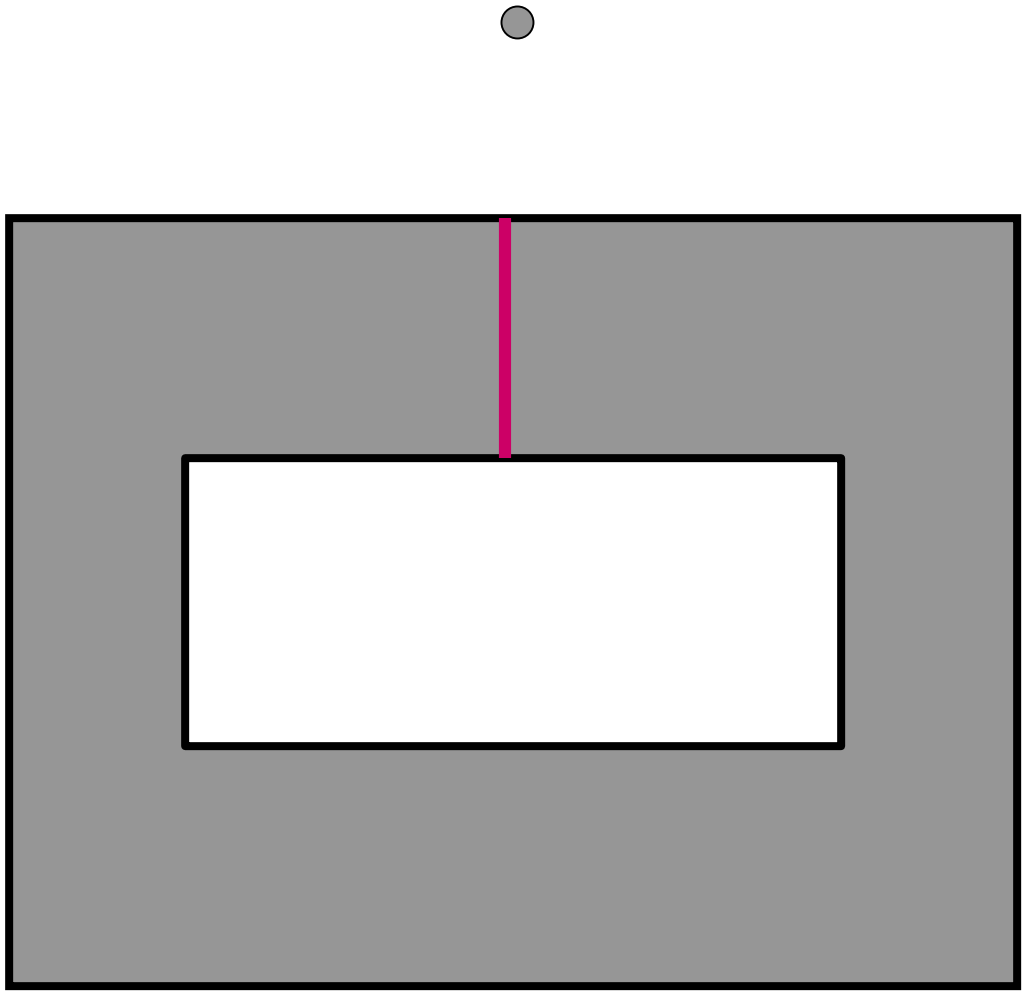
- Free, or
- in contact

The semi-free space is a closed subset of C. Its boundary is a superset of the boundary of F.

Example



Example



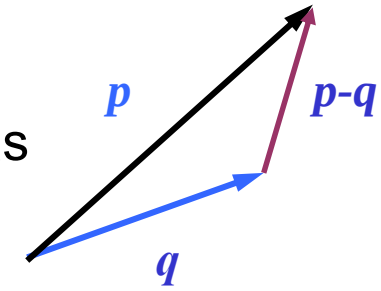
Minkowski sum

- The **Minkowski sum** of two sets P and Q, denoted by $P \oplus Q$, is defined as

$$P \oplus Q = \{ p+q \mid p \in P, q \in Q \}$$

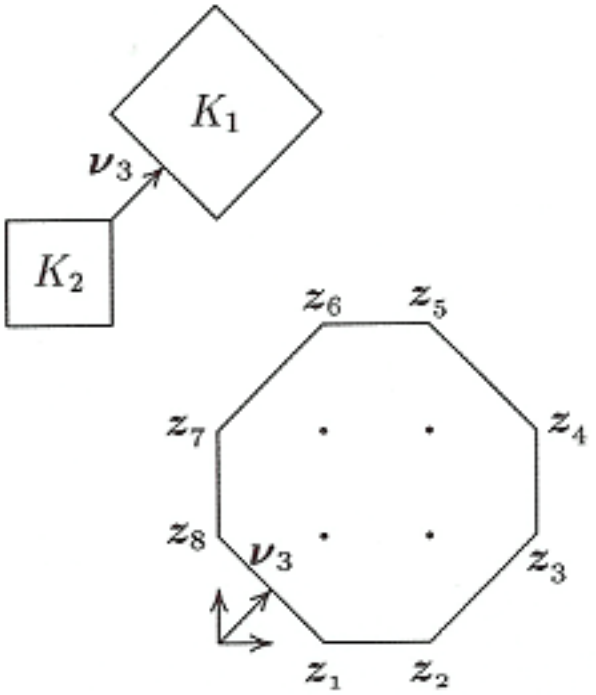
- Similarly, the **Minkowski difference** is defined as

$$P \ominus Q = \{ p-q \mid p \in P, q \in Q \}$$



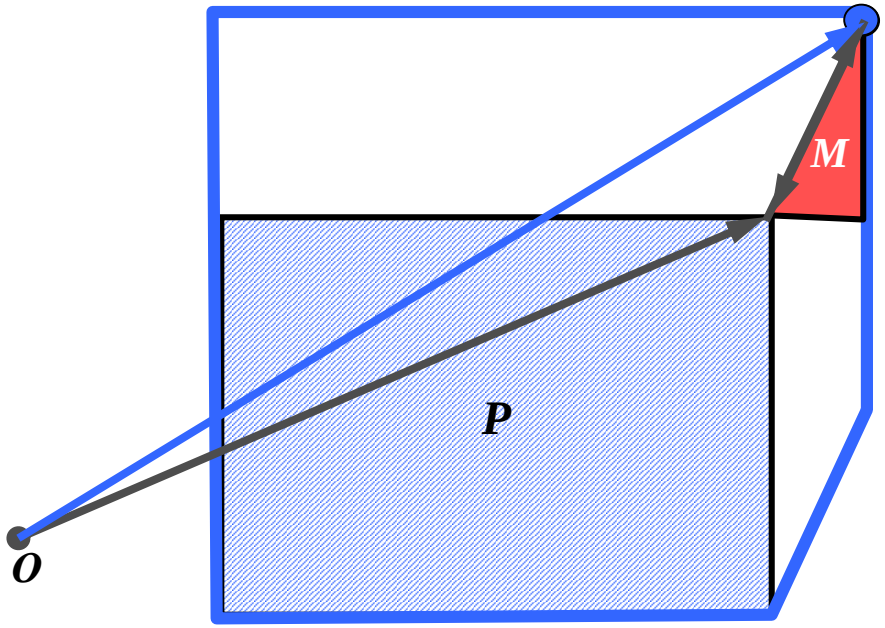
Minkowski sum of convex polygons

- The Minkowski sum of two convex polygons P and Q of m and n vertices respectively is a convex polygon $P \oplus Q$ of $m + n$ vertices.
  - The vertices of $P \oplus Q$ are the “sums” of vertices of P and Q.
- Example: Minkowski difference



Observation

- If P is an obstacle in the workspace and M is a moving object. Then the C-space obstacle corresponding to P is $P \ominus M(0)$.



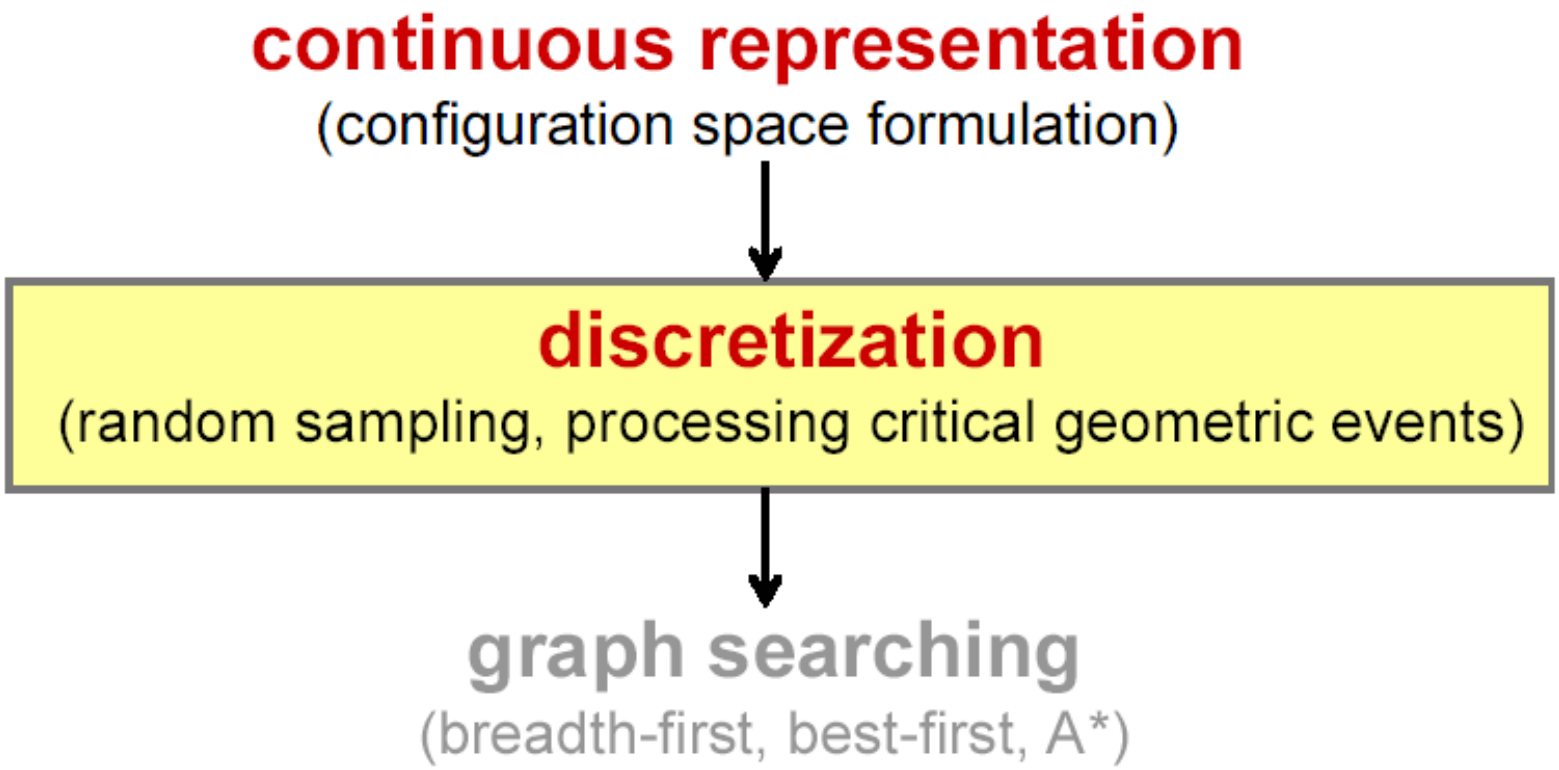
Planner Wish List

- Complete
- Handles narrow / tight passages.
- Bounded running time.
- Behavior well characterized.
- Low complexity
- Not too hard to understand
- Not too hard to implement
- Scalable

Motion Planning for Point Robots

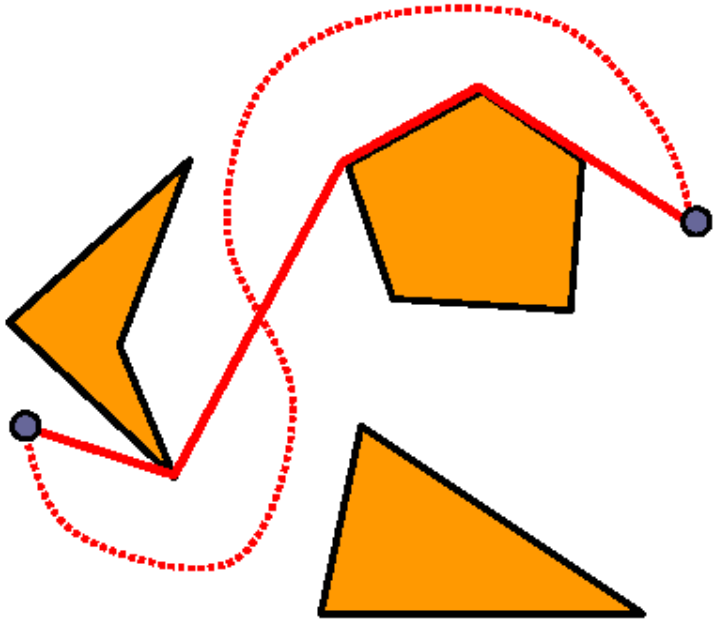
- Input
  - Robot represented as a **point** in the **plane**
  - Obstacles represented as polygons
  - Initial and goal positions
- Output
  - A collision-free path between the initial and goal positions

Framework


continuous representation
(configuration space formulation)
discretization
(random sampling, processing critical geometric events)
graph searching
(breadth-first, best-first, A*)

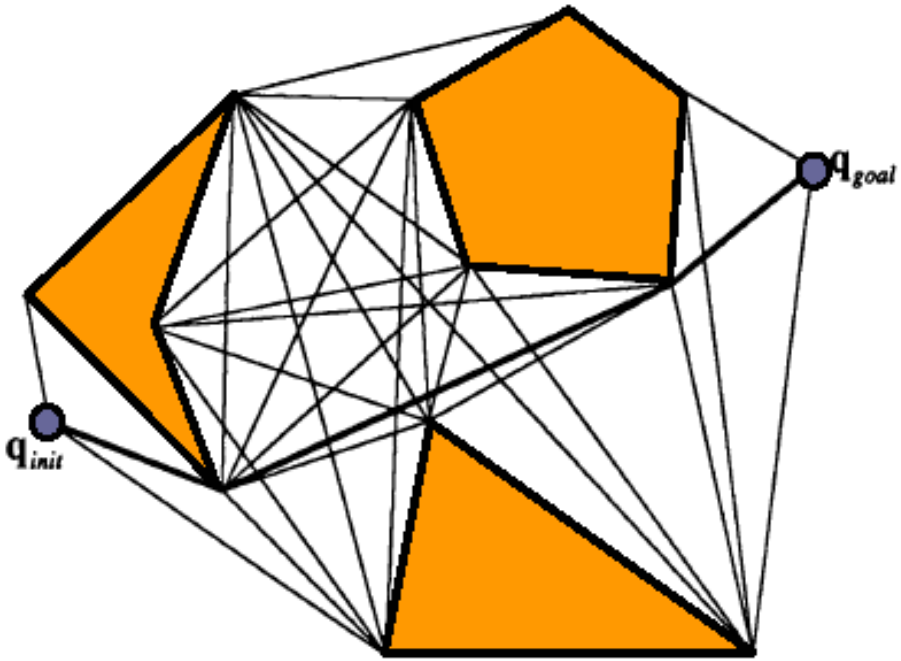
Visibility Graph Method

- **Observation**:
  - If there is a collision-free path between two points, then there is a polygonal path that bends only at the obstacles vertices.
- **Why ?**
  - Any collision-free path can be transformed into a polygonal path that bends only at the obstacle vertices.
- A **polygonal path** is a piecewise linear curve.



What is a Visibility Graph ?

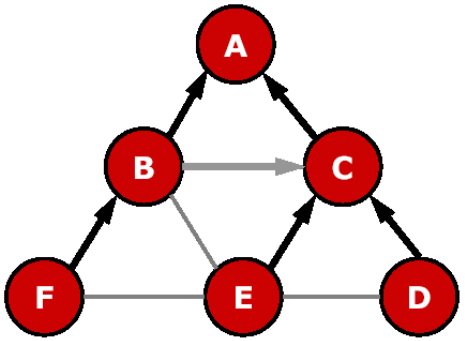
- A **visibility graph** is a graph such that
  - Nodes: *q*_init, *q*_goal, or an obstacle vertex.
  - Edges: An edge exists between nodes *u* and *v* if the line segment between *u* and *v* is an obstacle edge or it does not intersect the obstacles.



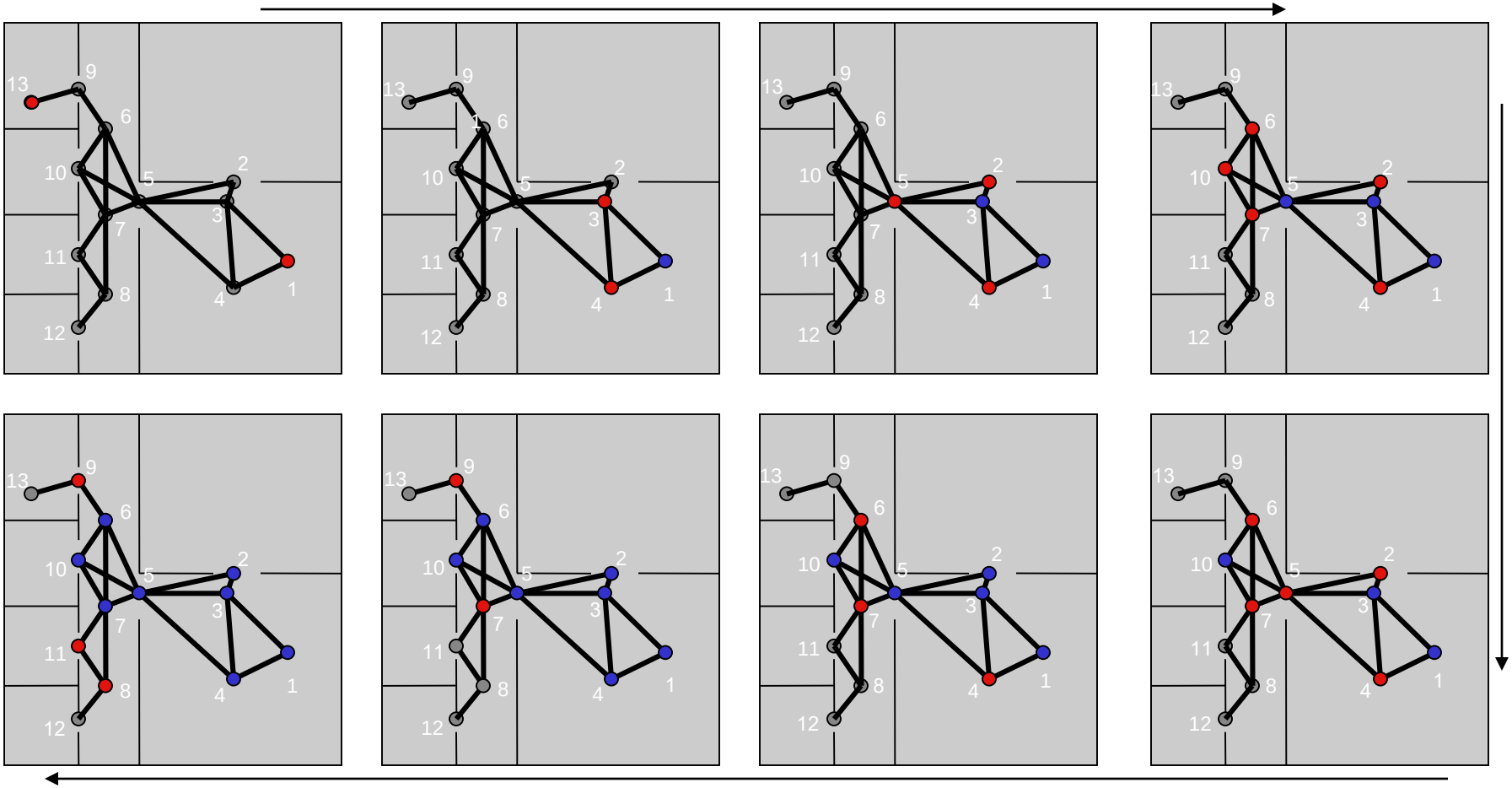
Breadth First Search

- **Input**:
  - q_init, q_goal, visibility graph G
- **Output**:
  - A path between q_init and q_goal

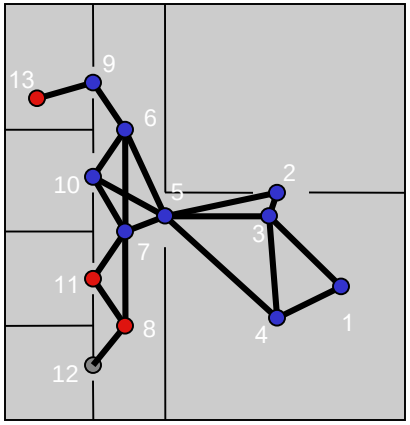
```
1: Q = new queue;
2: Q.enqueue(q_init);
3: mark q_init as visited;
4: while Q is not empty
5:    curr = Q.dequeue();
6:    if curr == q_goal then
7:      return curr;
8:    for each w adjacent to curr
10:      if w is not visited
11:        w.parent = curr;
12:        Q.enqueue(w)
13:        mark w as visited
```

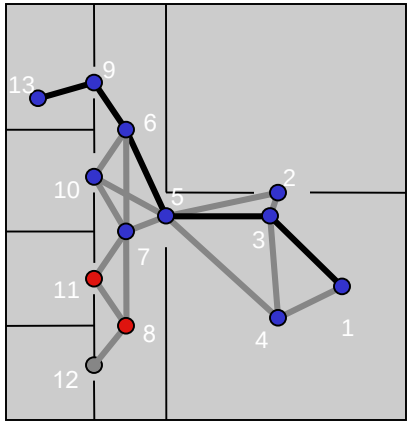


A* Search (Straight-Line-Distance Heuristic)



A* Search (Straight-Line-Distance Heuristic)





Note that A* expands fewer nodes than breadth-first, but more than greedy

It's the price you pay for optimality

Keys are:

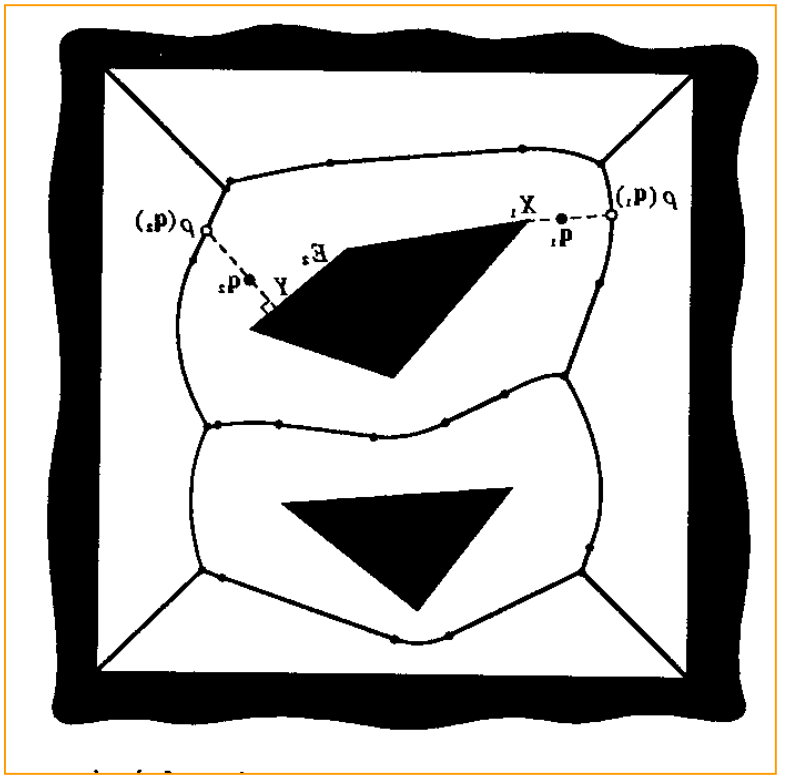
- Data structure for a node
- Priority queue for sorting open nodes
- Underlying graph structure for finding neighbours

Classic Motion Planning Approaches

- **Roadmap**
  - Represent the connectivity of the free space by a network of 1-D curves
- **Cell decomposition**
  - Decompose the free space into simple cells and represent the connectivity of the free space by the adjacency graph of these cells
- **Potential field**
  - Define a potential function over the free space that has a global minimum at the goal and follow the steepest descent of the potential function

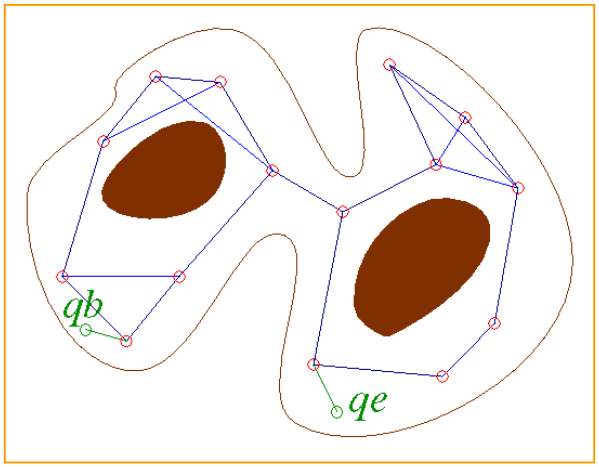
Motion Planning as a Computational Problem

- Goal: Characterize the connectivity of a space (e.g., the collision-free subset of configuration space)
- High computational complexity: Requires time exponential in number of degrees of freedom, or number of moving obstacles, or etc…
- Two main algorithmic approaches:
  - Planning by random sampling
  - **Planning by extracting criticalities**



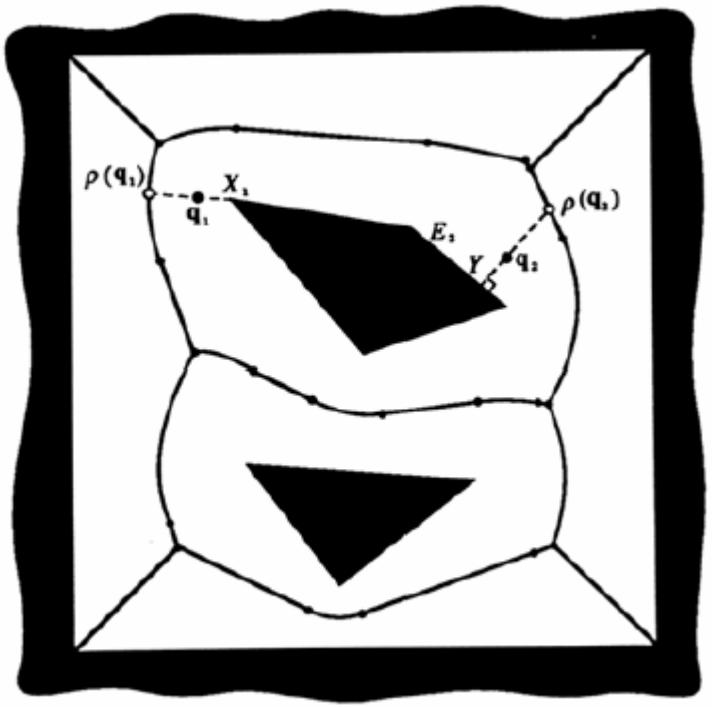
Motion Planning as a Computational Problem

- Goal: Characterize the connectivity of a space (e.g., the collision-free subset of configuration space)
- High computational complexity: Requires time exponential in number of degrees of freedom, or number of moving obstacles, or etc…
- Two main algorithmic approaches:
  - **Planning by random sampling**
  - Planning by extracting criticalities



Roadmap

- Visibility Graph
  - Shakey Project, SRI [Nilsson, 1969]
- Voronoi diagram
  - Introduced by computational geometry researchers. Generate paths that maximizes clearance.
  - Applicable mostly to 2-D configuration spaces.
  - Space O(n)
  - Running time O(n log n)

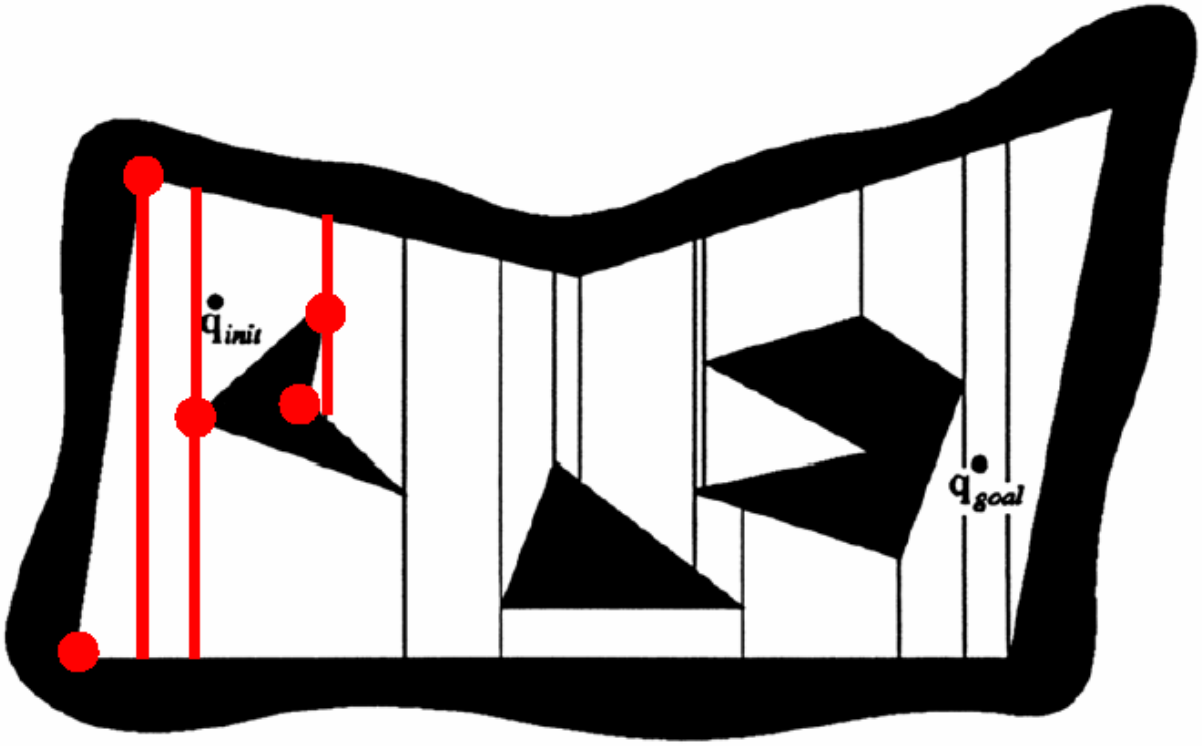


Classic Motion Planning Approaches

- **Roadmap**
  - Represent the connectivity of the free space by a
  - network of 1-D curves
- **Cell decomposition**
  - Decompose the free space into simple cells and represent the connectivity of the free space by the adjacency graph of these cells
- **Potential field**
  - Define a potential function over the free space that has a global minimum at the goal and follow the steepest descent of the potential function

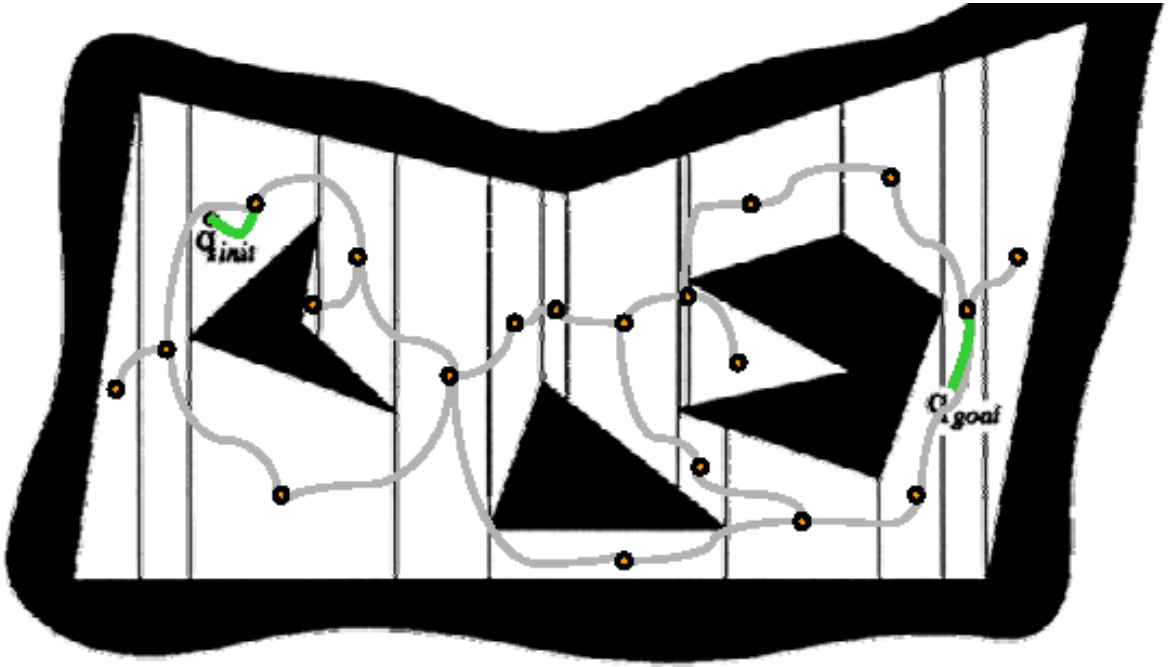
Cell Decomposition Methods

- **Exact cell decomposition**
  - The free space *F* is represented by a collection of nonoverlapping simple cells whose union **is exactly** *F*
- Examples of cells: trapezoids, triangles
- Example: trapezoidal decomposition
- Sweep Line Algorithm



Computational Efficiency

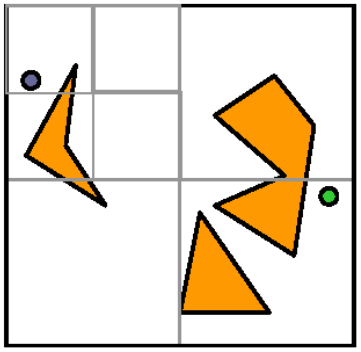
- Running time $O(n \log n)$ by planar sweep
- Space $O(n)$
- Mostly for 2-D configuration spaces
- Adjacency graph
  - **Nodes**: cells
  - **Edges**: There is an edge between every pair of nodes whose corresponding cells are adjacent.

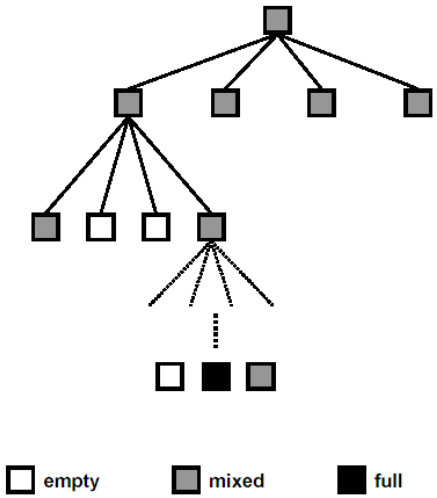


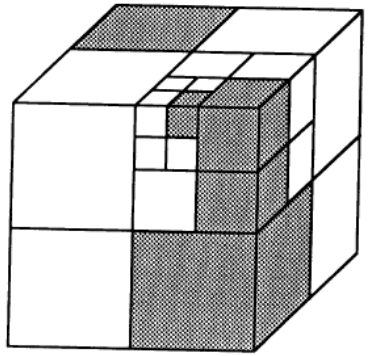
Cell Decomposition Methods

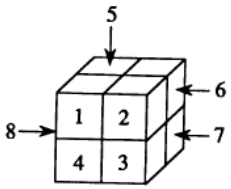
- Exact cell decomposition
- **Approximate cell decomposition**
  - Freespace *F* is represented by a collection of non-overlapping cells whose union is **contained** in *F.*
  - Cells usually have simple, regular shapes, *e.g.,* rectangles, squares.
  - Facilitate hierarchical space decomposition
  - 2D Quadtree

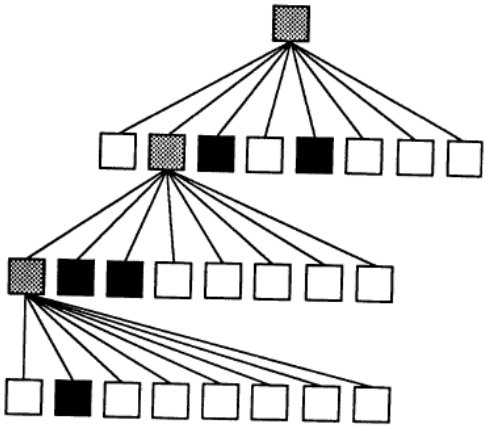
3D Octree

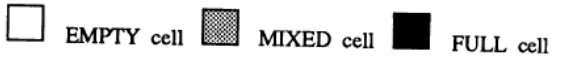






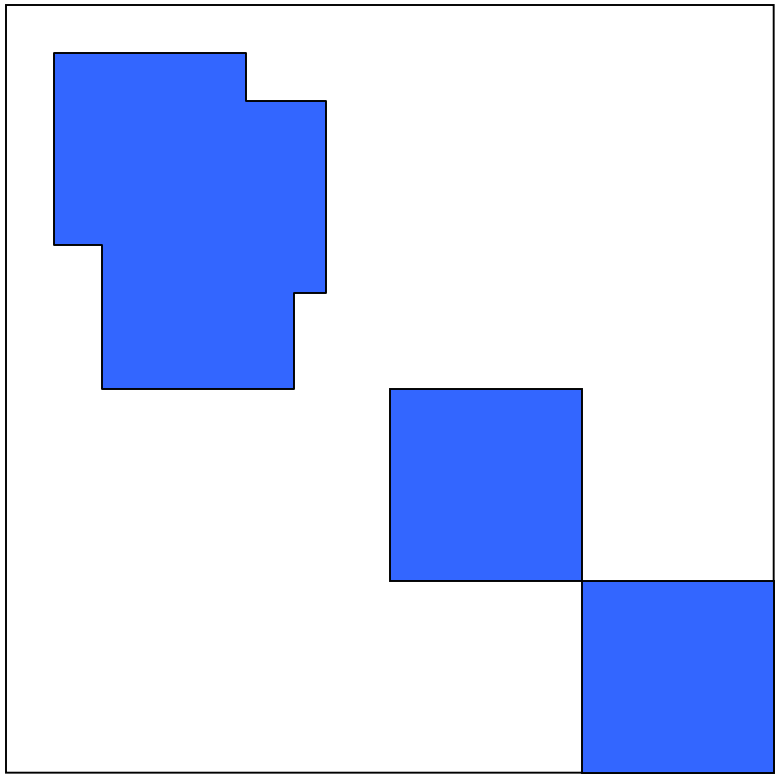






Quadtree Example

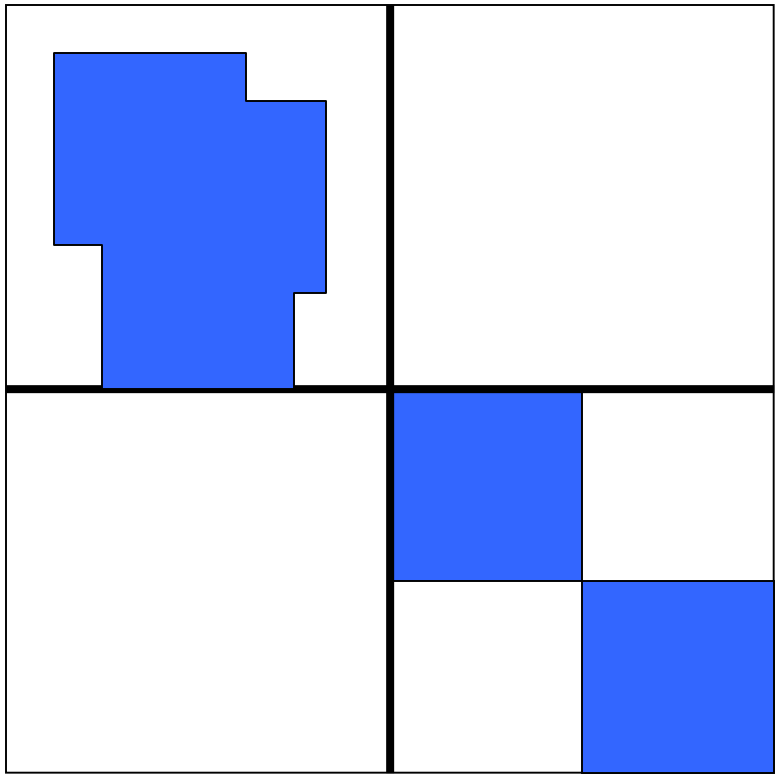
Space Representation



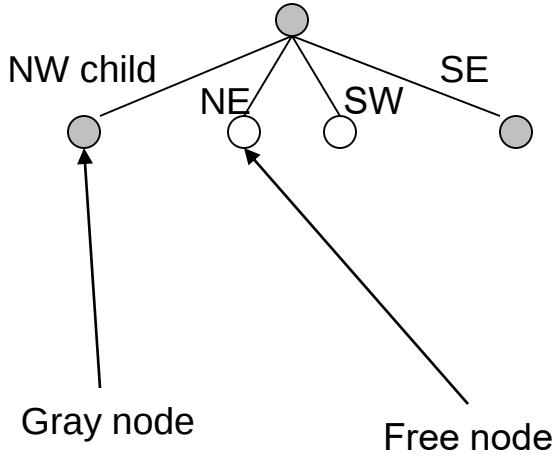
Equivalent quadtree

Quadtree Example

Space Representation

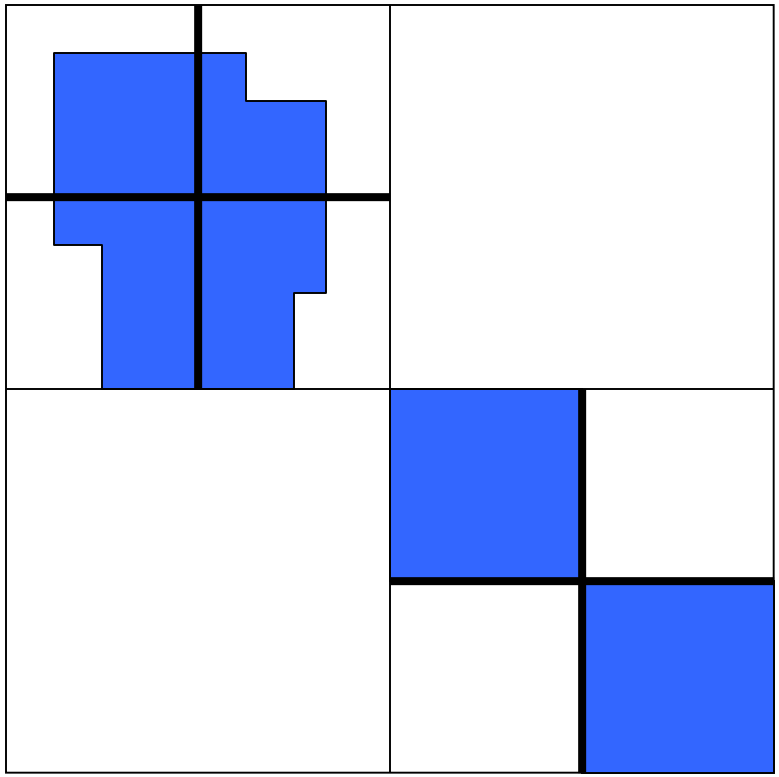


Equivalent quadtree

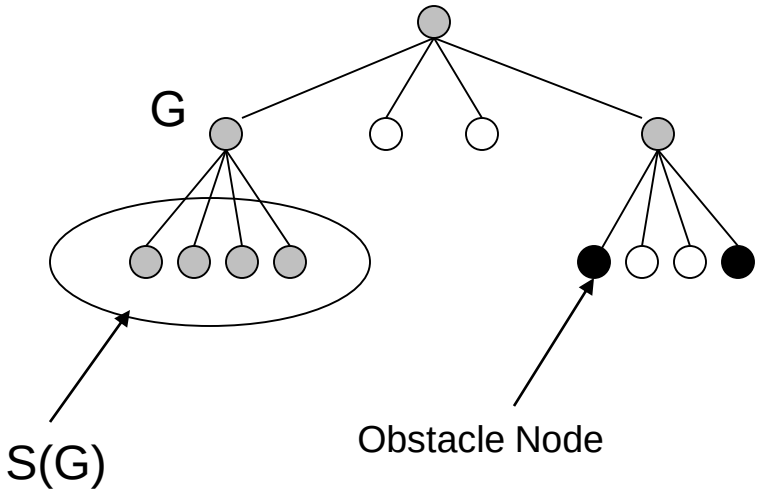


Quadtree Example

Space Representation

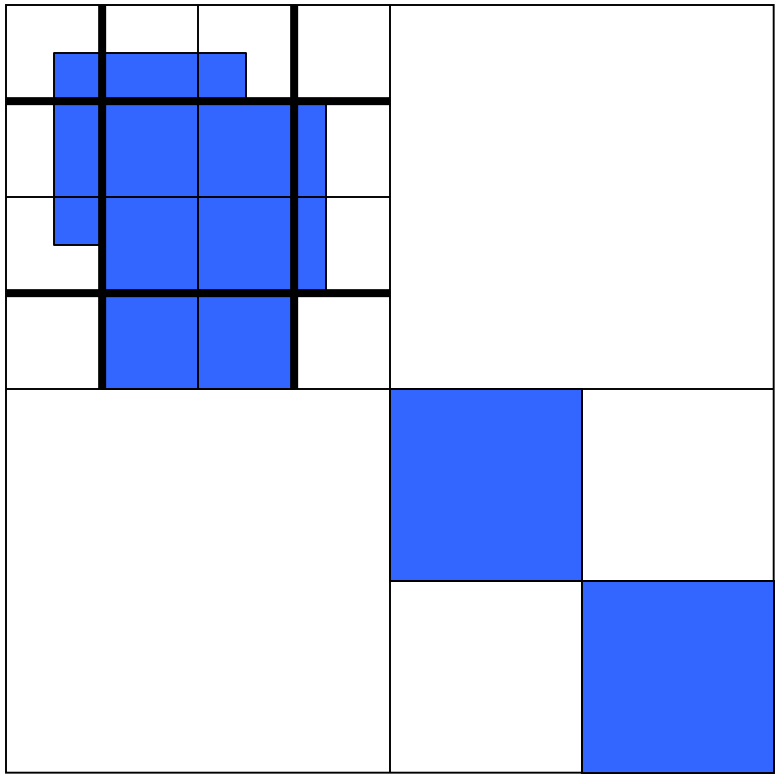


Equivalent quadtree

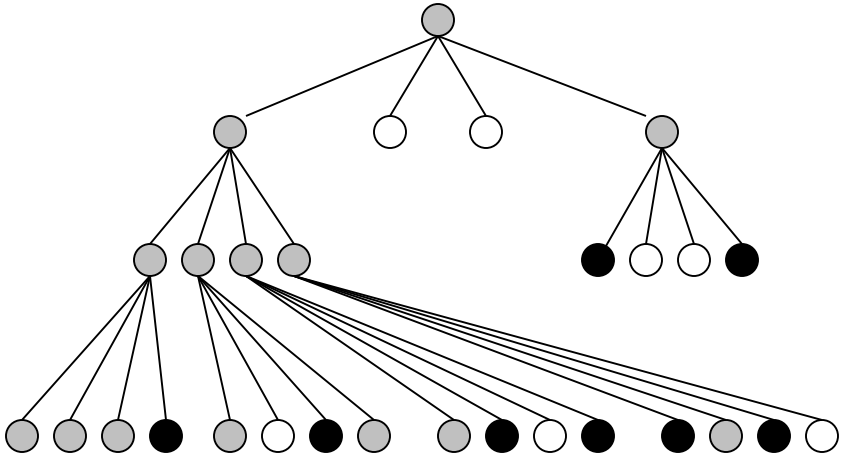


Quadtree Example

Space Representation



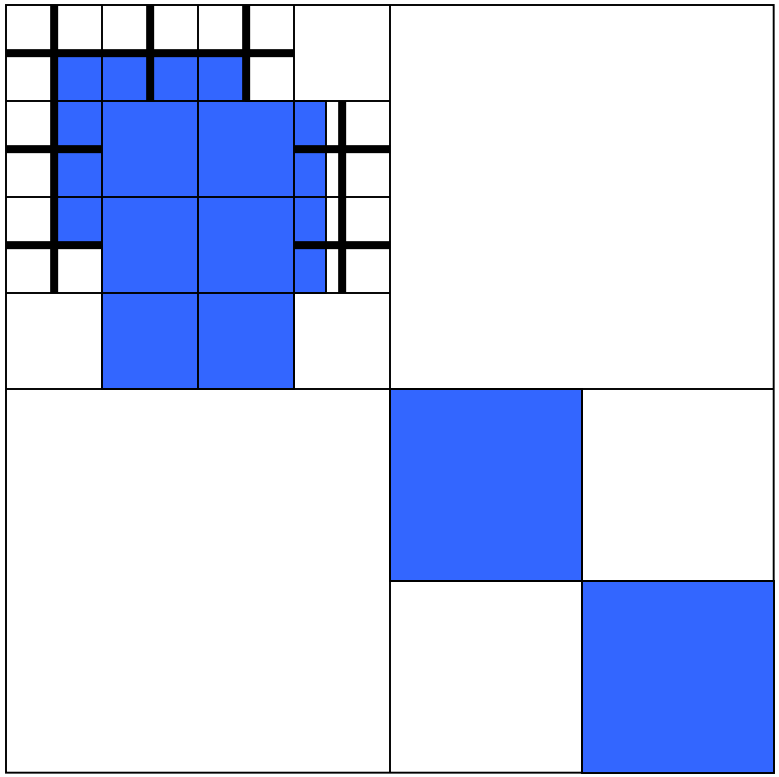
Equivalent quadtree



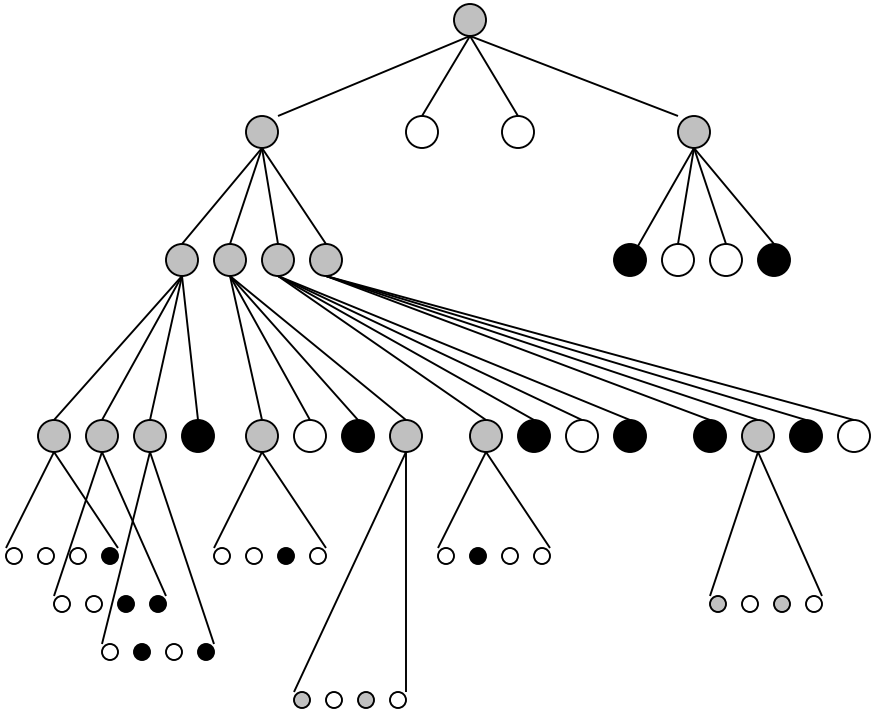
Each of these steps are examples of pruned quadtrees, or the space at different resolutions

Quadtree Example

Space Representation

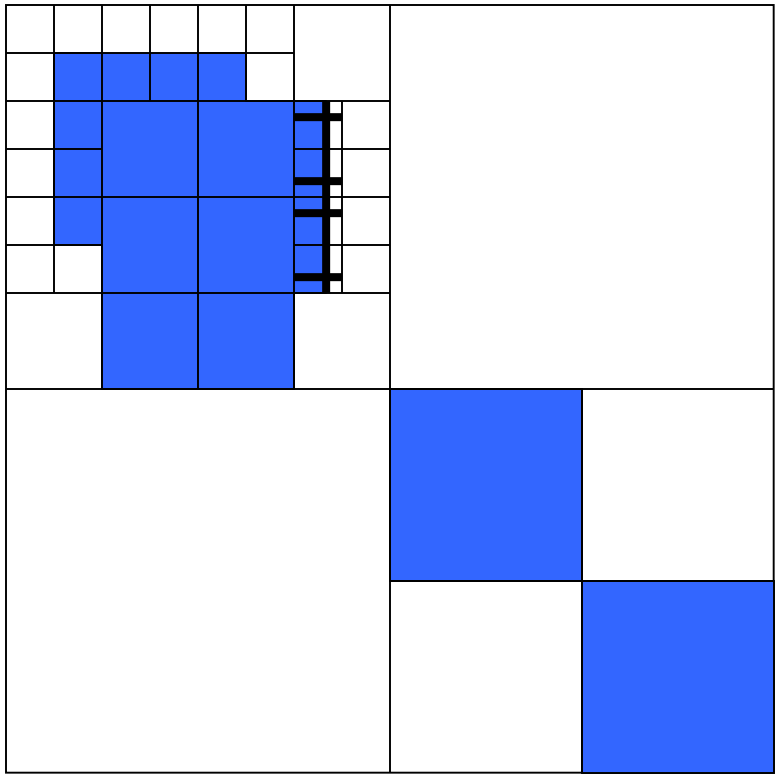


Equivalent quadtree

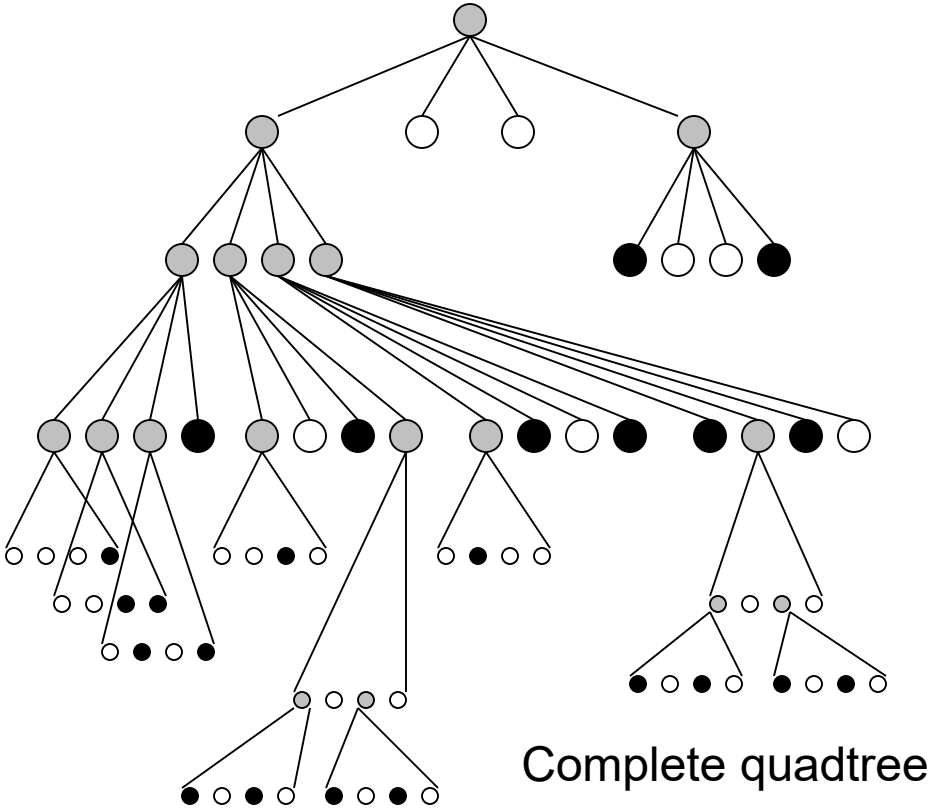


Quadtree Example

Space Representation



Equivalent quadtree



Complete quadtree

Classic Motion Planning Approaches

- **Roadmap**
  - Represent the connectivity of the free space by a network of 1-D curves
- **Cell decomposition**
  - Decompose the free space into simple cells and represent the connectivity of the free space by the adjacency graph of these cells
- **Potential field**
  - Define a potential function over the free space that has a global minimum at the goal and follow the steepest descent of the potential function

Potential Fields

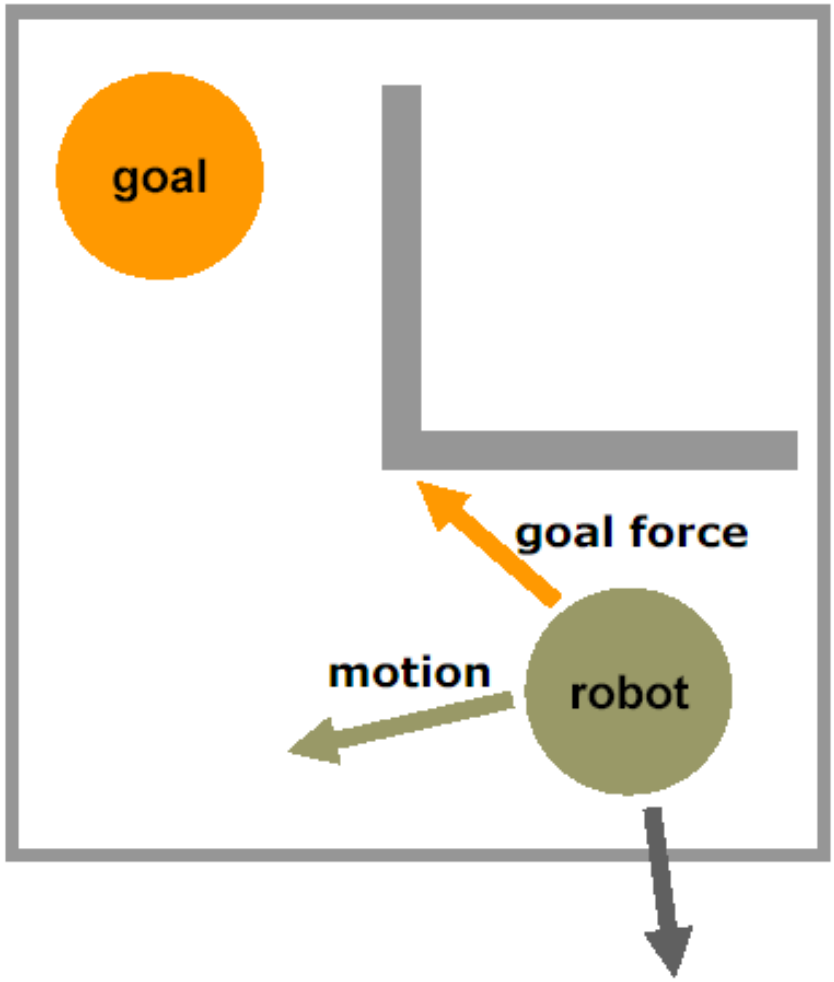
- Initially proposed for real-time collision avoidance [Khatib, 1986]. Hundreds of papers published on this topic.
- A potential field is a scalar function over the free space.
- To navigate, the robot applies a force proportional to the negated gradient of the potential field.
- A **navigation function** is an ideal potential field that
  - has global minimum at the goal
  - has no local minima (hard to obtain)
  - grows to infinity near obstacles
  - is smooth

Attractive and repulsive fields

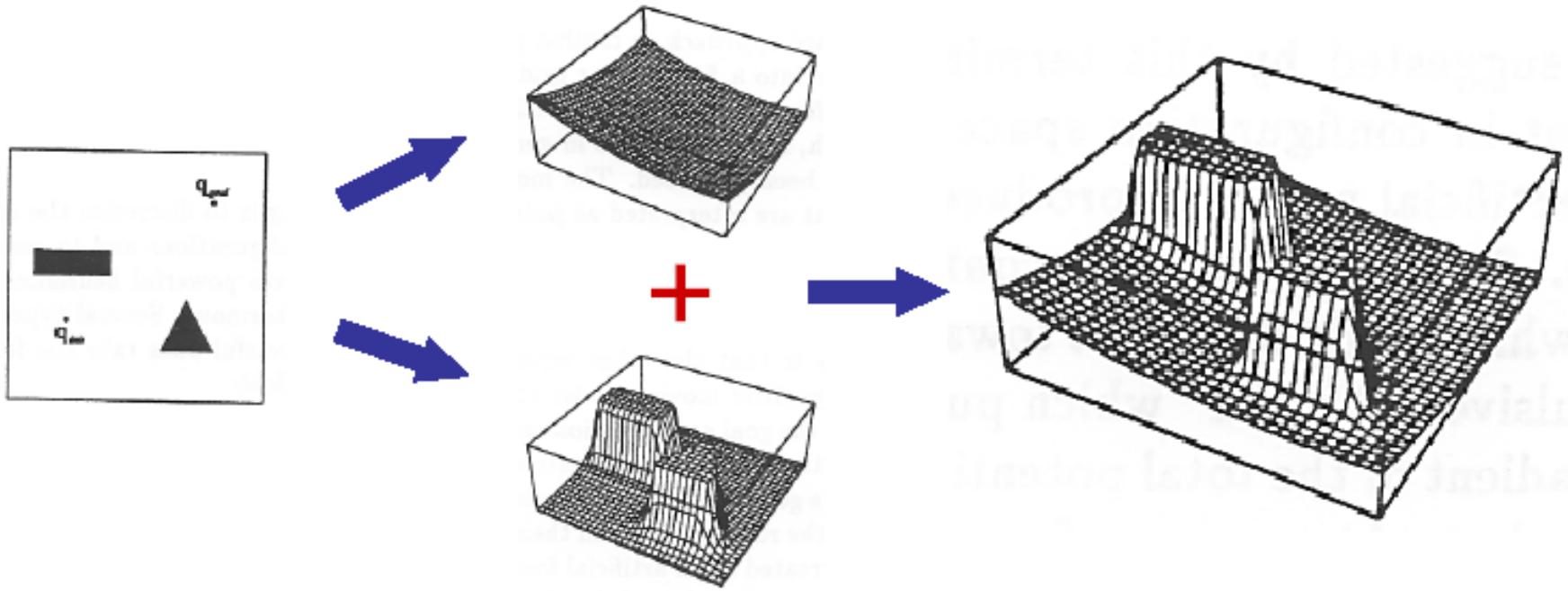
$$F_{att} = -k_{att}(x - x_{goal})$$

$$F_{rep} = \begin{cases} k_{rep}\left(\frac{1}{d} - \frac{1}{d_0}\right)\frac{1}{d^2}\frac{\partial d}{\partial x}, & d \geq d_0 \\ 0, & d > d_0 \end{cases}$$

k_{att}, k_{rep}: positive *scaling* factors
x: positon of the robot
d: distance to obstacle
d_0: distance of influence

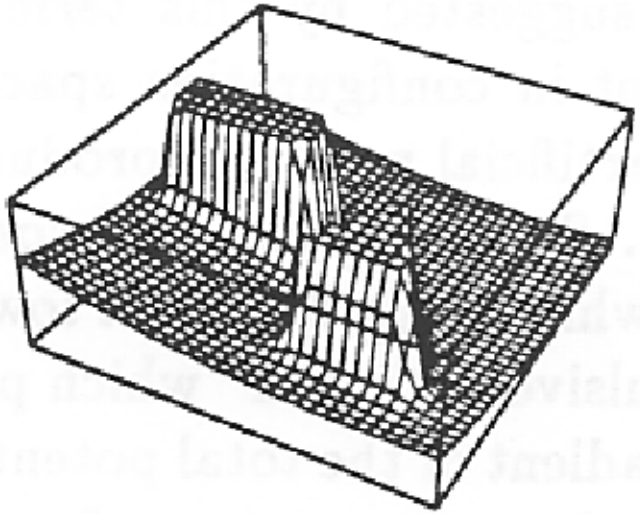


Algorithm in pictures

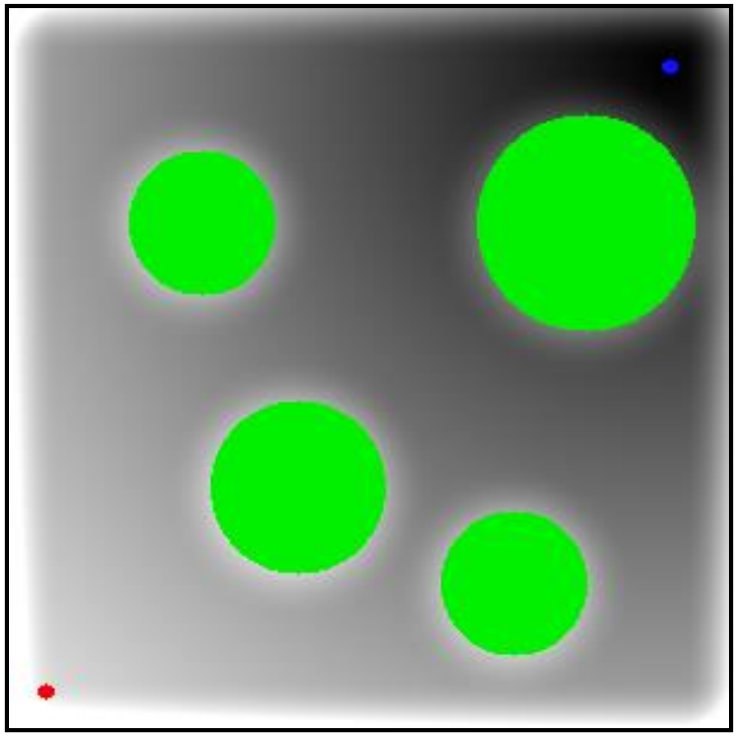


Motion Planning for Point Robots

- Place a regular grid *G* over the configuration space
- Compute the potential field over *G*
- Search *G* using a best-first algorithm with potential field as the heuristic function
- Local minima: What can we do?
  - Escape from local minima by taking random walks
  - Build an ideal potential field – navigation function – that does not have local minima
  - Can such an ideal potential field be constructed efficiently in general?



Potential Field

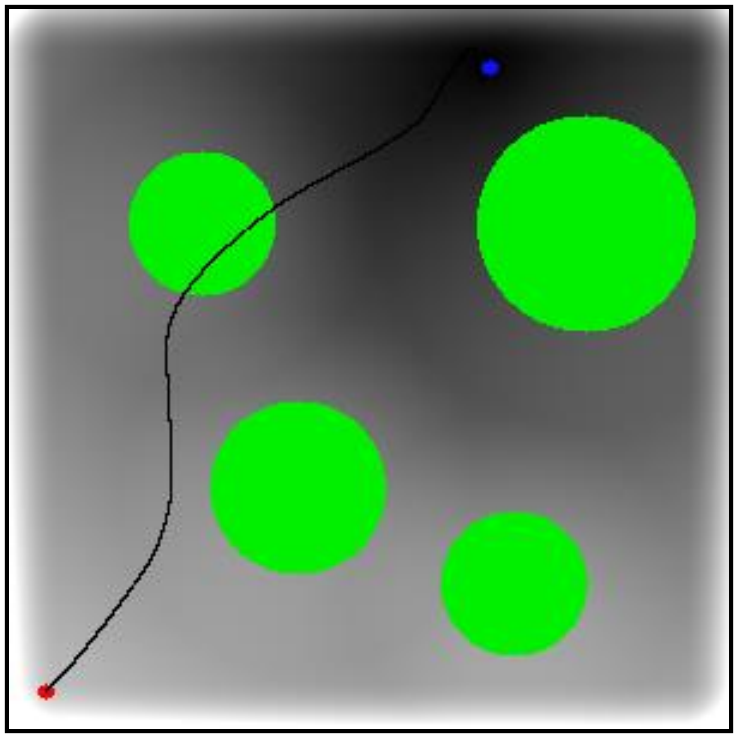


- Red is start point, blue is goal
- This used a quadratic field strength around the obstacles
- Note that the boundaries of the world also contribute to the field

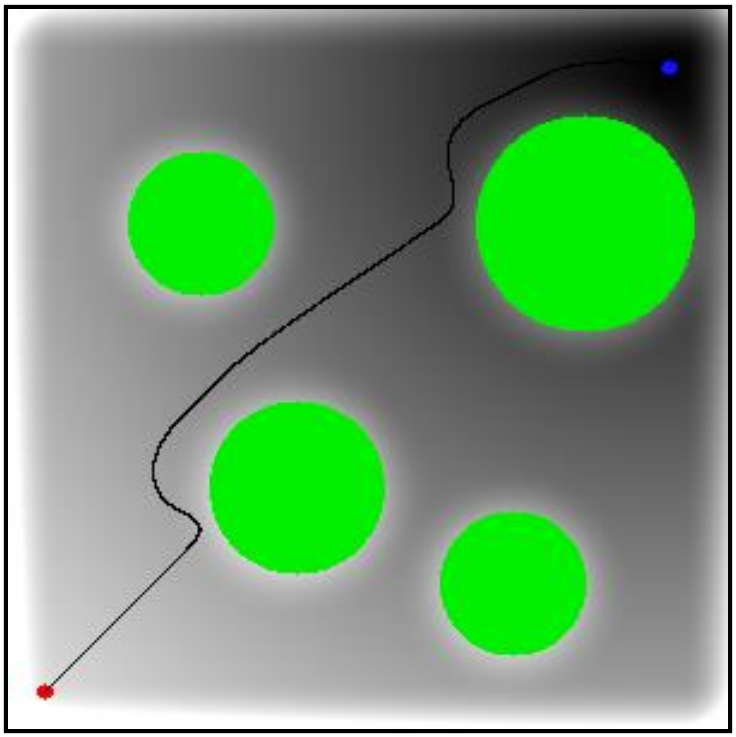
Potential Field Problems

- There are many parameters to tune
  - Strength of the field around each obstacle
  - Function for field strength around obstacle
  - Steepness of force toward the goal
- Goals conflict
  - High field strength avoids collisions, but produces big forces and hence unnatural motion
- Local minima cause huge problems

Problems

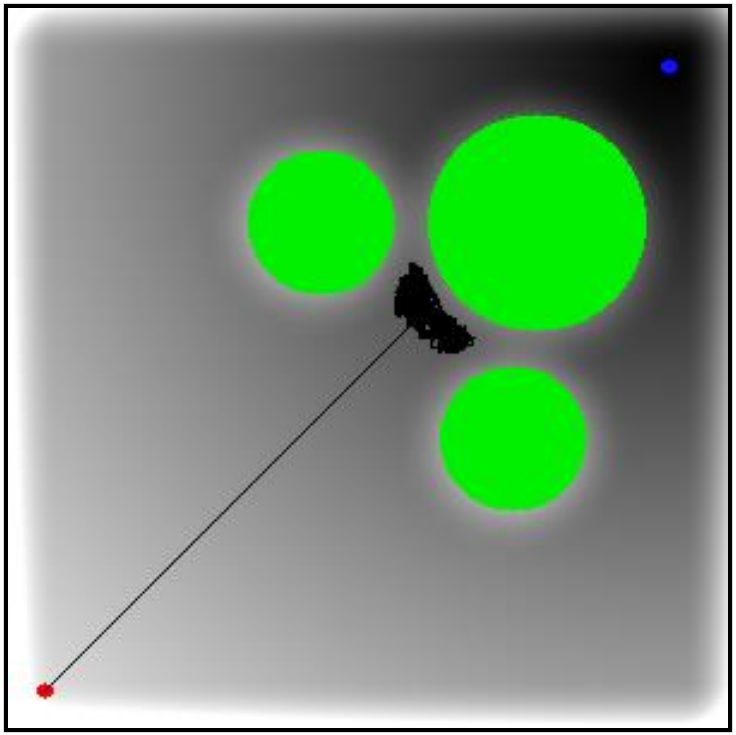


Field too weak



Field too strong

Local Minima Example



The Local Minima Problem

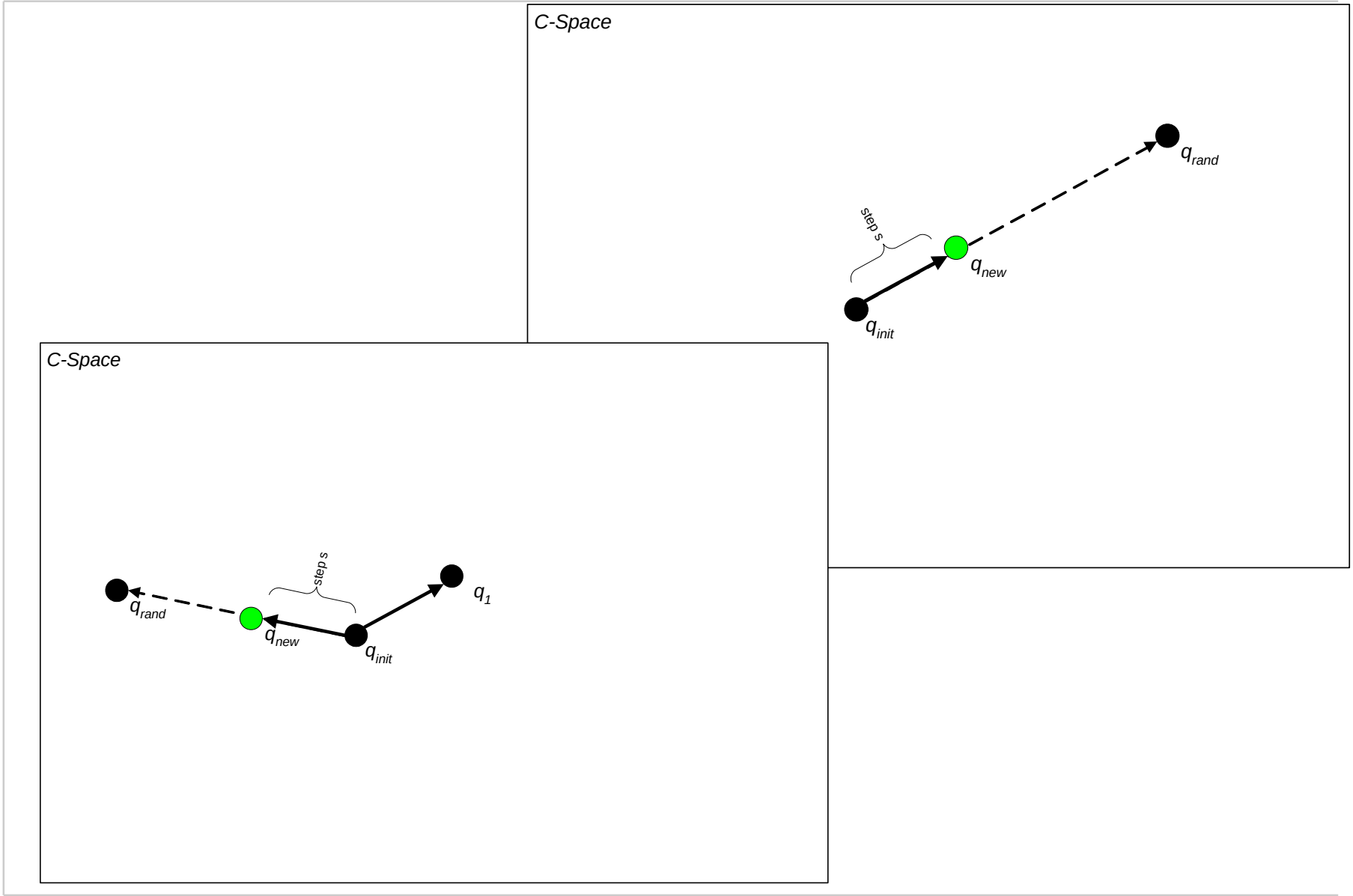
- Recall, path planning can be viewed as optimization
- Potential field planning is gradient descent optimization
- The biggest problem with gradient descent is that it gets stuck in local minima
- Potential field planning suffers from exactly the same problem
- Must have a way to work around this
  - Go back if a minima is found, and try another path
- With what sorts of environments will potential fields work best?

Rapidly Exploring Random Tree (RRT) Algorithm

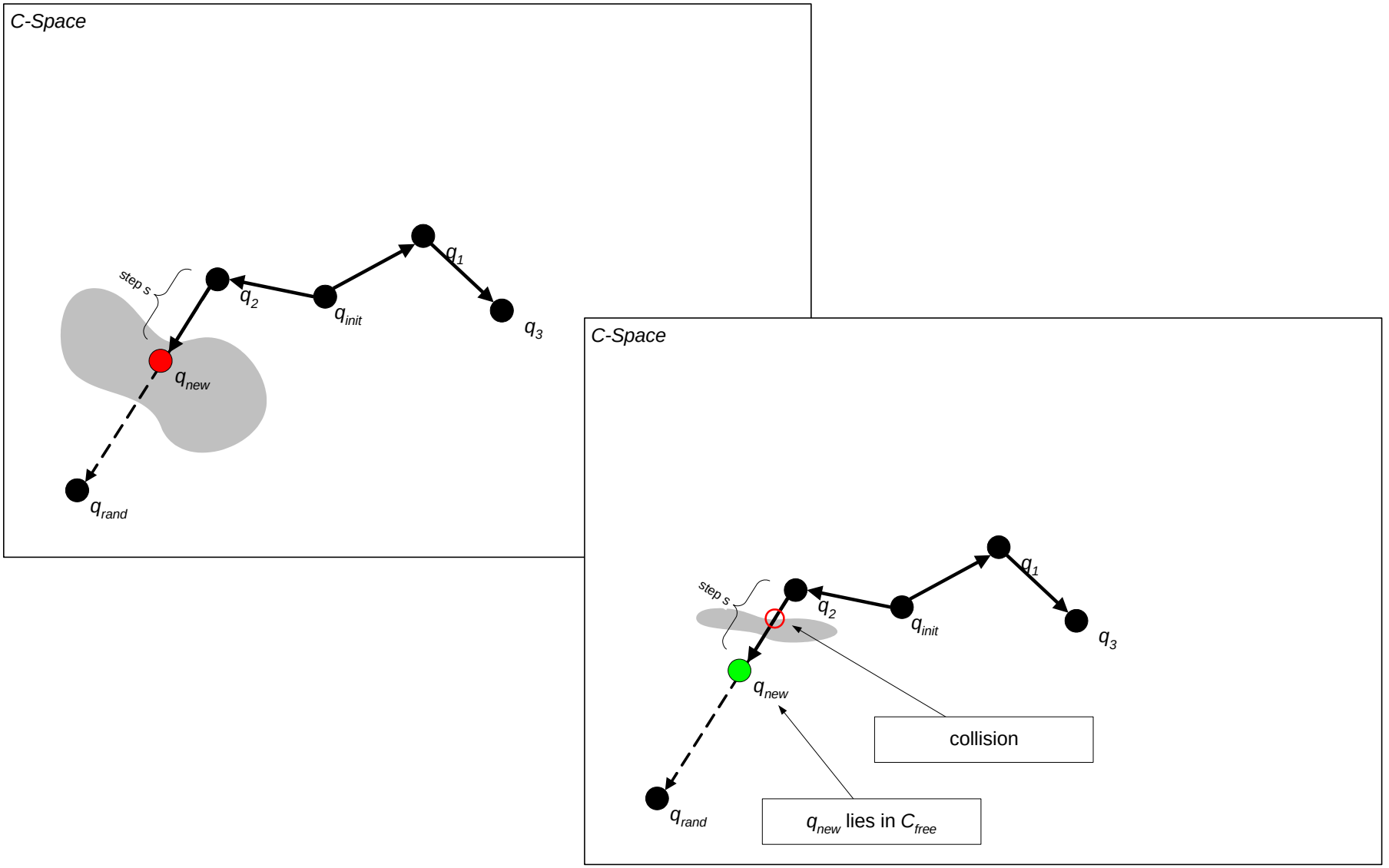
- **Init:** Add *q_init* to the RRT
- **1.** Generate a random position *q_rand* in *C-free*
- **2.** Find the node *q_near* in the so far built-ob RRT which is the nearest to the random position *q_rand*
- **3.** Check if you can reach a node *q_new* starting from *q_near* in direction to *q_rand* in a fixed distance *s* without collision.
  If *q_new* can be reached without collision add the new node with an edge from *q_near* to *q_new* to the RRT.
- **4.** Check if *q_new* is near enough to *q_goal* to reach it.
  If the distance is too big, continue with step 1.
- **5.** If the distance is below a certain value try to connect *q_new* with *q_goal*. If there is a collision on the way from *q_new* to *q_goal*, continue with step 1.
- **6.** If *q_goal* could be reached without collision add it to the RRT and connect it with *q_new*.

Now the path from *q_init* to *q_goal* can be determined by going backwards starting from *q_goal*.

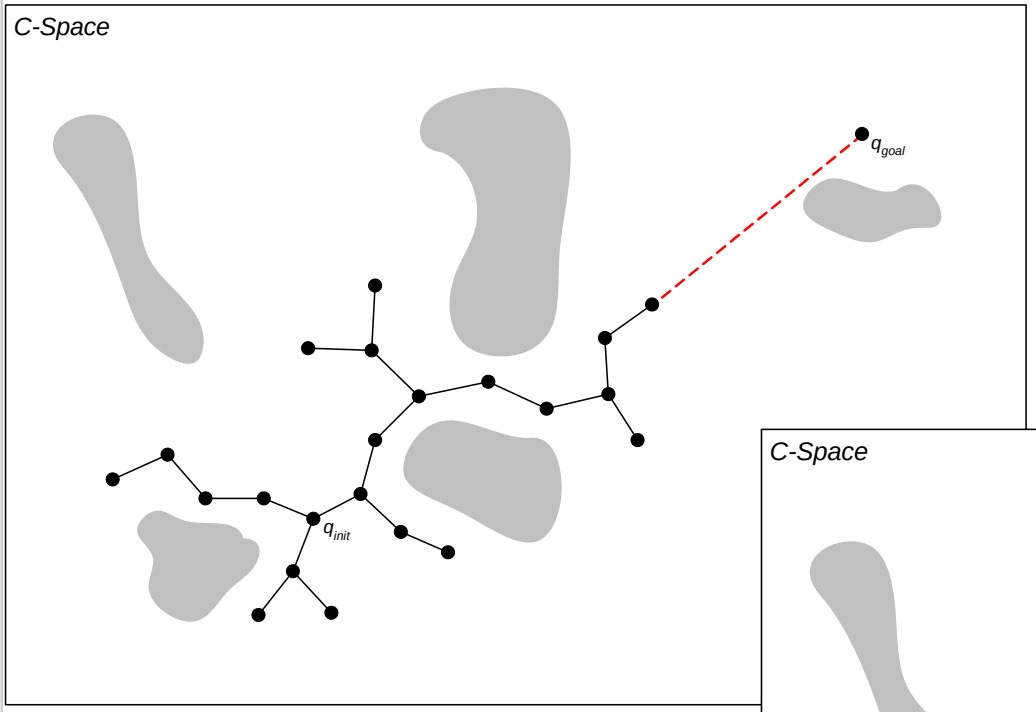
RRT Algorithm

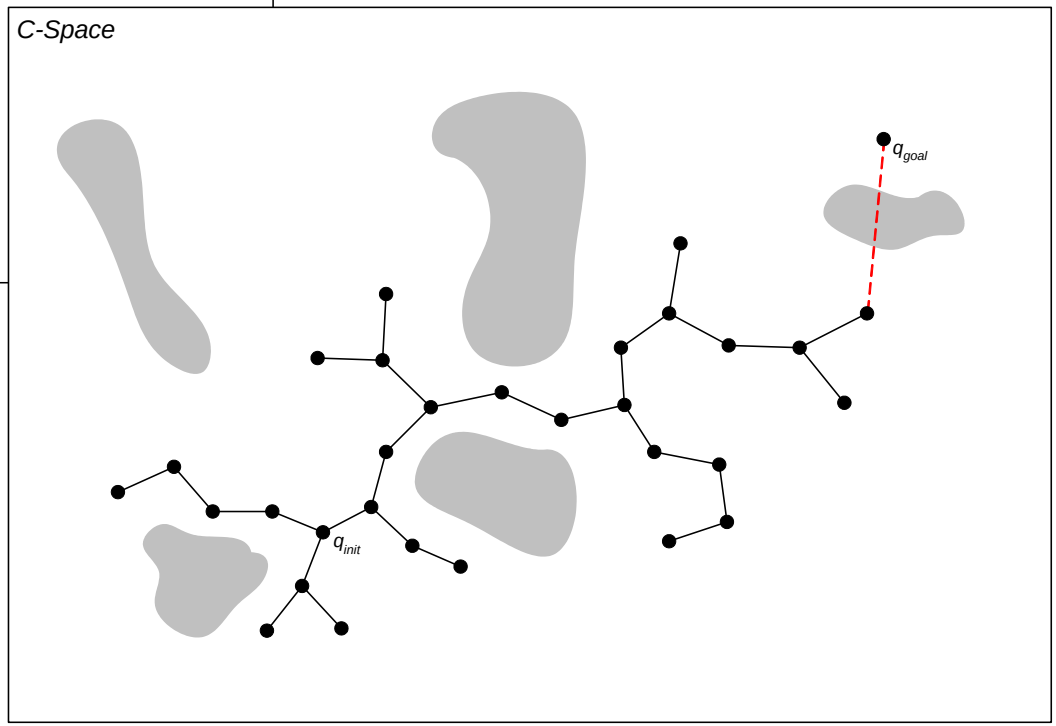


RRT Algorithm

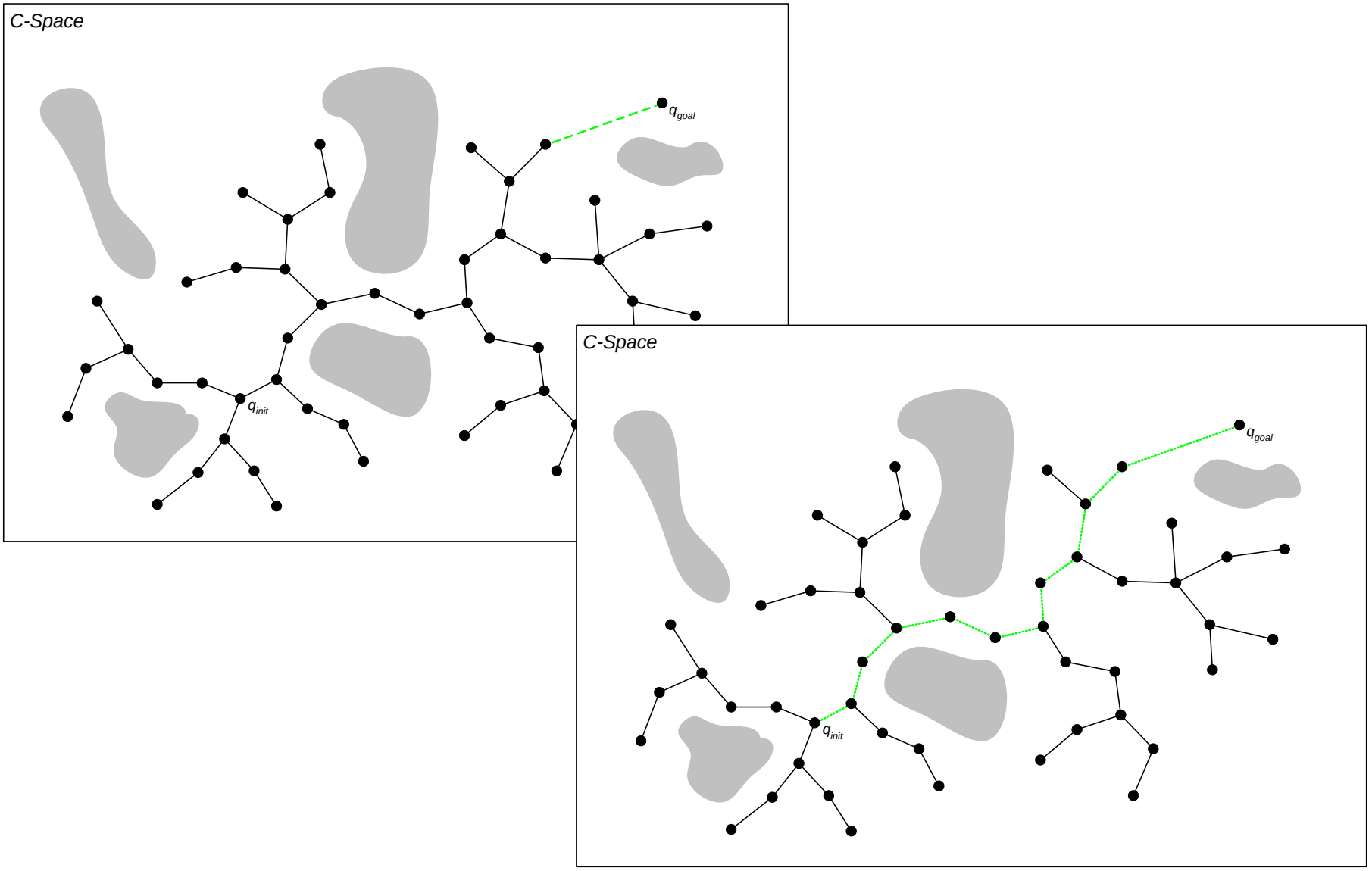


RRT Algorithm



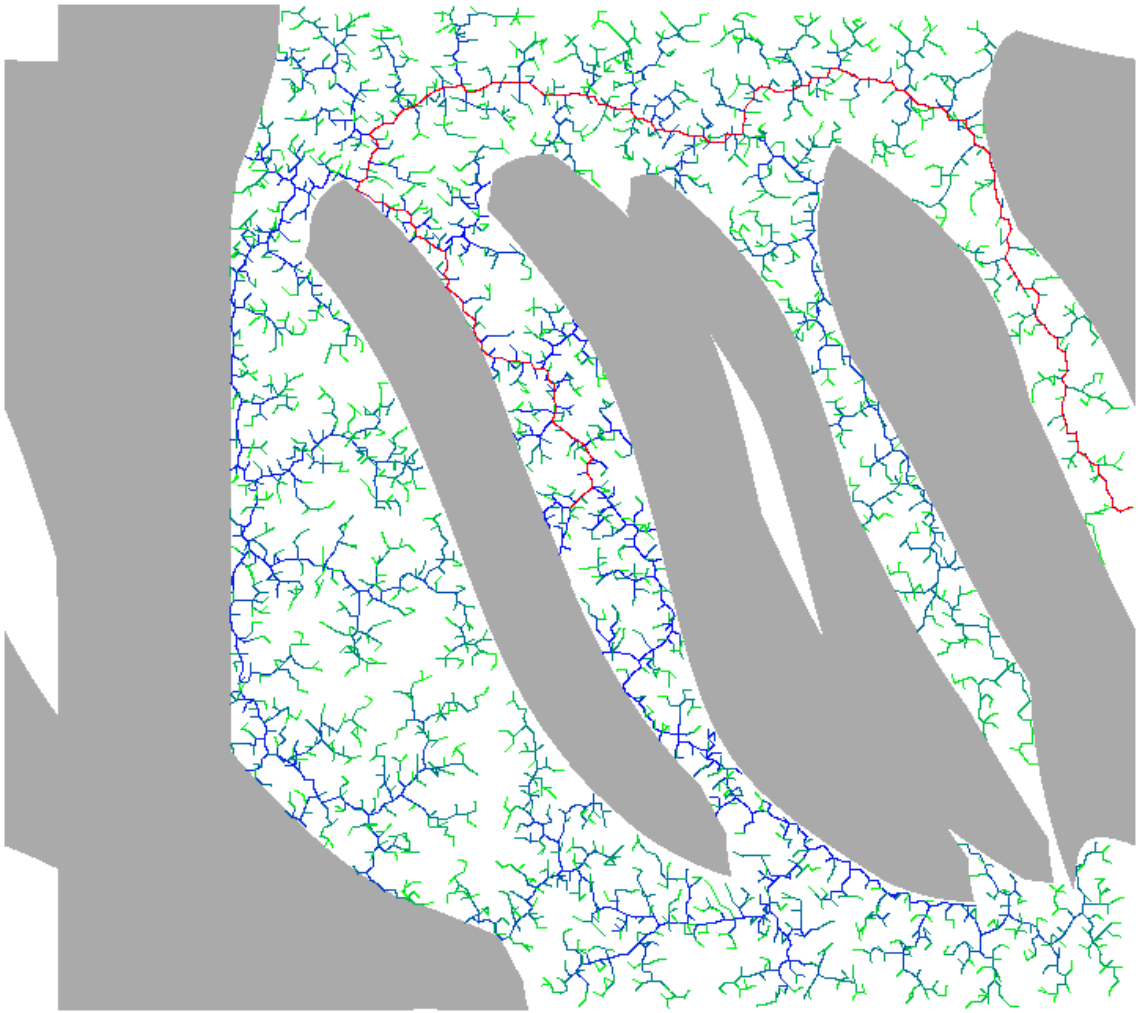


RRT Algorithm

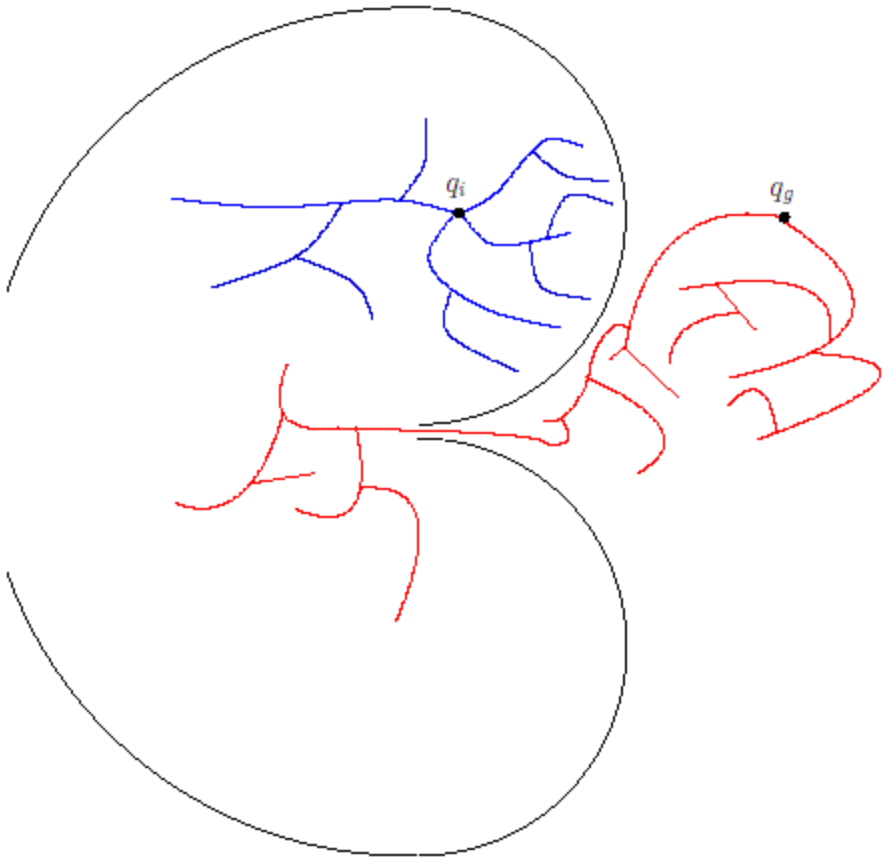


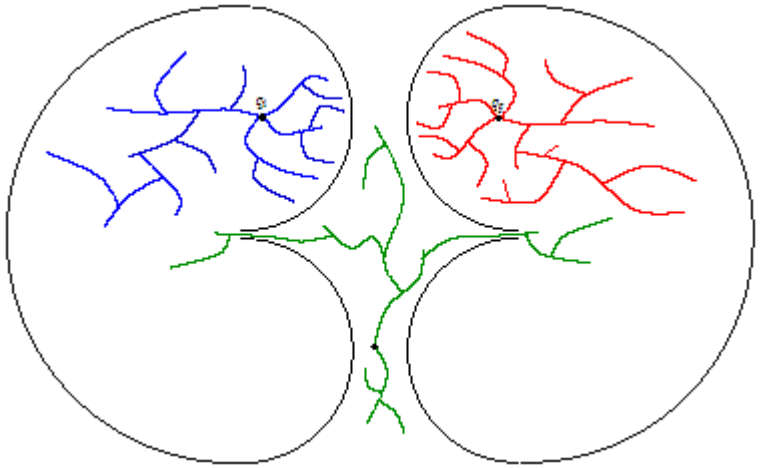
RRT Algorithm

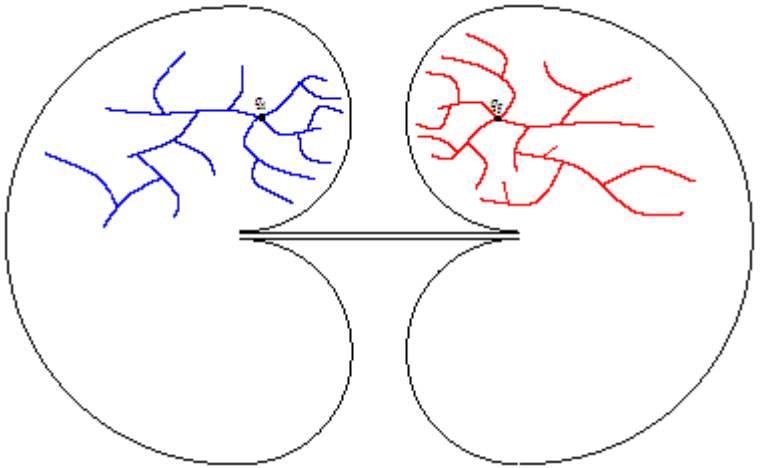
colors represent the growth of the RRT over time



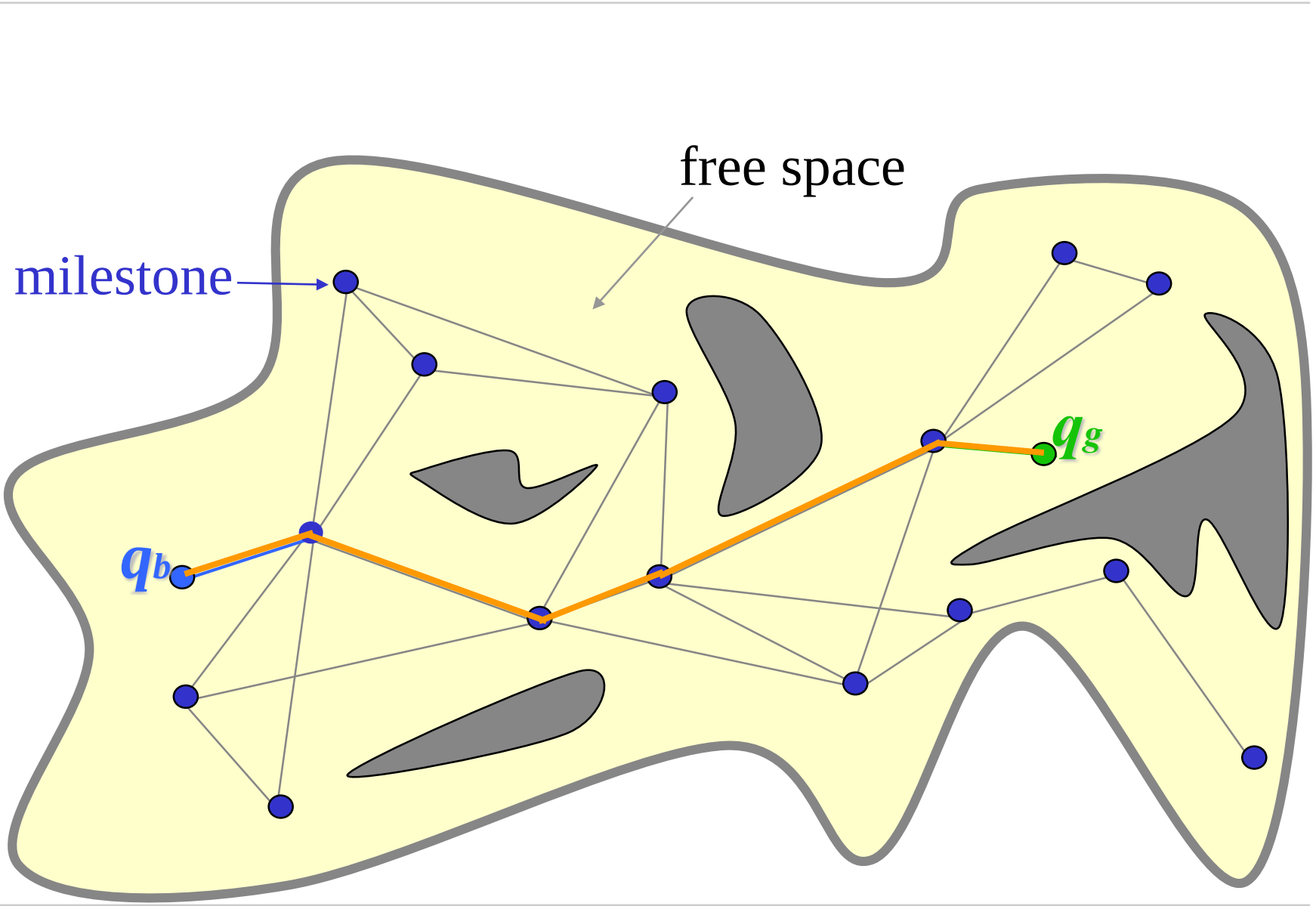
Bug traps motivates bidirectional search







Principle of Randomized Planning



PRM Planner …

in theory

- is probabilistically complete, i.e., whenever a solution exists, the probability that it finds one tends toward 1 as the number N of milestones increases
- under rather general hypotheses, the rate of convergence is exponential in the number N of milestones, i.e.:

$$\text{Prob[failure]} \sim \exp(-N)$$

in practise

- are fast
- deal effectively with many-dof robots
- are easy to implement
- have solved complex problems

Roadmap Construction

We shall construct the roadmap $R = (V, E)$ using the following algorithm:

1. Start with an empty graph ($V \leftarrow \emptyset$, $E \leftarrow \emptyset$).
2. Perform the following loop:
   - Select a random free configuration $c \in C_{\text{free}}$.
   - Randomly pick a set of neighboring configurations $V_c \subseteq V$.
   - $V \leftarrow V \cup \{c\}$
   - For each $q \in V_c$, in increasing order of $D(c,q)$, check if there is a free local path between c and q.
     If there exists such a path, add the edge: $E \leftarrow E \cup \{(c,q)\}$

Some Open Questions

How to create random free configurations?

It is possible to uniformly pick up a configuration and check if it is free.

How many neighbors should be chosen for each new configuration (the size of *Vc*)?

Pick up the closest neighbours.

How to define the distance function (in order to compute *D*(*c*,*q*))?

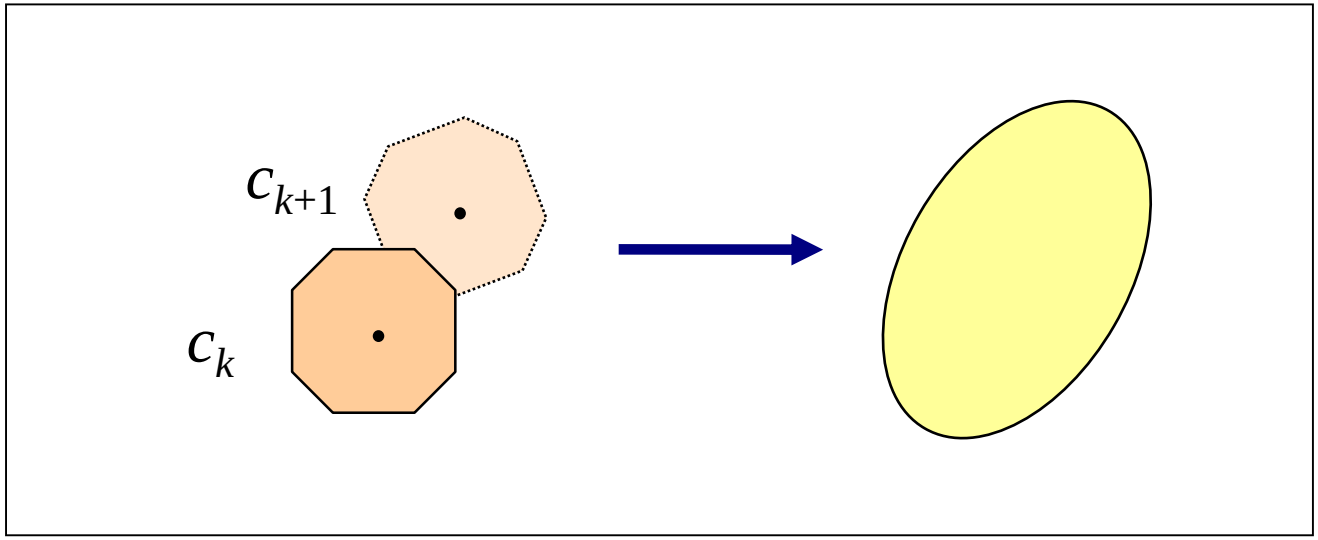
This depends on the nature of the problem ...

How to check if there exists a local path between two neighboring configurations?

We must rely on a local planner (preferably a deterministic one).

The Local Planner

Given two free configurations c' and c'', subdivide the straight line between them into m segments at the intermediate points:
$c_0 = c'$, c_1, … , $c_m = c''$.
Try to connect each pair of adjacent point c_k and c_{k+1}. This is done by inflating the robot, and the path existence query is thus reduced to an interference query.



Roadmap Expansion

After constructing the initial roadmap $\mathcal{R} = (V, E)$ we can use the following algorithm to improve its connectivity:

1. Pick up a set of "problematic" configurations $U \subseteq V$.
2. For each configuration $u \in U$.
   - Compute a random-walk track from u until reaching a new configuration c.
   - $V \leftarrow V \cup \{c\}$
   - Connect: $E \leftarrow E \cup \{(u,c)\}$ and store the path with this edge (as this path was created randomly).
   - Try to connect c with neighboring configurations that belong to different connected components of the roadmap.
3. Remove small connected components from the roadmap.

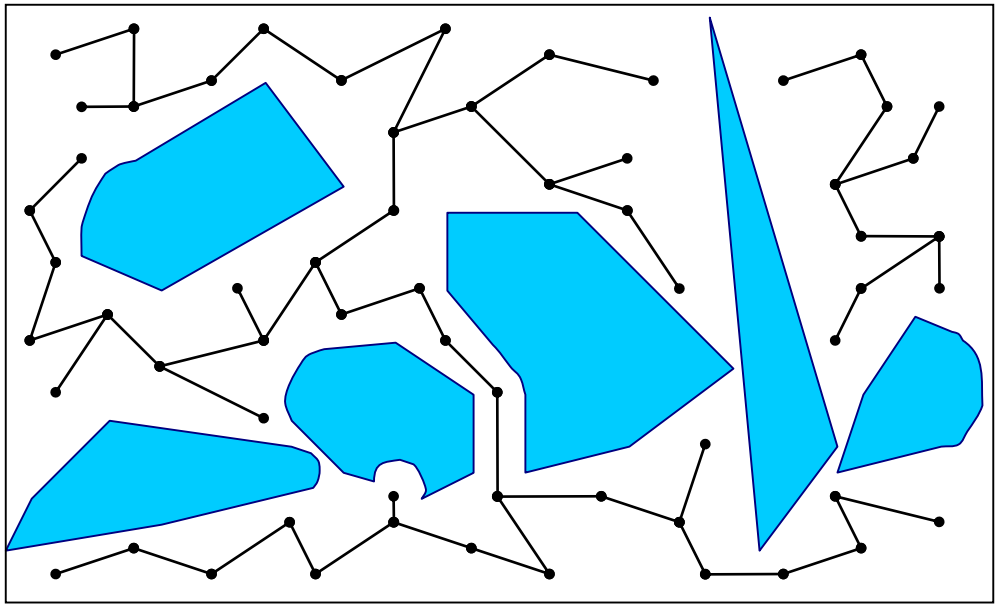
Answering Queries

Given a roadmap $\mathcal{R} = (V, E)$ with a start and target free configurations s and t:

1. Locate a vertex $s' \in V$ near s and try to connect s and s'. Report on failure if it is impossible to locate such s'.
2. Locate a vertex $t' \in V$ near t and try to connect t and t'. Report on failure if it is impossible to locate such t'.
3. In case of success, find a path between s' and t' in the roadmap. If the two vertices are not connected, report on failure.
4. If all went well, return the path $s \rightsquigarrow s' \rightsquigarrow t' \rightsquigarrow t$.

Analysis of Roadmaps

To have a better understanding of the capabilities of the probabilistic roadmaps, we should be able to estimate the number of configurations we should sample (or roadmap vertices) in order to capture the connectivity of the configuration space.



Path Planning in the Real World

Real World Robots

- Have inertia
- Have limited controllability
- Have limited sensors
- Face a dynamic environment
- Face an unreliable environment

Static planners like PRM are not sufficient

Extensions to the Basic Problem

- Moving obstacles
- Multiple robots
- Movable objects
- Deformable objects
- Goal is to gather data by sensing
- Nonholonomic constraints
- Dynamic constraints
- Optimal planning
- Uncertainty in control and sensing

Two Approaches to Path Planning

***Kinematic*: only concerns the motion, without regard to the forces that cause it**

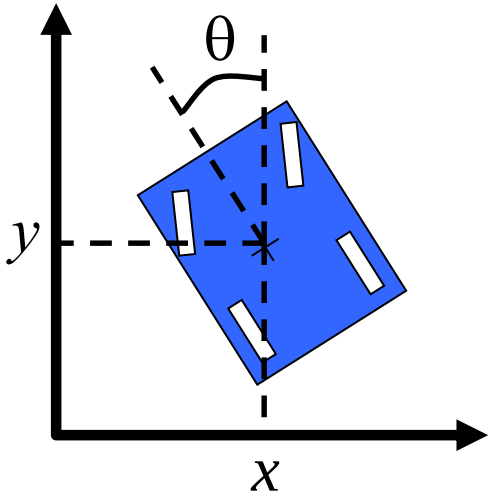
- Performs well for systems where position can be controlled directly
- Does not work well for systems with significant inertia

***Kinodynamic*: incorporates dynamic constraints**

- Plans velocity as well as position

Configuration space represents the position and orientation of a robot
Sufficient for static planners like PRM

Example: Steerable car
Configuration space
(x, y, q)



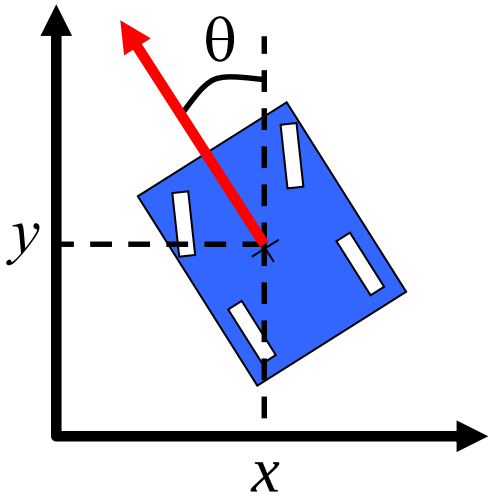
Representing Dynamic State

- State space incorporates the dynamic state of a robot

Example: Steerable car

State space

$X = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$



Working in state space allows planner to incorporate dynamic constraints on path

- *Examples:* maximum velocity, minimum turning radius

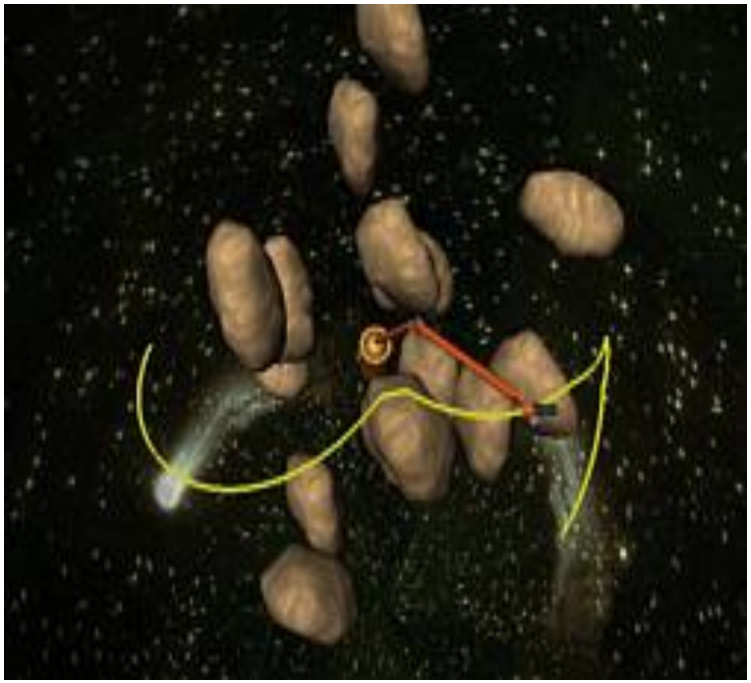
Working in state space doubles the dimensionality of the planning problem

- Math becomes exponentially harder

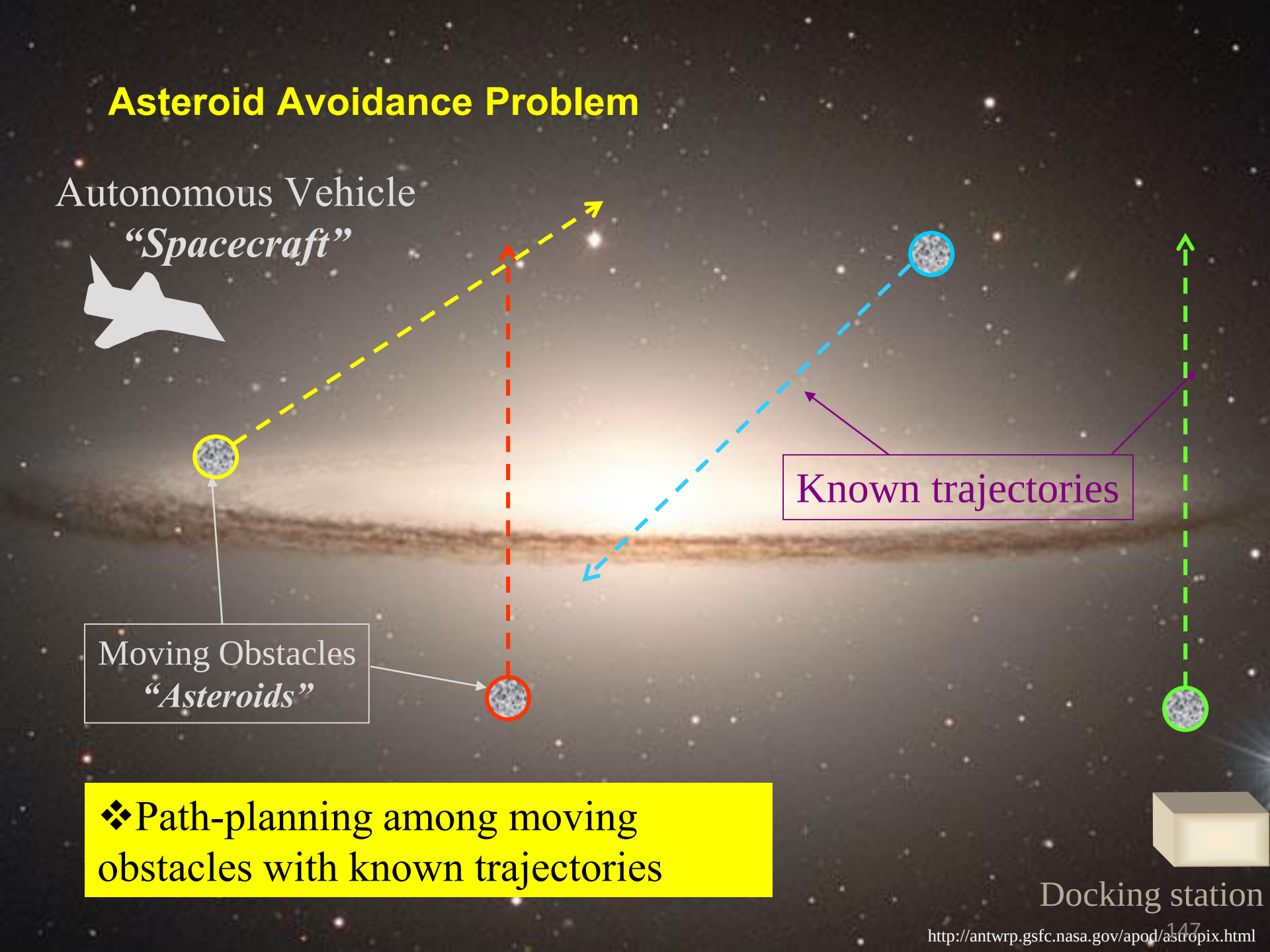
Planning Amidst Moving Obstacles

- Moving Obstacles Planner (MOP):
- A PRM extension that accounts for both kinematic and dynamic constraints of a robot navigating amidst moving obstacles
- We discuss: Kinodynamic motion planning amidst moving obstacles with known trajectories

- Example: Asteroid avoidance problem



Asteroid Avoidance Problem



❖Path-planning among moving obstacles with known trajectories

http://antwrp.gsfc.nasa.gov/apod/astropix.html

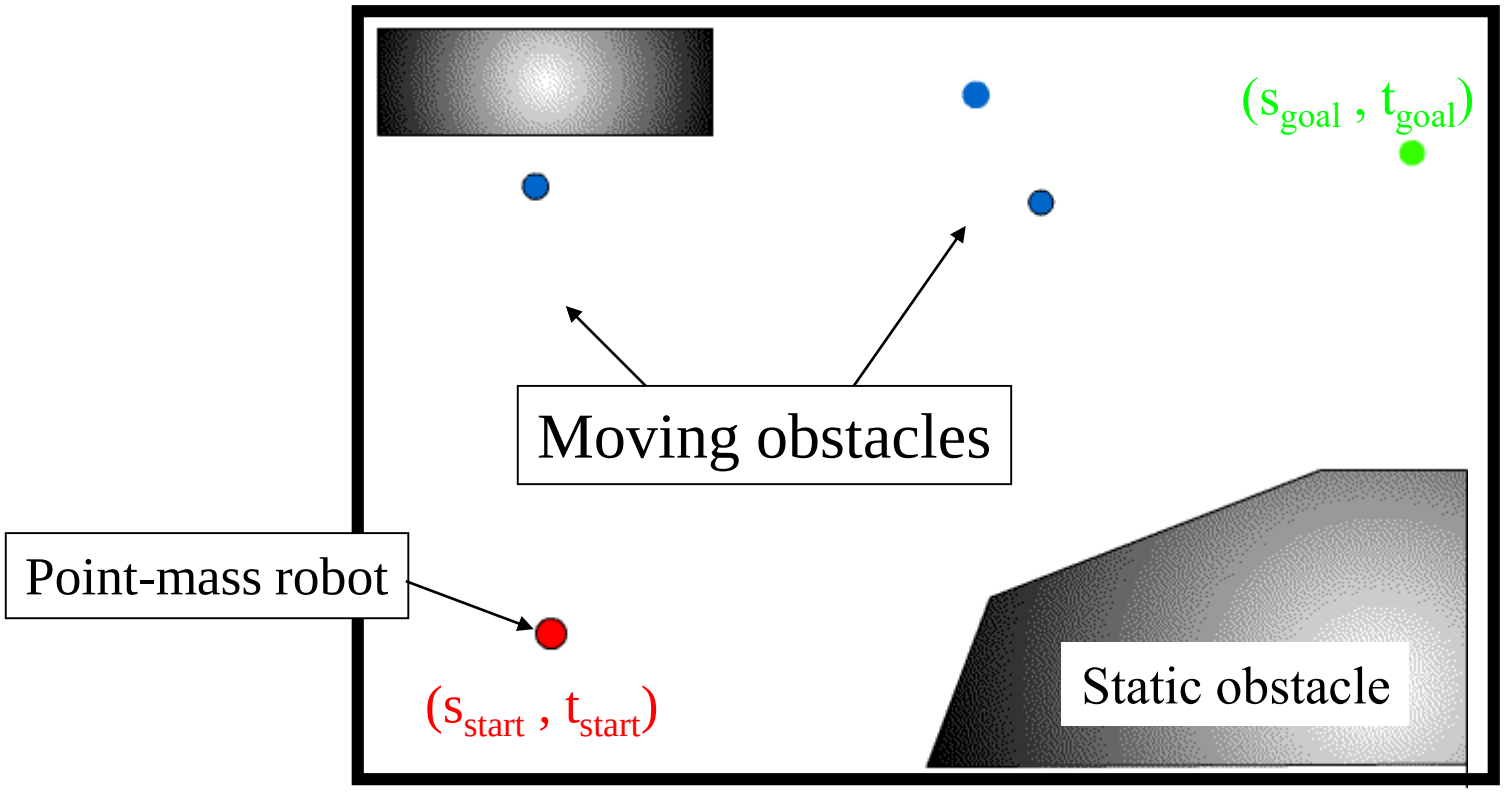
MOP Overview

- Similar to PRM, except
- Does not pre-compute the roadmap
- Incrementally constructs the roadmap by extending it from existing nodes (similar to RRT)
- Roadmap is a **directed tree** rooted at initial **state × time** point and oriented along time axis

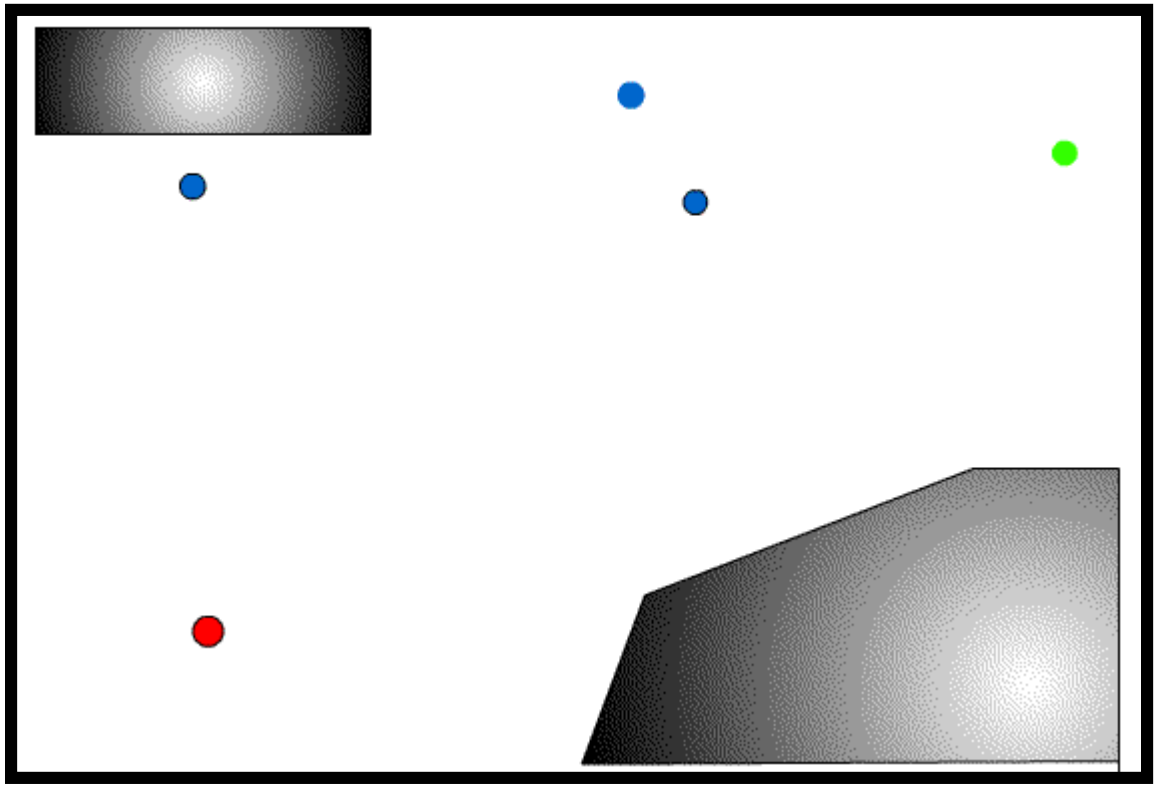
- For each query, the planner incrementally builds new roadmap in **state × time** space
  - ***Why?*** The environment includes moving obstacles that change location (state) continuously with time
  - Each node in the roadmap must be indexed by both its state and the time it is attained
  - Example: node *n* is valid at time t, however at time t+δ node *n* collides with a moving obstacle

Demonstration of MOP

- *Point–mass* robot moving in a plane
  - State s = $(x, y, \dot{x}, \dot{y})$



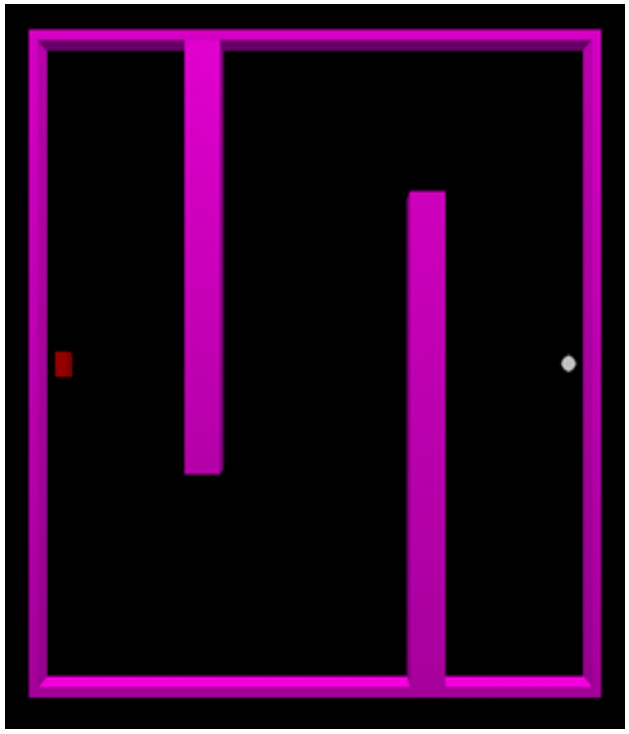
Demonstration of MOP

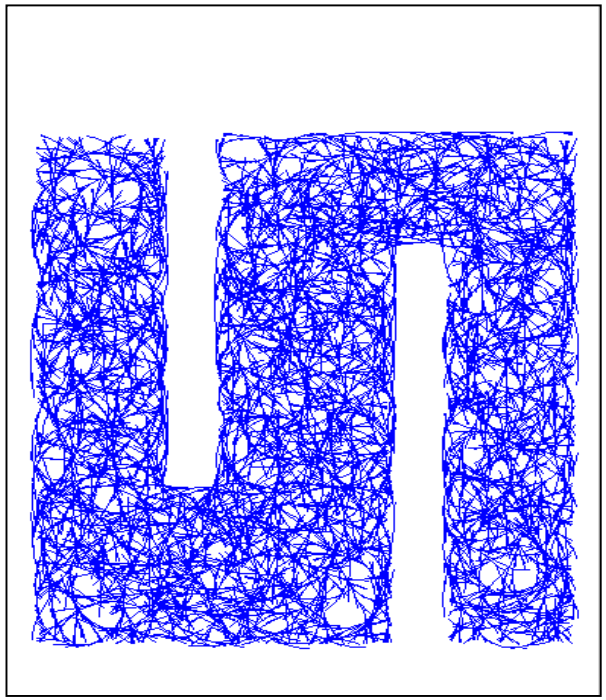


Planning with RRTs

- RRTs: Rapidly-exploring Random Trees
- Similar to MOP
  - Incrementally builds the roadmap tree
  - Integrates the control inputs to ensure that the kinodynamic constraints are satisfied
- Different exploration strategy from MOP
- Extends to more advanced planning techniques

Example: Simple RRT Planner



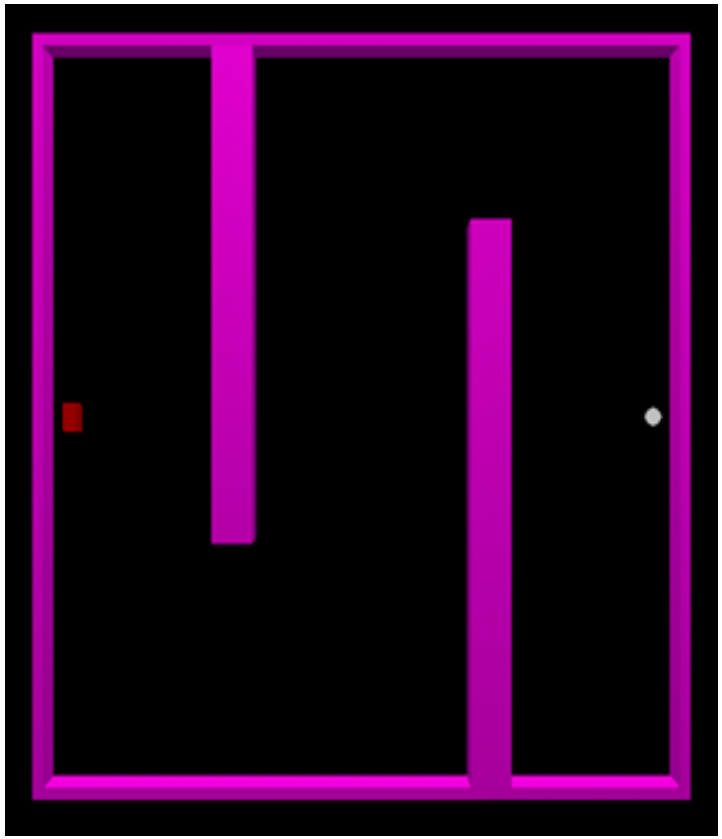


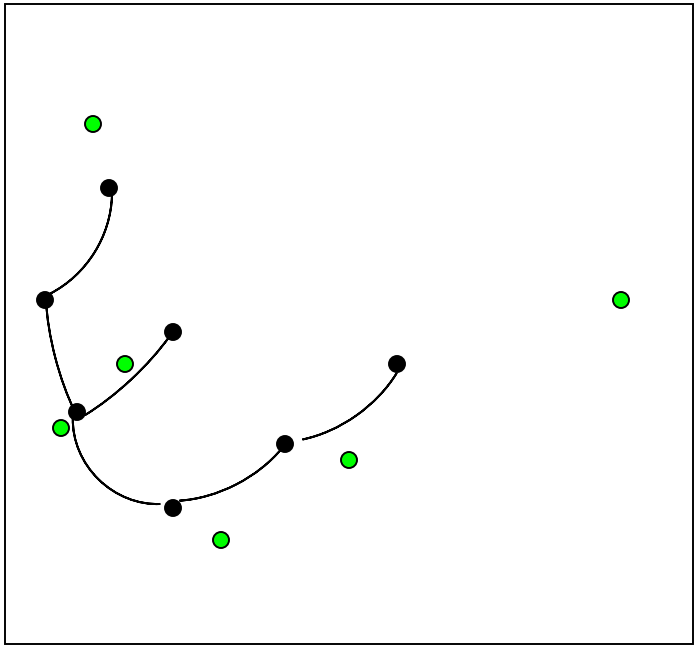
Problem: ordinary RRT explores *X* uniformly

→ slow convergence

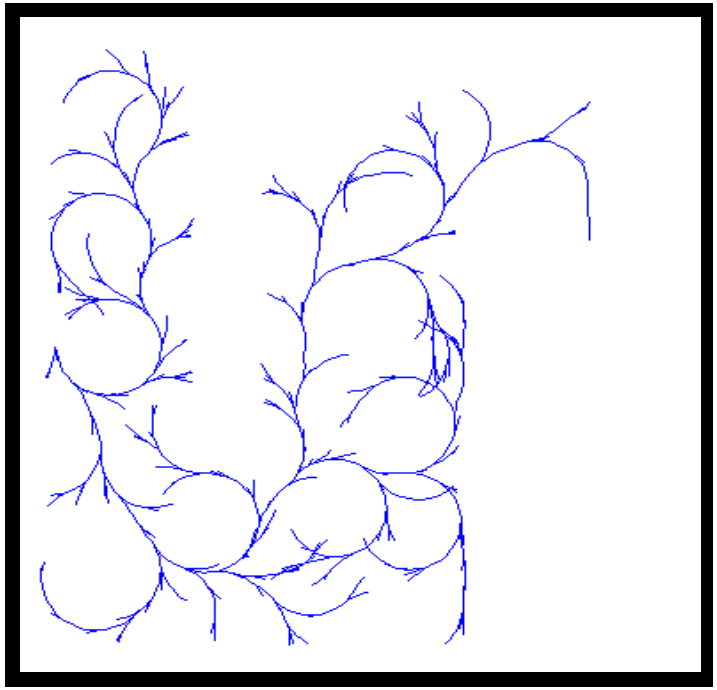
Solution: bias distribution towards the goal

Goal-biased RRT





The world is full of local minima

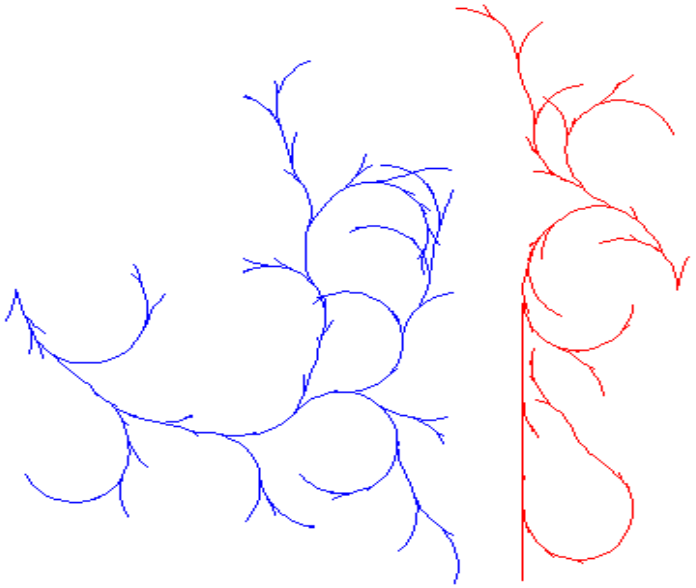


- If too much bias, the planner may get trapped in a local minimum
- A different strategy:
- Pick a point near s_{goal} based on distance from goal to the nearest *v* in *G*
- Gradual bias towards s_{goal}

Rather slow convergence

Bidirectional Planners

- **Build two RRTs, from start and goal state**



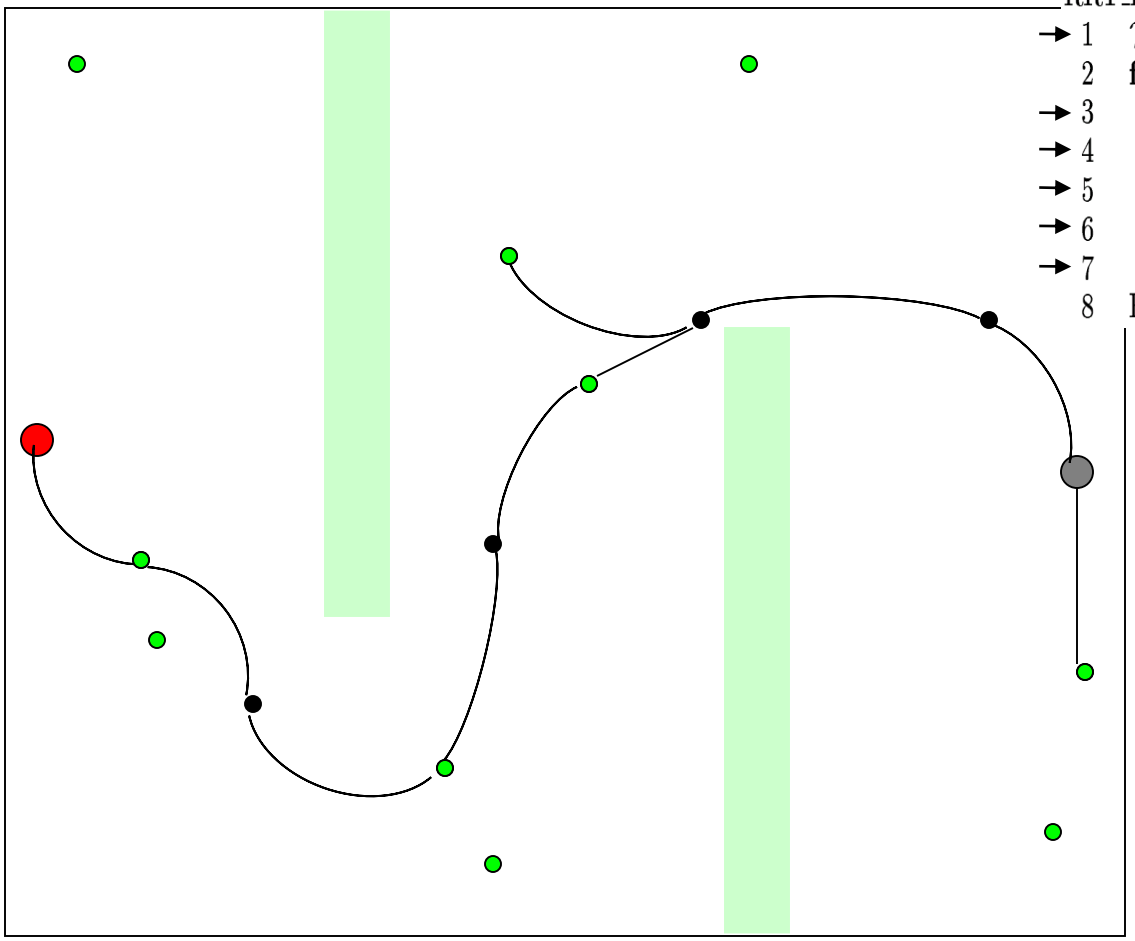
Complication: need to connect two RRTs

- local planner will not work (dynamic constraints)
- bias the distribution, so that the trees meet

Bidirectional Planner Algorithm

```
RRT_BIDIRECTIONAL(x_init, x_goal)
 1   T_a.init(x_init); T_b.init(x_goal);
 2   for k = 1 to K do
 3       x_rand ← RANDOM_STATE();
 4       if not (EXTEND(T_a, x_rand) = Trapped) then
 5           if (EXTEND(T_b, x_new) = Reached) then
 6               Return PATH(T_a, T_b);
 7       SWAP(T_a, T_b);
 8   Return Failure
```

Bidirectional Planner Example



RRT_BIDIRECTIONAL(x_{init}, x_{goal})
1 $\mathcal{T}_a$.init(x_{init}); $\mathcal{T}_b$.init(x_{goal});
2 **for** $k = 1$ **to** K **do**
3 $x_{rand} \leftarrow$ RANDOM_STATE();
4 **if not** (EXTEND($\mathcal{T}_a, x_{rand}$) = *Trapped*) **then**
5 **if** (EXTEND($\mathcal{T}_b, x_{new}$) = *Reached*) **then**
6 Return PATH($\mathcal{T}_a, \mathcal{T}_b$);
7 SWAP($\mathcal{T}_a, \mathcal{T}_b$);
8 Return *Failure*

Bidirectional Planner Example

Conclusions

- Path planners for real-world robots must account for dynamic constraints
- Building the roadmap tree incrementally
  - ensures that the kinodynamic constraints are satisfied
  - avoids the need to reconstruct control inputs from the path
  - allows extensions to moving obstacles problem
- MOP and RRT planners are similar
- Well-suited for single-query problems
- RRTs benefit from the ability to steer a robot toward a point
  - RRTs explore the state more uniformly
  - RRTs can be biased towards a goal or to grow into another RRT

Motion Planning based on (only) local knowledge

- Based on local (not global) knowledge
- Bug algorithms

- Family of algorithms
- Bug 0
- Bug 1
- Bug 2
- Tangent Bug

What's Special About Bugs

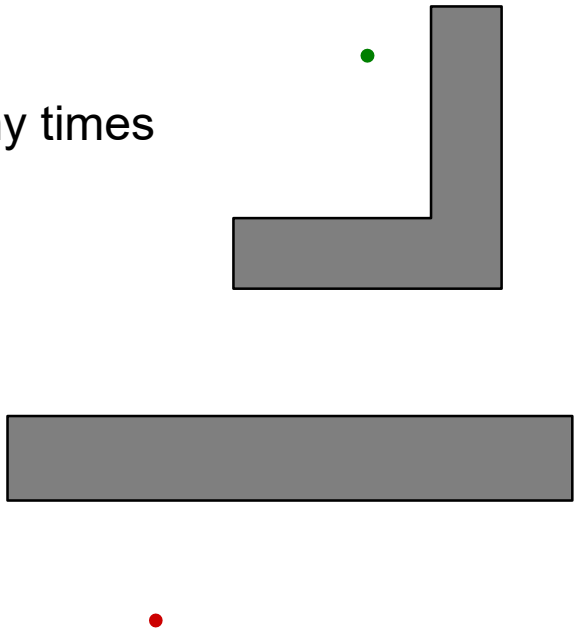
- Based on local (not global) knowledge
- Bug algorithms assume only local knowledge of the environment and a global goal
- Bug behaviors are simple:
  - 1) Follow a wall (right or left)
  - 2) Move in a straight line toward goal
- Family of algorithms called Bug algorithms:
- Bug 0
- Bug 1 (assume essentially tactile sensing)
- Bug 2 (assume essentially tactile sensing)
- Tangent Bug (deals with finite distance sensing)

A Few General Concepts

- Workspace W
  - R2 or R3 depending on the robot
  - could be infinite (open) or bounded (compact)
- Obstacles WO_i
- Free workspace $W_{free} = W \cup WO_i$
- Some notation:
- q_{start} and q_{goal}
- “hit point” $q_{H,i}$
- „leave point“ $q_{L,i}$
- A path is a sequence of hit/leave pairs bounded by q_{start} and q_{goal}

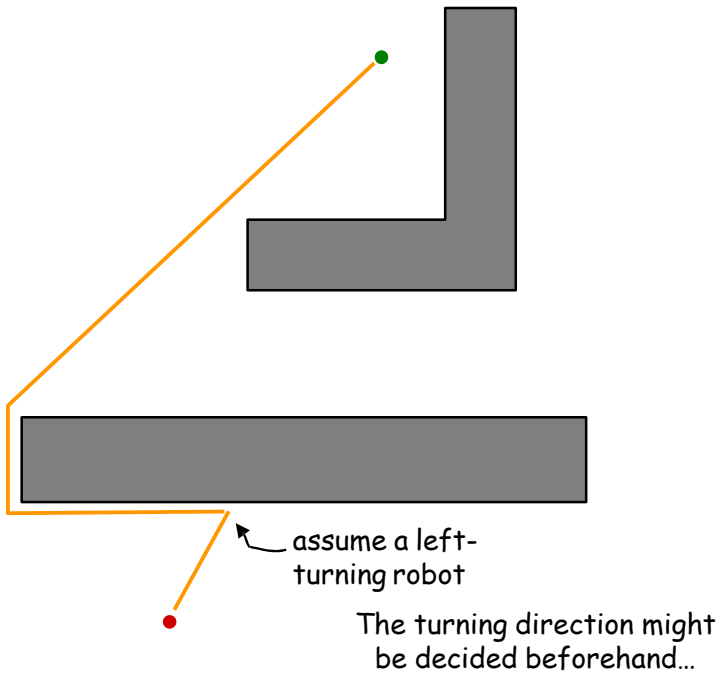
Assumptions

- known direction to goal
  - robot can measure distance d(x,y) between pts x and y
- otherwise local sensing
  - walls/obstacles & encoders
- reasonable world
- finitely many obstacles in any finite area
- a line will intersect an obstacle finitely many times
- Workspace is bounded
  - $W \subset B_r(x), r < \infty$
  - $B_r(\mathrm{x}) = \{\mathrm{y} \in R^2 | d(x,y) < r\}$



Beginner Strategy - Bug 0

- known direction to goal
  - robot can measure distance d(x,y) between pts x and y
- otherwise local sensing
  - walls/obstacles & encoders



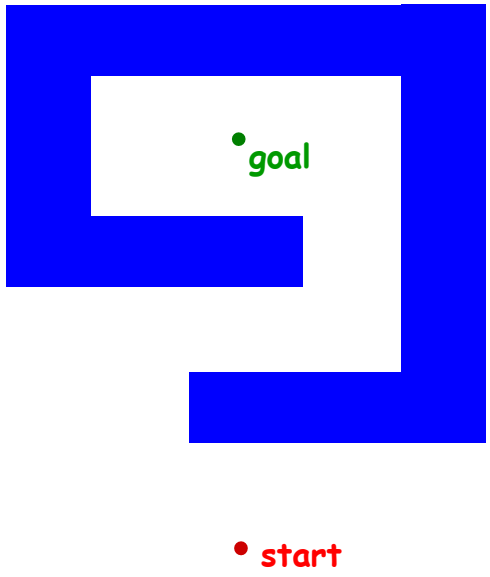
1) head toward goal

2) follow obstacles until you can head toward the goal again

3) continue

Beginner Strategy - Bug 0

- What map will foil Bug 0 ?



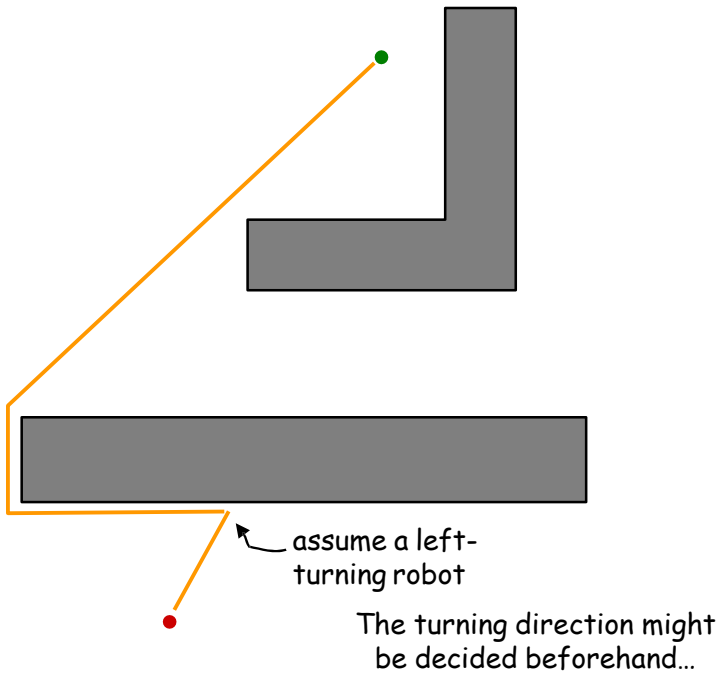
1) head toward goal

2) follow obstacles until you can head toward the goal again

3) continue

Bug 1

- known direction to goal
  - robot can measure distance d(x,y) between pts x and y
- otherwise local sensing
  - walls/obstacles & encoders



1) head toward goal

2) follow obstacles until you can head toward the goal again

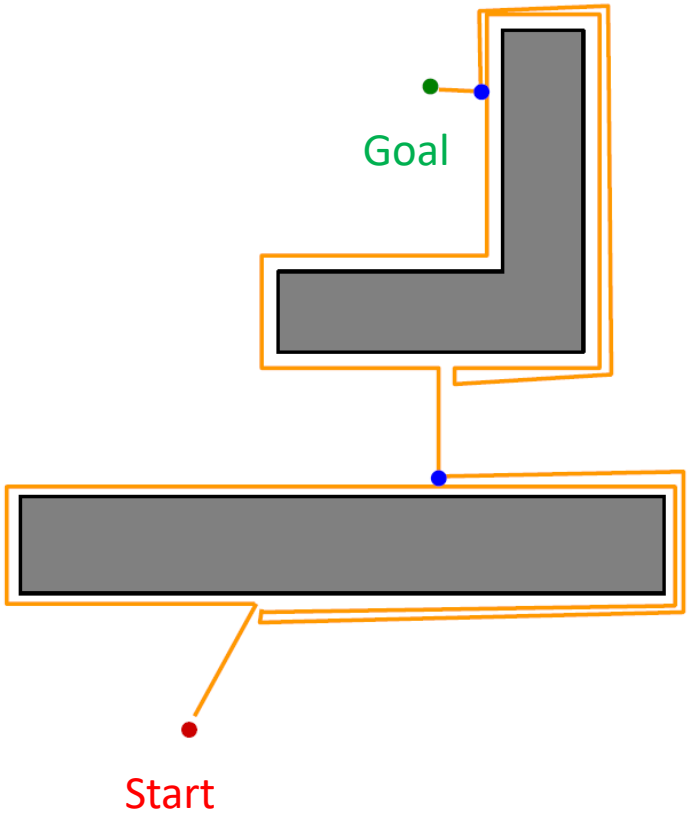
3) continue

Bug 1

But some computing power!

- known direction to goal
- otherwise local sensing

walls/obstacles & encoders



"Bug 1" algorithm

1) head toward goal

2) if an obstacle is encountered, circumnavigate it *and* remember how close you get to the goal

3) return to that closest point (by wall-following) and continue

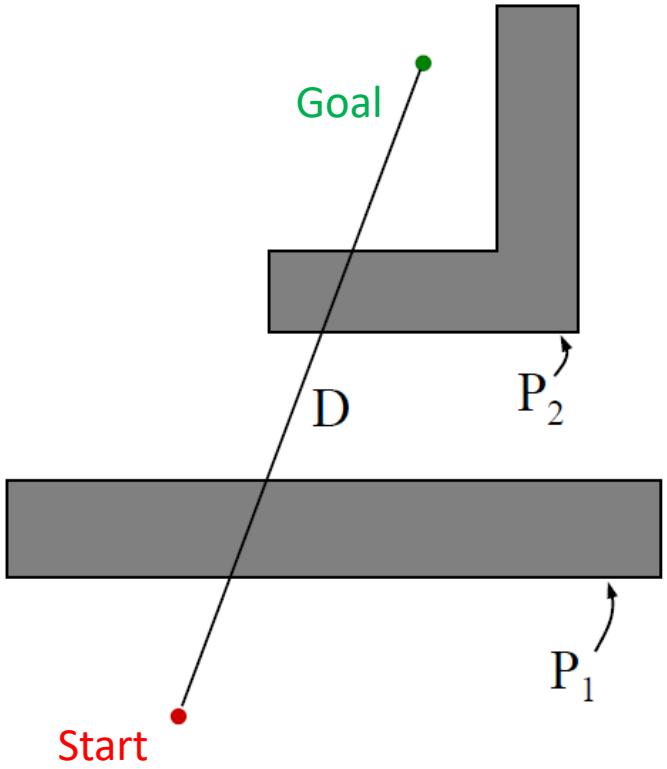
Bug 1 - more formally

- Let $q^L_0 = q_{start}$; i = 1
- repeat
  - repeat
    - from q^L_{i-1} move toward q_{goal}
  - until goal is reached or obstacle encountered at q^H_i
  - if goal is reached, exit
  - repeat
    - follow boundary recording pt q^L_i with shortest distance to goal
  - until q_{goal} is reached or q^H_i is re-encountered
  - if goal is reached, exit
  - Go to q^L_i
  - if move toward q_{goal} moves into obstacle
    - exit with failure
  - else
    - i=i+1
    - continue

Bug 1 analysis

Bug 1: Path Bounds

What are upper/lower bounds on the path length that the robot takes?



D = straight-line distance from start to goal

P_i = perimeter of the *i* th obstacle

Lower bound:

What's the shortest distance it might travel?

$$D$$

Upper bound:

What's the longest distance it might travel?

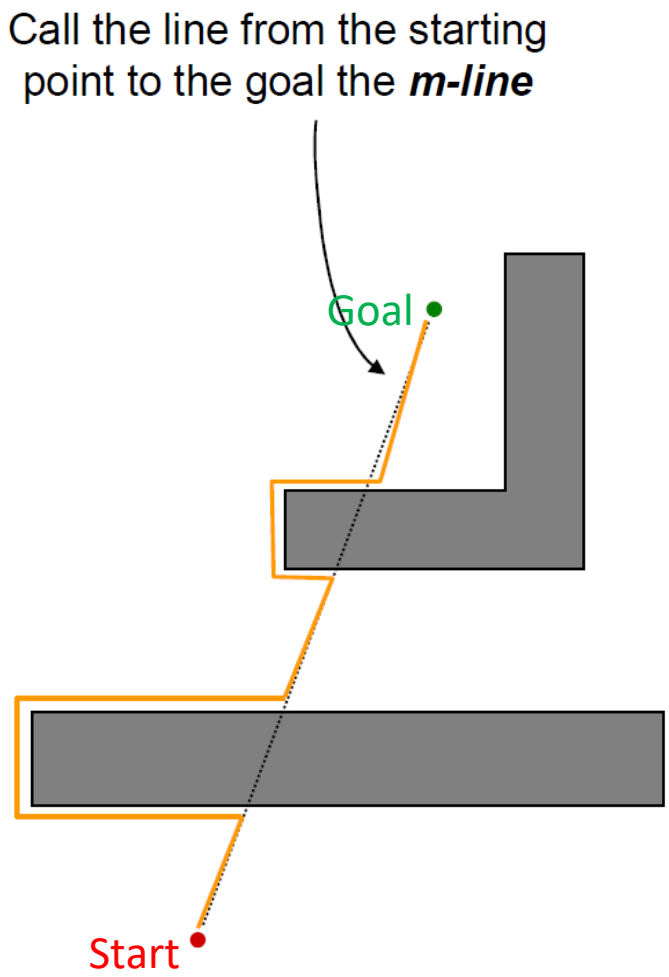
$$D + 1.5 \sum_i P_i$$

How Can We Show Completeness?

- An algorithm is *complete* if, in finite time, it finds a path if such a path exists or terminates with failure, if it does not.

- Suppose BUG1 were incomplete
  - Therefore, there is a path from start to goal
    - By assumption, it is finite length, and intersects obstacles a finite number of times.
  - BUG1 does not find it
    - Either it terminates incorrectly, or, it spends an infinite amount of time
    - Suppose it never terminates
      - but each leave point is closer to the obstacle than corresponding hit point
      - Each hit point is closer than the last leave point
      - Thus, there are a finite number of hit/leave pairs; after exhausting them, the robot will proceed to the goal and terminate
    - Suppose it terminates (incorrectly)
    - Then, the closest point after a hit must be a leave where it would have to move into the obstacle
      - But, the line from robot to goal must intersect object even number of times (Jordan curve theorem)
      - But then there is another intersection point on the boundary closer to object. Since we assumed there is a path, we must have crossed this pt on boundary which contradicts the definition of a leave point.

Bug 2



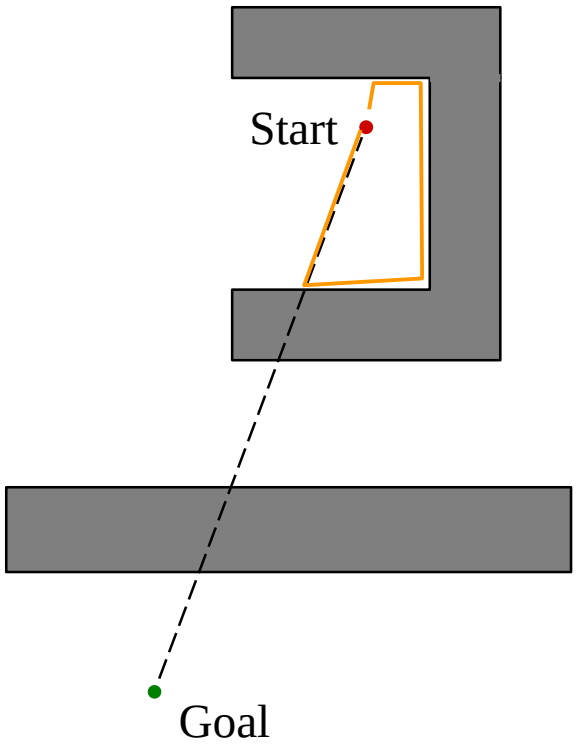
"Bug 2" Algorithm

1) head toward goal on the *m-line*

2) if an obstacle is in the way, follow it until you encounter the m-line again.

3) Leave the obstacle and continue toward the goal

Bug 2 – a better bug



"Bug 2" Algorithm

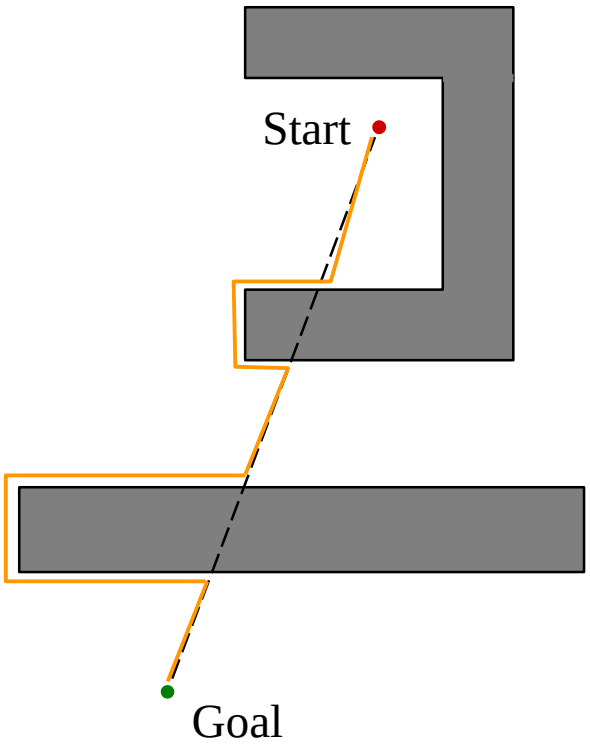
1)head toward goal on the *m-line*

2)if an obstacle is in the way, follow it until you encounter the m-line again.

3)Leave the obstacle and continue toward the goal

NO! How do we fix this?

Bug 2 – a better bug



"Bug 2" Algorithm

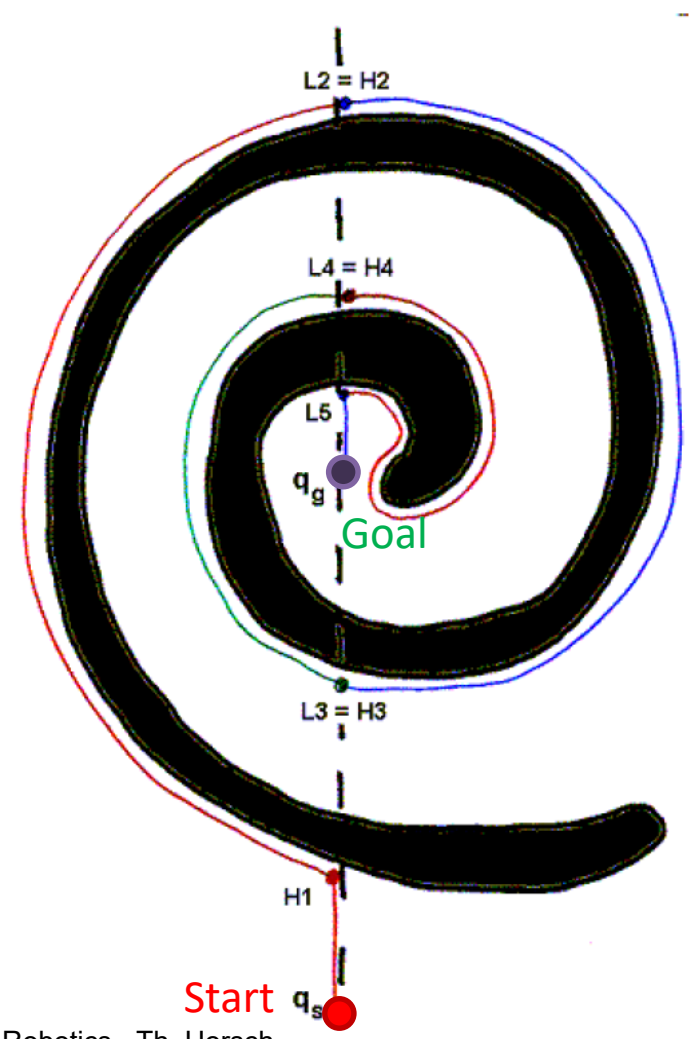
1)head toward goal on the *m-line*

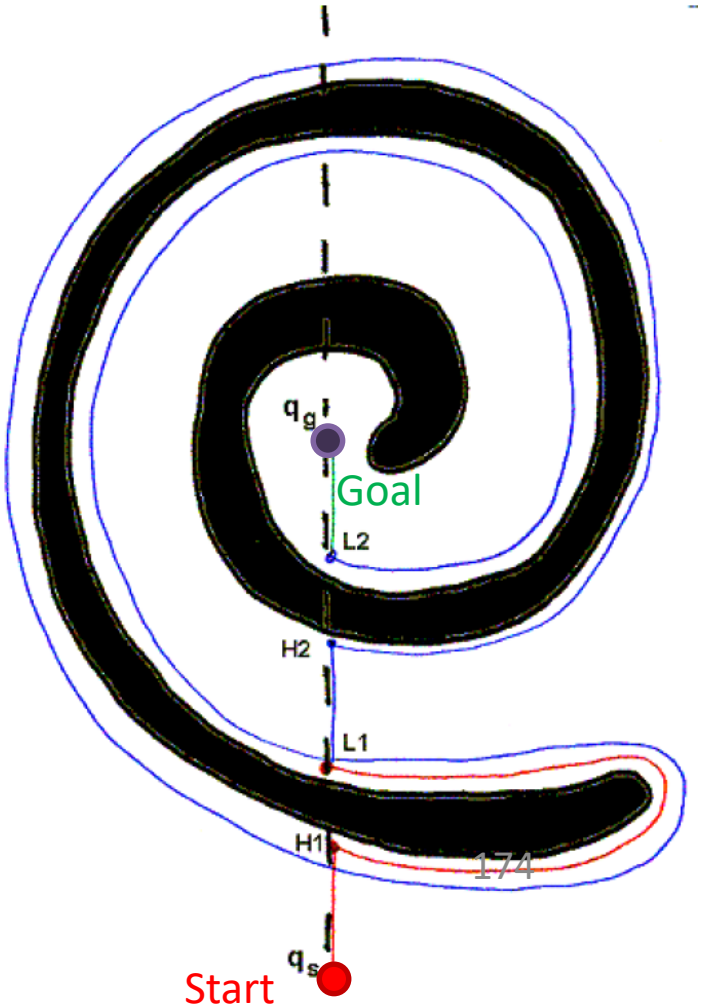
2)if an obstacle is in the way, follow it until you encounter the m-line again ***closer to the goal***.

3)Leave the obstacle and continue toward the goal

Better or worse than Bug1?

The Spiral

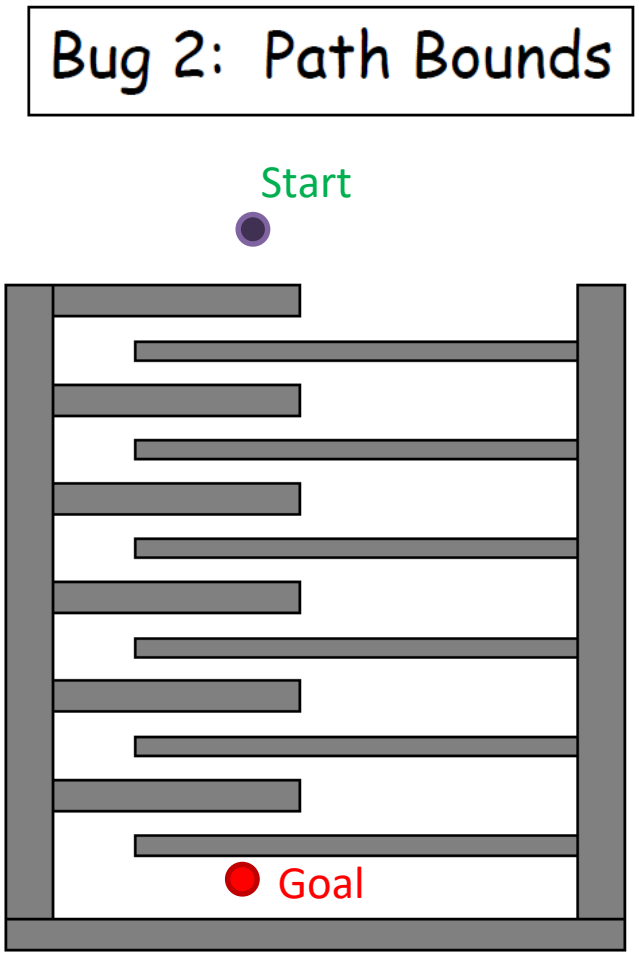

L2 = H2
L4 = H4
L5
q_g
Goal
L3 = H3
H1
Start q_s


q_g
Goal
L2
H2
L1
H1
q_s
Start

Bug 2 More formally

- Let $q^L_0 = q_{start;}$ i = 1
- repeat
  - repeat
    - from q^L_{i-1} move toward q_{goal} along the m-line
  - until goal is reached or obstacle encountered at q^H_i
  - if goal is reached, exit
  - repeat
    - follow boundary
  - until q_{goal} is reached or q^H_i is re-encountered or m-line is re-encountered, x is not q^H_i, $d(x,q_{goal}) < d(q^H_i,q_{goal})$ and way to goal is unimpeded
  - if goal is reached, exit
  - if q^H_i is reached, return failure
  - else
    - $q^L_i = m$
    - i=i+1
    - continue

Bug 2 analysis



What are upper/lower bounds on the path length that the robot takes?

D = straight-line distance from start to goal

P_i = perimeter of the *i* th obstacle

Lower bound:

What's the shortest distance it might travel?

$$D$$

Upper bound:

What's the longest distance it might travel?

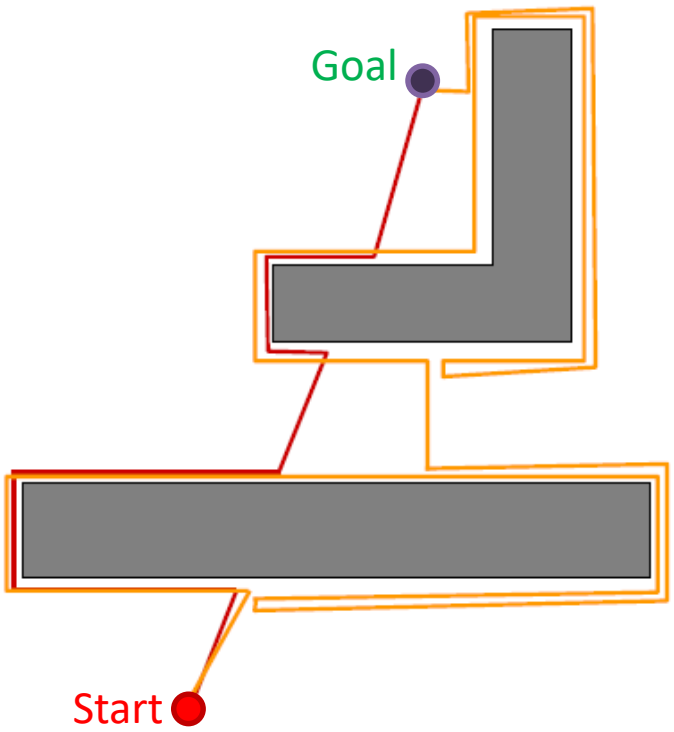
$$D + \sum_i \frac{n_i}{2} P_i$$

n_i = # of s-line intersections of the *i* th obstacle

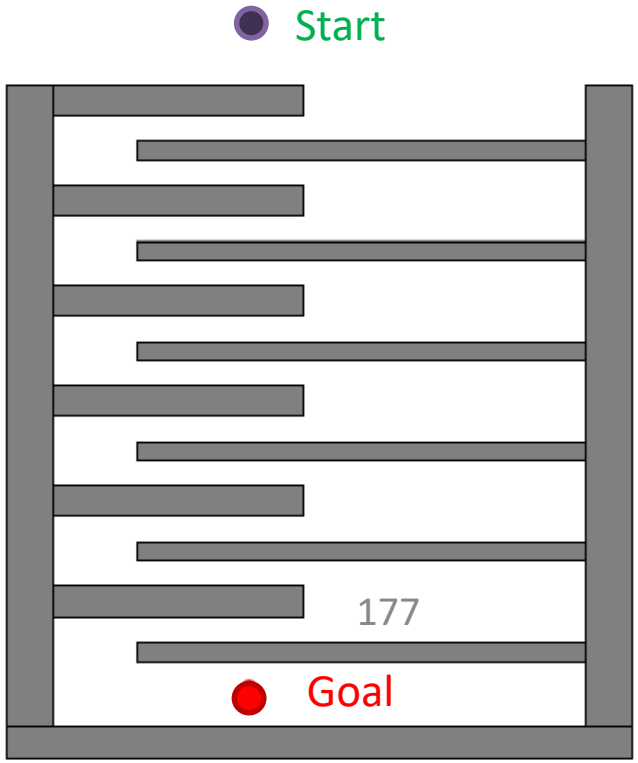
head-to-head comparison

Draw worlds in which Bug 2 does better than Bug 1 (and vice versa).

Bug 2 beats Bug 1



Bug 1 beats Bug 2



BUG 1 vs. BUG 2

- BUG 1 is an *exhaustive search algorithm*
  - it looks at all choices before commiting
- BUG 2 is a *greedy* algorithm
  - it takes the first thing that looks better
- In many cases, BUG 2 will outperform BUG 1, but
- BUG 1 has a more predictable performance overall

Tangent BUG - Raw Distance Function

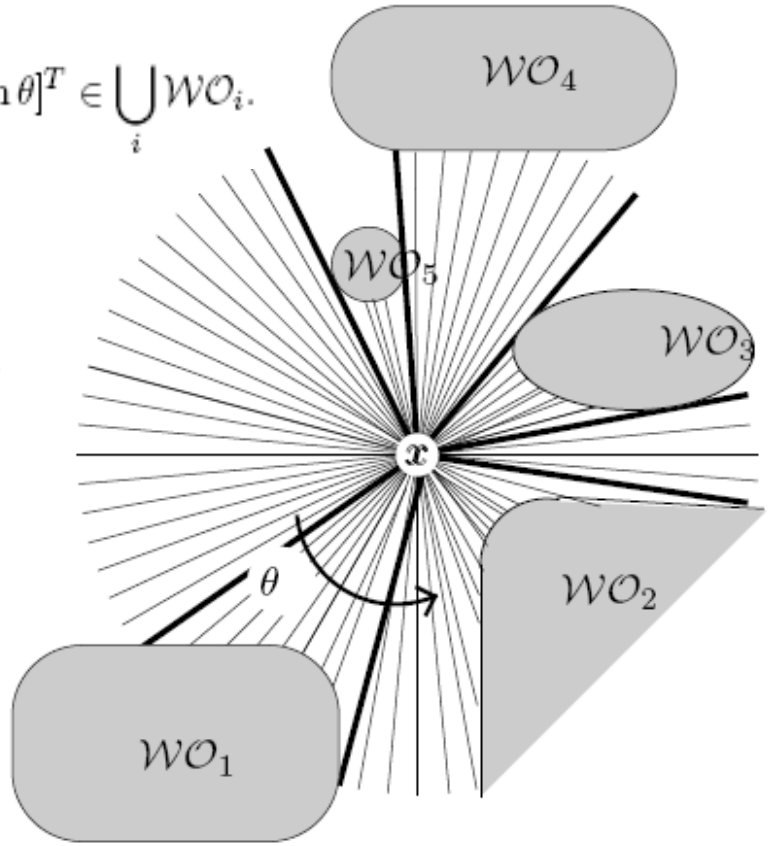
$$\rho(x,\theta) = \min_{\lambda \in [0,\infty]} d(x, x + \lambda[\cos\theta, \sin\theta]^T),$$

$$\text{such that} \quad x + \lambda[\cos\theta, \sin\theta]^T \in \bigcup_i \mathcal{WO}_i.$$

$$\rho : \mathbb{R}^2 \times S^1 \longrightarrow \mathbb{R}$$

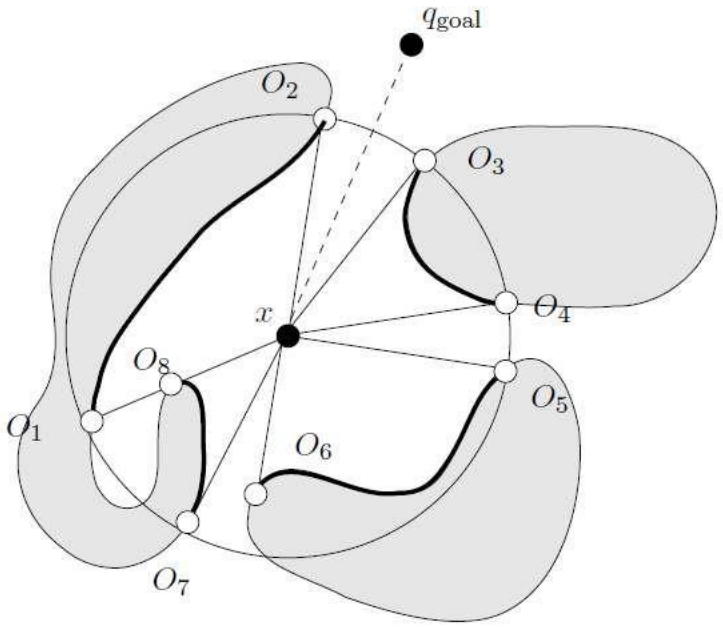
Saturated raw distance function

$$\rho_R(x,\theta) = \begin{cases} \rho(x,\theta), & \text{if } \rho(x,\theta) < R \\ \infty, & \text{otherwise.} \end{cases}$$



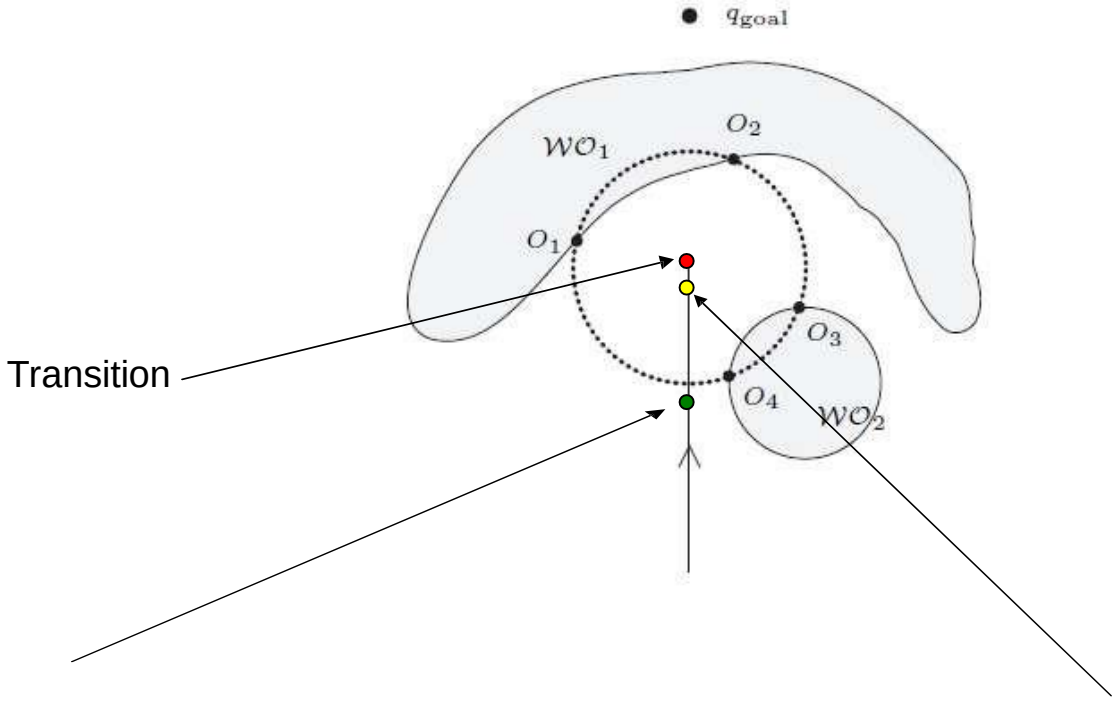
Intervals of Continuity

- Tangent Bug relies on finding endpoints of finite, continuous segments of ρ_R



- Concave obstacles can have few segments

Motion-to-Goal Transition from Moving Toward goal to "following obstacles"



Currently, the motion-to-goal behavior "thinks" the robot can get to the goal

Now, it starts to see something --- what to do?

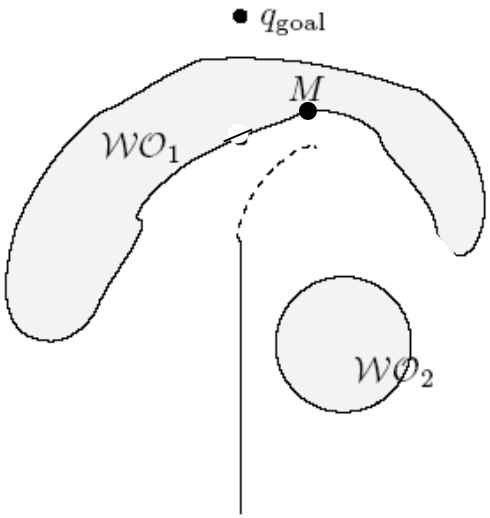
Answer: For any O_i such that $d(O_i,q_{goal}) < d(x,q_{goal})$, choose the pt O_i that minimizes $d(x,O_i) + d(O_i,q_{goal})$

Transition *from* Motion-to-Goal

Choose the pt O_i that minimizes $d(x,O_i) + d(O_i,q_{goal})$

Problem: what if this distance starts to go up?

Answer: start to act like a BUG and follow boundary



M is the point on the "sensed" obstacle which has the shortest distance to the goal

Followed obstacle: the obstacle that we are currently sensing

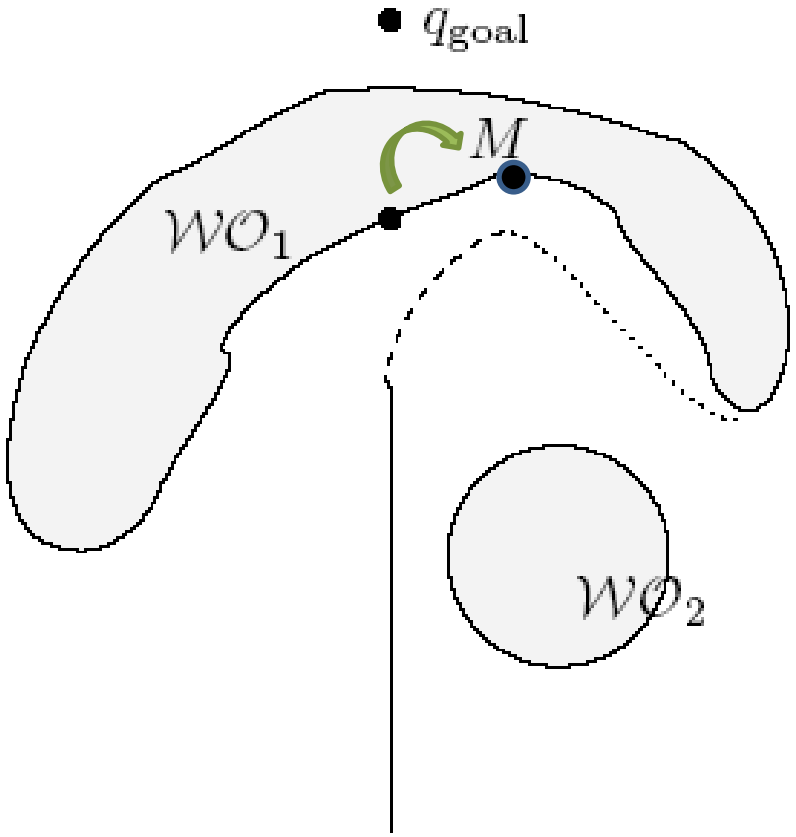
Blocking obstacle: the obstacle that intersects the segment

$$(1-\lambda)x + \lambda q_{goal} \quad \forall \lambda \in [0,1]$$

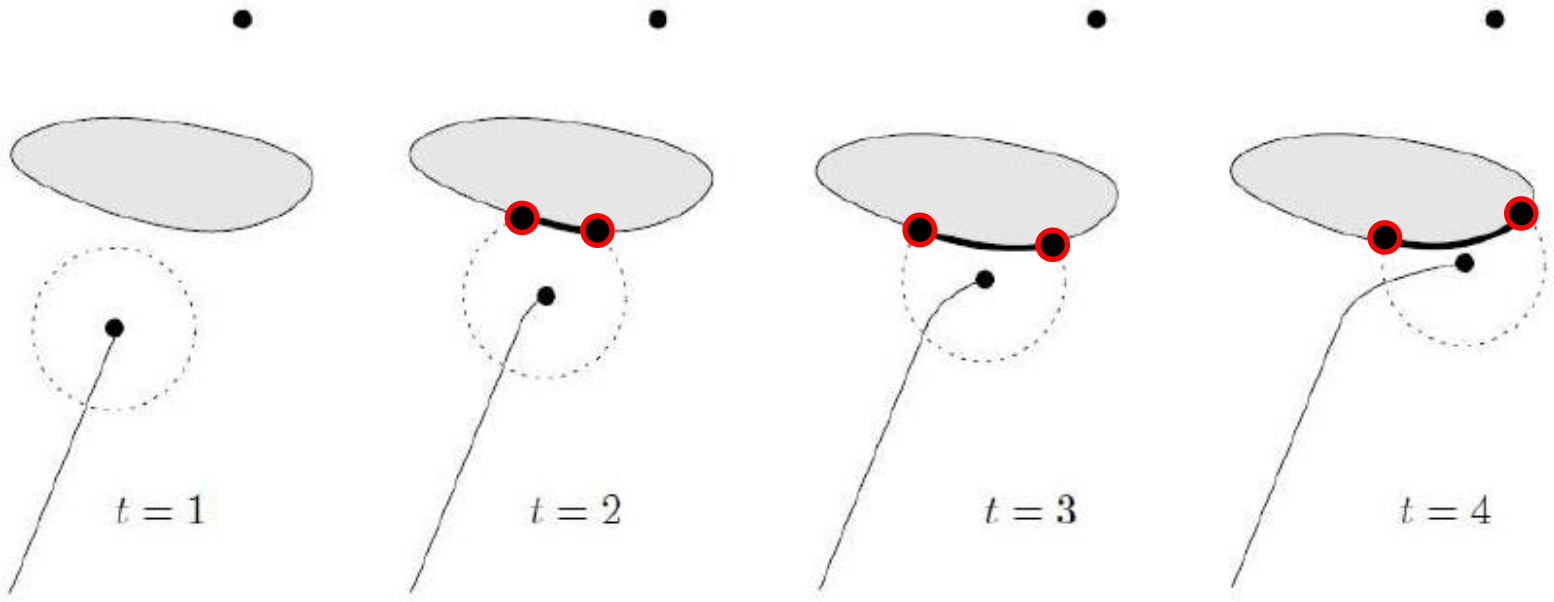
They start as the same

For any O_i such that $d(O_i,q_{goal}) < d(x,q_{goal})$,
choose the pt O_i that minimizes $d(x,O_i) + d(O_i,q_{goal})$

Transition ***from Motion-to-Goal***



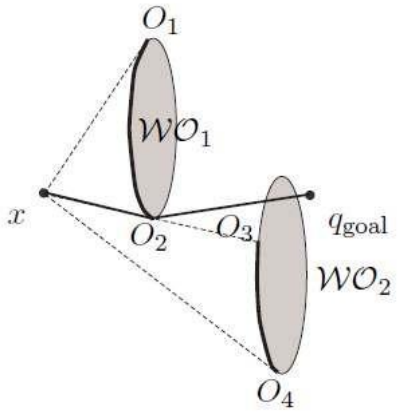
Motion To Goal Example



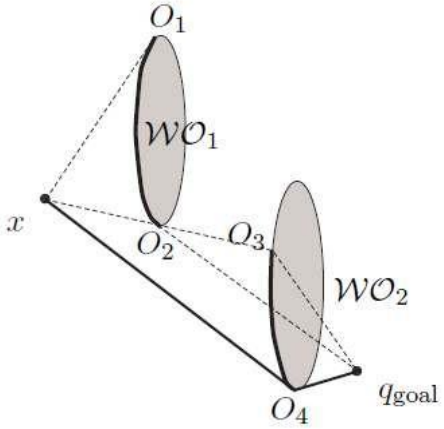
For any O_i such that $d(O_i,q_{goal}) < d(x,q_{goal})$,
choose the pt O_i that minimizes $d(x,O_i) + d(O_i,q_{goal})$

Minimize Heuristic Example

At x, robot knows only what it sees and where the goal is,



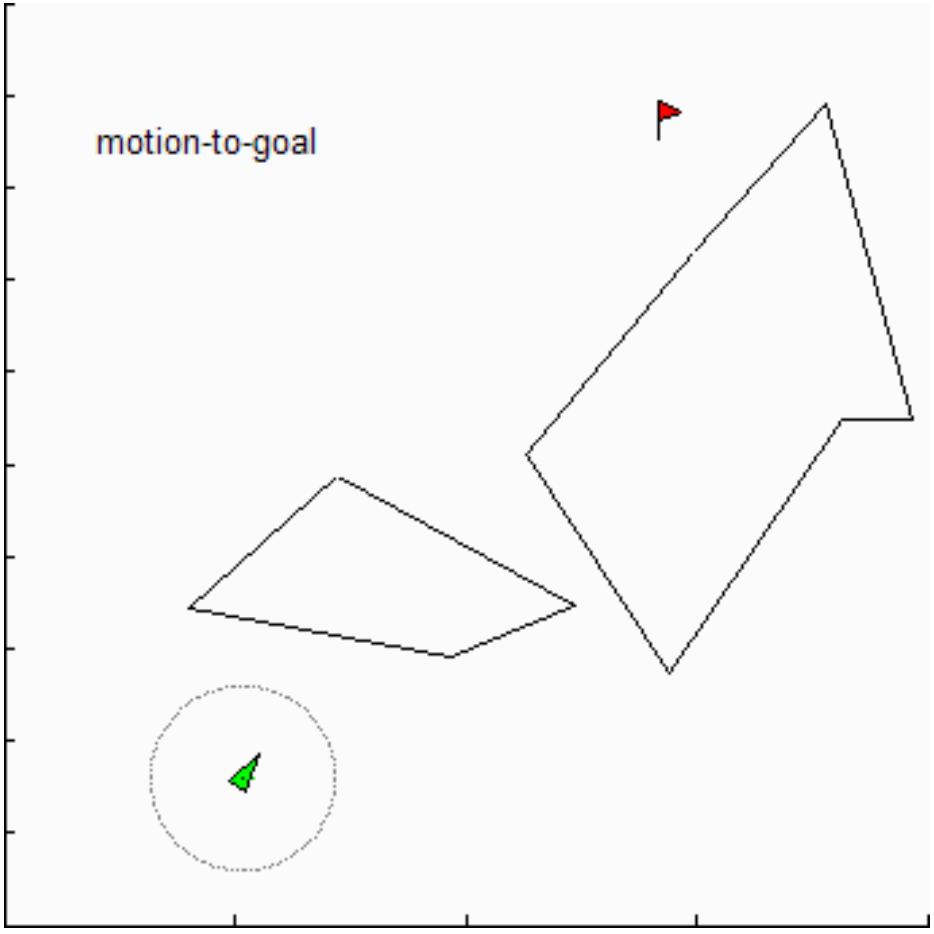
so moves toward O_2. Note the line connecting O_2 and goal pass through obstacle



so moves toward O_4. Note some "thinking" was involved and the line connecting O_4 and goal pass through obstacle

For any O_i such that $d(O_i,q_{goal}) < d(x,q_{goal})$,
choose the pt O_i that minimizes $d(x,O_i) + d(O_i,q_{goal})$

Motion To Goal Example

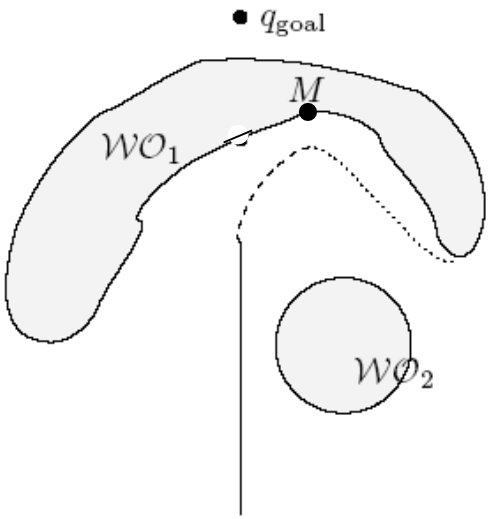


The Basic Ideas

- A **motion-to-goal** behavior as long as way is clear or there is a visible obstacle boundary pt that decreases heuristic distance
- A **boundary following** behavior invoked when heuristic distance increases.
- A value d_{min} which is the shortest distance observed thus far between the sensed boundary of the obstacle and the goal
- A value d_{leave} which is the shortest distance between any point in the currently sensed environment and the goal
- Terminate boundary following behavior when $d_{leave} < d_{min}$

Boundary Following

Move toward the O_i on the followed obstacle in the “chosen” direction



M is the point on the “sensed” obstacle which has the shorted distance to the goal

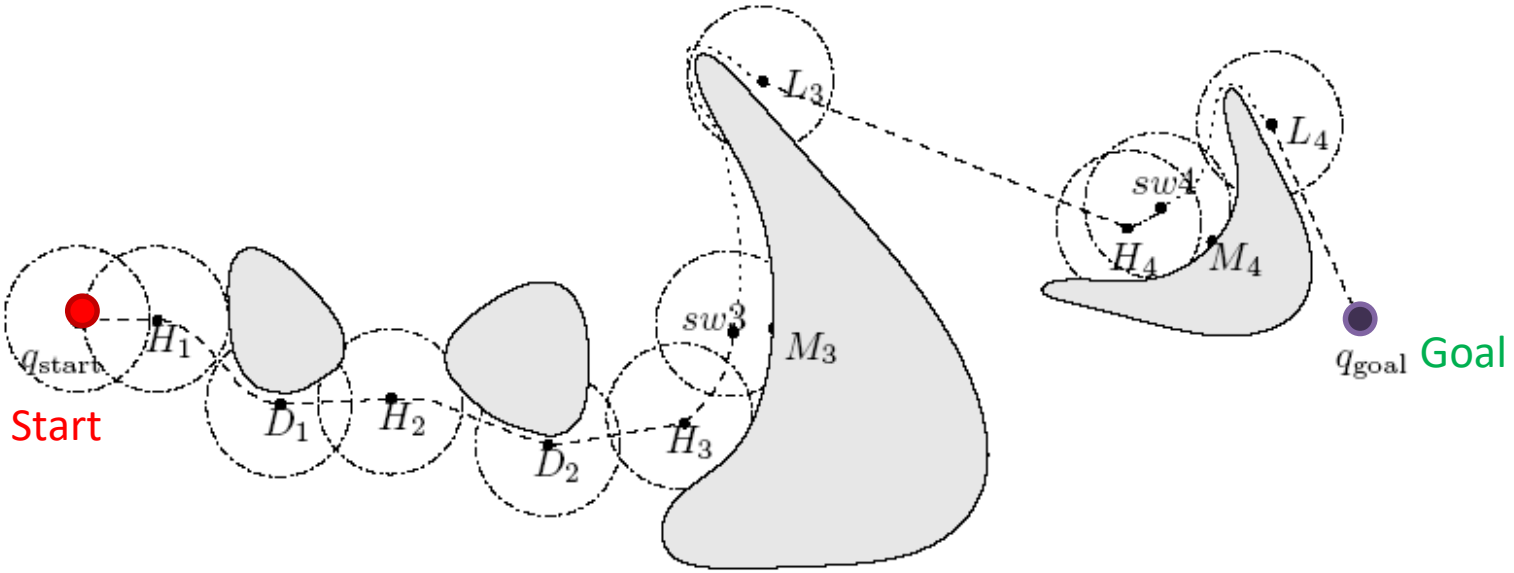
Followed obstacle: the obstacle that we are currently sensing

Blocking obstacle: the obstacle that intersects the segment

Maintain d_{min} and d_{leave}

They start as the same

Tangent Bug Algorithm



H : hit point
M : minimum point
D : depart point
L : leave point

Terminate boundary following behavior when $d_{leave} < d_{min}$

Tangent Bug Algorithm

1) repeat
   a) Compute continuous range segments in view
   b) Move toward $n \in \{T, O_i\}$ that minimizes $h(x,n) = d(x,n) + d(n,q_{goal})$

   until
   a) goal is encountered, or
   b) the value of h(x,n) begins to increase

1) follow boundary continuing in same direction as before repeating
   a) update $\{O_i\}$, d_{leave} and d_{min}

   until
   a) goal is reached
   b) a complete cycle is performed (goal is unreachable)
   c) $d_{leave} < d_{min}$

Note the same general proof reasoning as before applies, although the definition of hit and leave points is a little trickier.

d_{min} and d_{leave}

- d_{min} is the shortest distance, observed thus far, between the sensed boundary of the obstacle and the goal

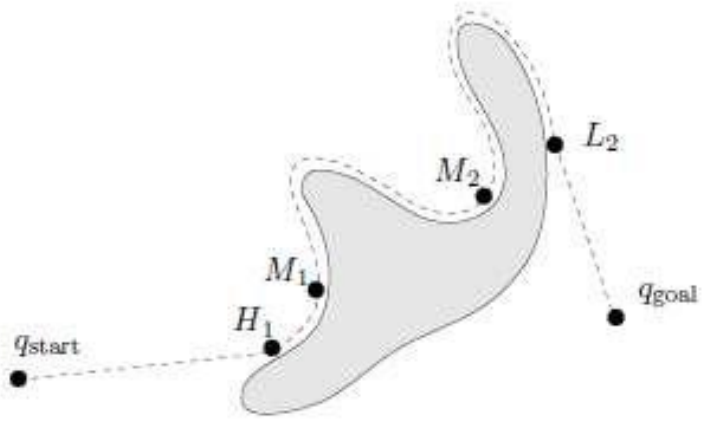
- d_{leave} is the shortest distance between any point in the currently sensed environment and the goal

$$V_R(x) = \{y \in W_{free} \mid d(x,y) < R \text{ and } \lambda x + (1-\lambda) y \in W_{free} \text{ for all } \lambda \in [0,1]\}$$

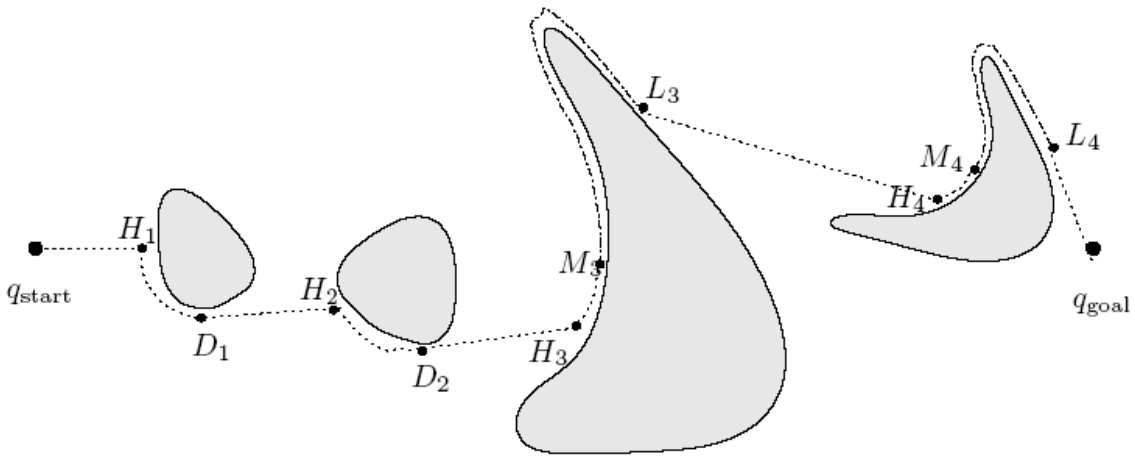
$$d_{leave}(x) = \min_{y \in V(x)} d(goal, y)$$

- Terminate boundary following behavior when $d_{leave} < d_{min}$
- Initialize with $x = q_{start}$ and $d_{leave} = d(q_{start}, q_{goal})$

d_{min} is constantly updated

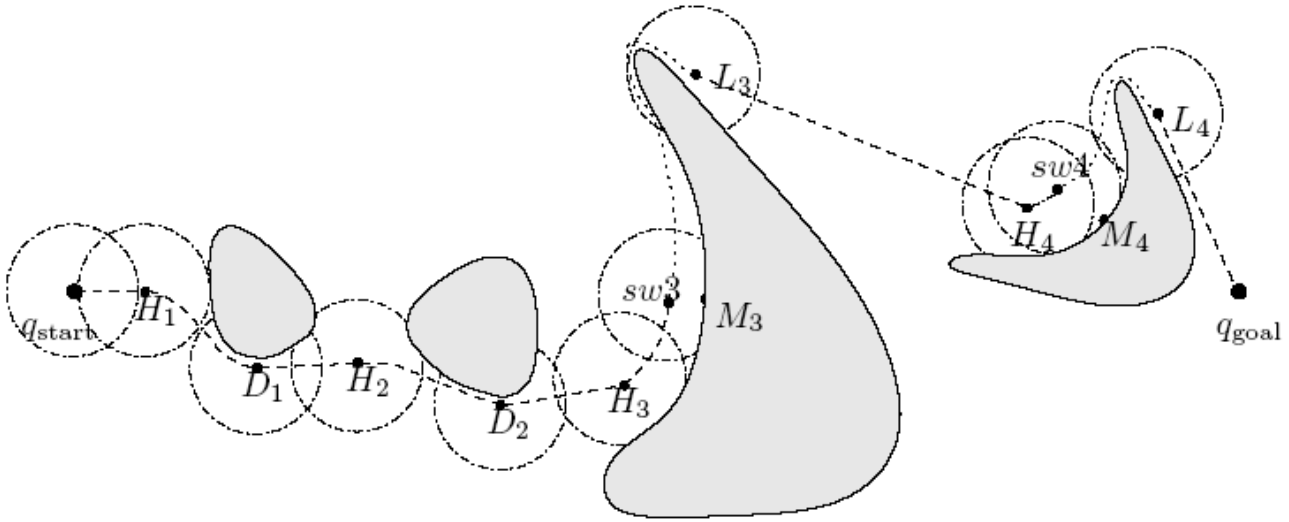


Example: Zero Senor Range

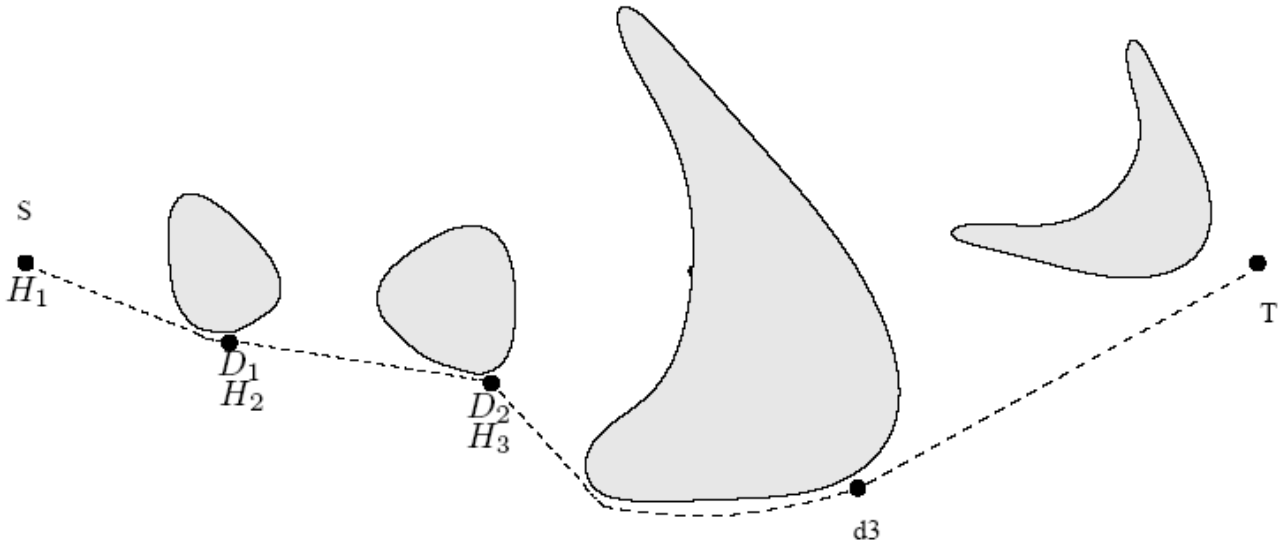


1. Robot moves toward goal until it hits obstacle 1 at H1
2. Pretend there is an infinitely small sensor range and the Oi which minimizes the heuristic is to the right
3. Keep following obstacle until robot can go toward obstacle again
4. Same situation with second obstacle
5. At third obstacle, the robot turned left until it could not increase heuristic
6. d_{min} is distance between M_3 and goal, d_{leave} is distance between robot and goal because sensing distance is zero

Example: Finite Sensor Range

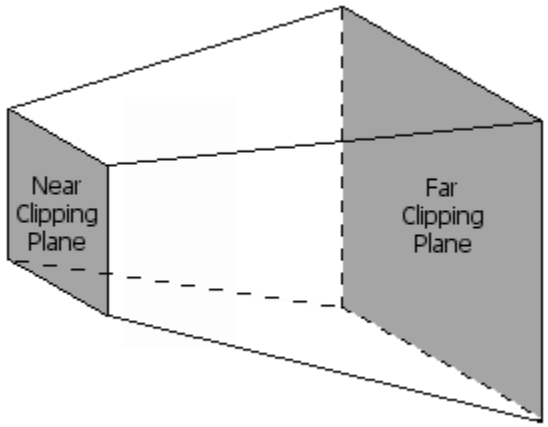


Example: Infinite Sensor Range



Collision Detection

- What is collision detection
- Given two geometric objects, determine if they overlap.
- Typically, at least one of the objects is a set of triangles.
  - Rays/lines
  - Planes
  - Polygons / Polyhedrons
  - Frustums (german: Pyramidenstumpf, Sichtvolumen)



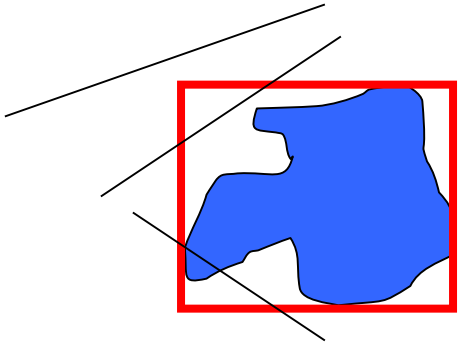
  - Spheres
  - Curved surfaces

When Used

- Often in simulations
  - Objects move – find when they hit something else
- Other examples
  - Ray tracing speedup
  - Culling objects/classifying objects in regions
- Usually, needs to be fast
  - Applied to lots of objects, often in real-time applications

Key idea:

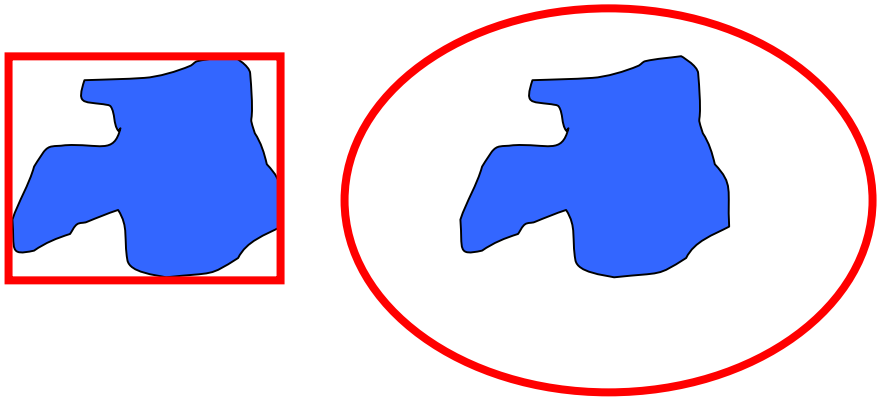
- Surround the object with a (simpler) bounding object (the bounding volume).
- If something does not collide with the bounding volume, it does not collide with the object inside.
- Often, to intersect two objects, first intersect their bounding volumes



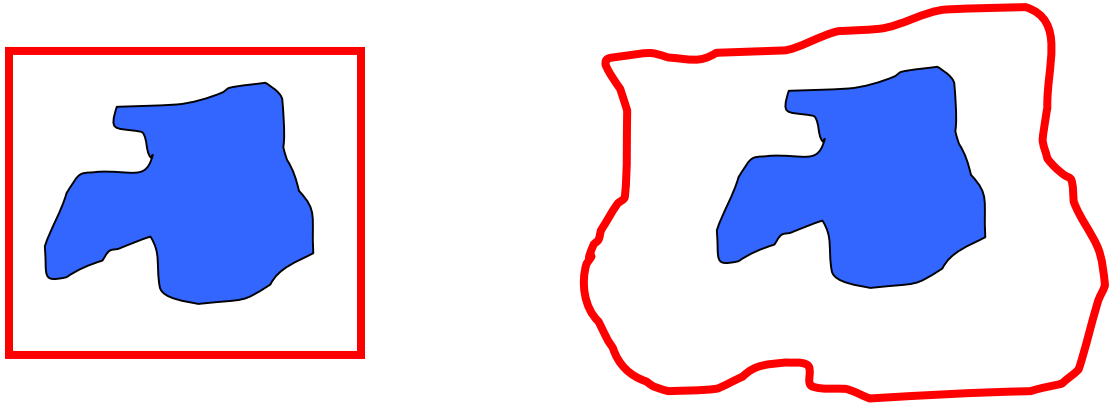
Choosing a Bounding Volume

Lots of choices, each with tradeoffs

- Tighter fitting is better
  - More likely to eliminate “false” intersections



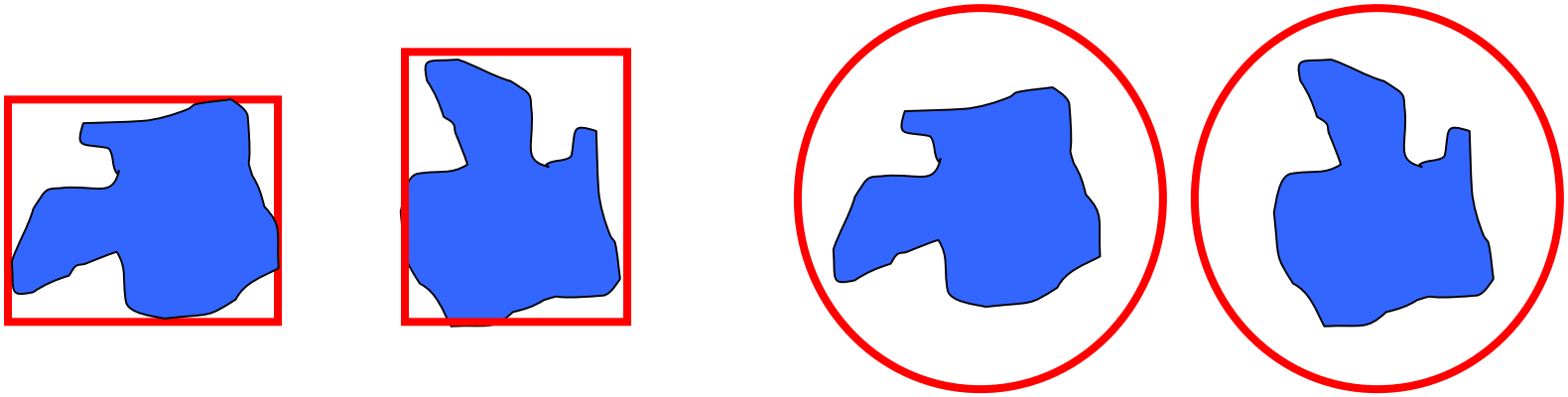
- Simpler shape is better
  - Makes it faster to compute with



Choosing a Bounding Volume

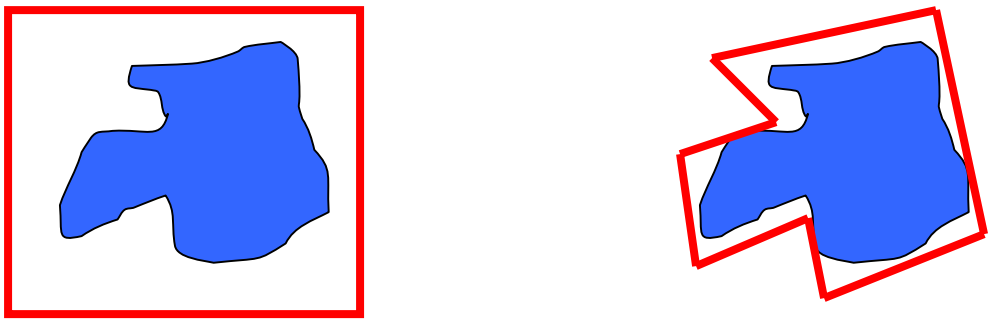
Rotation Invariant is better

- Easier to update as object moves



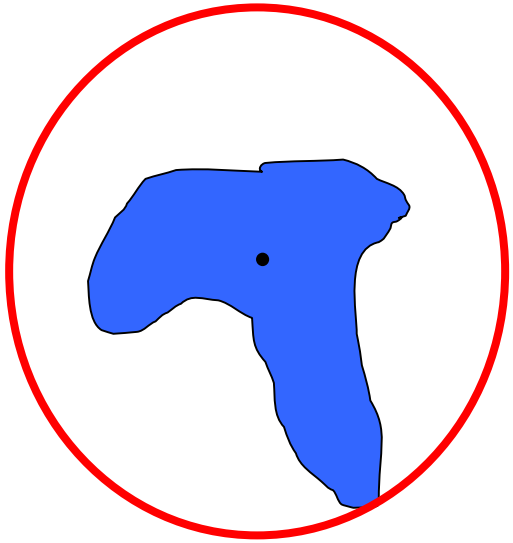
Convex is usually better

- Gives simpler shape, easier computation



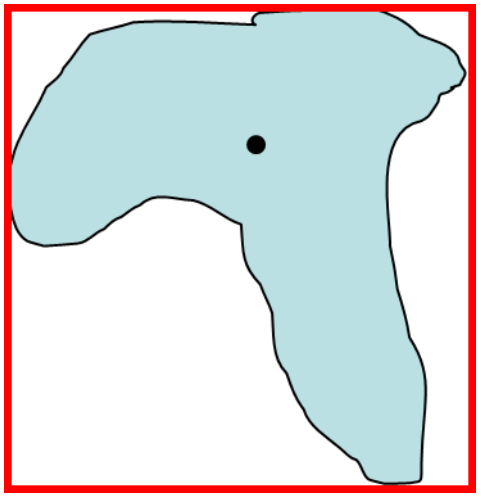
Common Bounding Volumes: Sphere

- Rotationally invariant
  - Usually
- Usually fast to compute with
- Store: center point and radius
  - Center point: object's center of mass
  - Radius: distance of farthest point on object from center of mass.
- Often not very tight fit



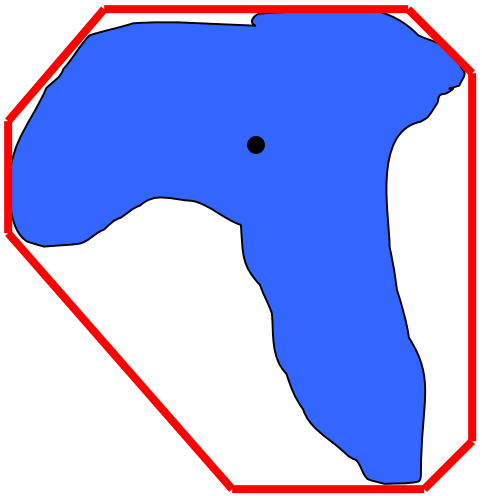
Common Bounding Volumes: Axis Aligned Bounding Box (AABB)

- Very fast to compute with
- Store: max and min along x,y,z axes.
  - Look at all points and record max, min
- Moderately tight fit
- Must update after rotation, unless a loose box that encompasses the bounding sphere



Common Bounding Volumes: k-dops

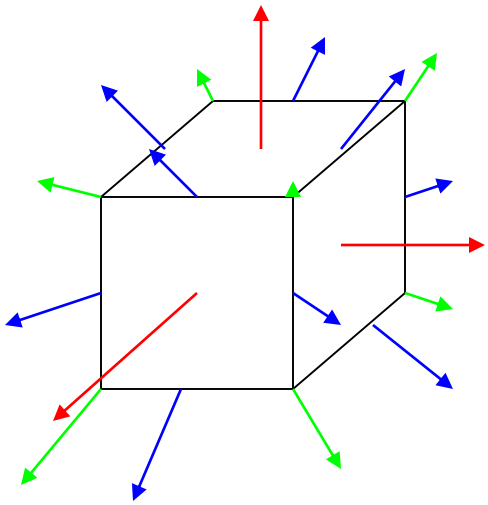
- k-discrete oriented polytopes
- Same idea as AABBs, but use more axes.
- Store: max and min along fixed set of axes.
  - Need to project points onto other axes.
- Tighter fit than AABB, but also a bit more work.



Choosing axes for k-dops

Common axes:

consider axes coming out from center of a cube:



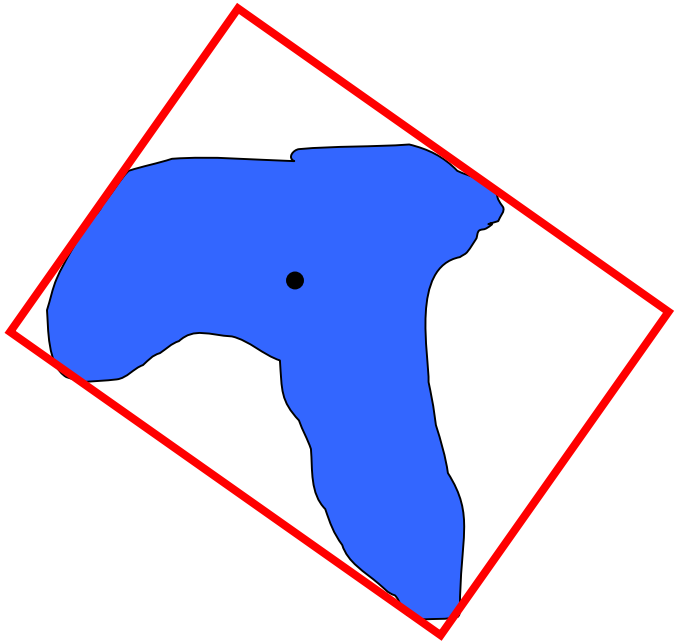
Through faces: 6-dop
- same as AABB

Faces and vertices: 14-dop
Faces and edge centers: 18-dop
Faces, vertices, and edge centers; 26-dop

More than that not really helpful
Empirical results show 14 or 18-dop performs best.

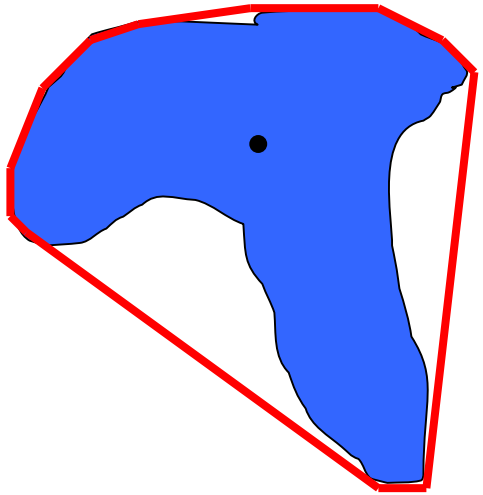
Common Bounding Volumes: Oriented Bounding Box (OBB)

- Store rectangular parallelepiped oriented to best fit the object
- Store:
  - Center
  - Orthonormal set of axes
  - Extent along each axis
- Tight fit, but takes work to get good initial fit
- OBB rotates with object, therefore only rotation of axes is needed for update
- Computation is slower than for AABBs,
  but not as bad as it might seem



Common Bounding Volumes: Convex Hull (CH)

- Very tight fit (tightest convex bounding volume)
- Slow to compute with
- Store: set of polygons forming convex hull
- Can rotate CH along with object.
- Can be efficient for some applications



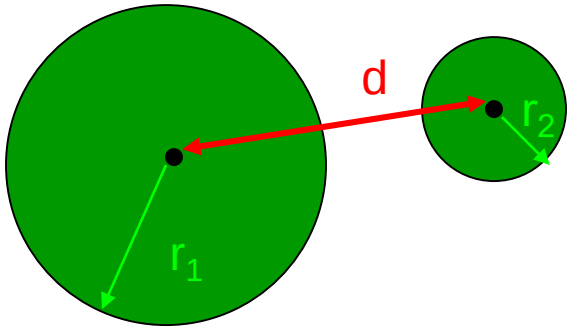
Testing for Collision

- Will depend on type of objects and bounding volumes.
- Specialized algorithms for each:

- Sphere/sphere
- AABB/AABB
- OBB/OBB
- Ray/sphere
- Triangle/Triangle

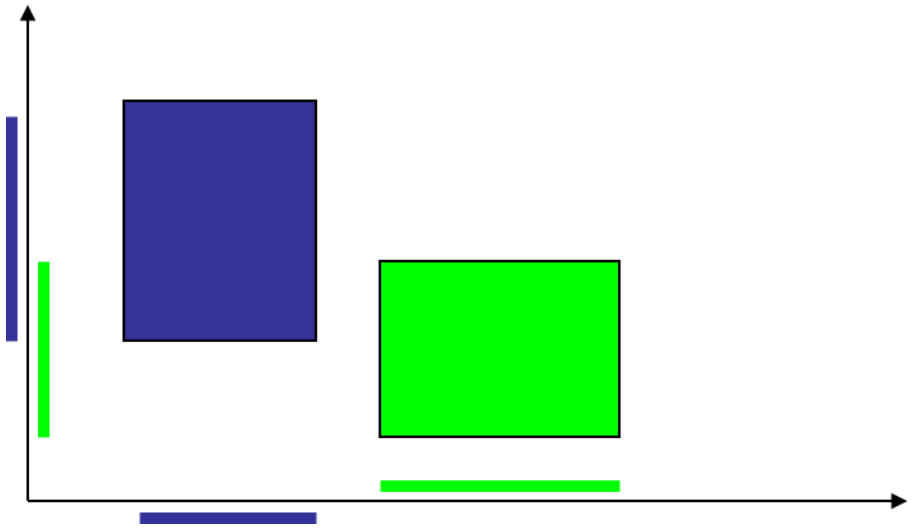
Collision Test Example Sphere-Sphere

- Find distance between centers of spheres
- Compare to sum of sphere radii
  - If distance is less, they collide
- For efficiency, check squared distance vs. square of sum of radii



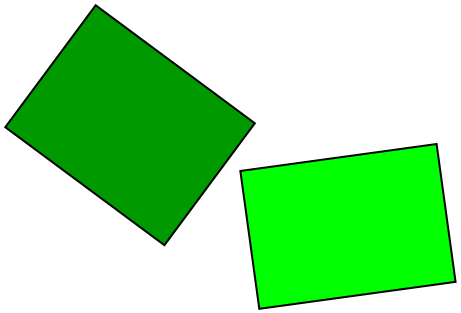
Collision Test Example AABB vs. AABB

- Project AABBs onto axes
  - i.e. look at extents
- If overlapping on all axes, the boxes overlap.
- Same idea for k-dops.

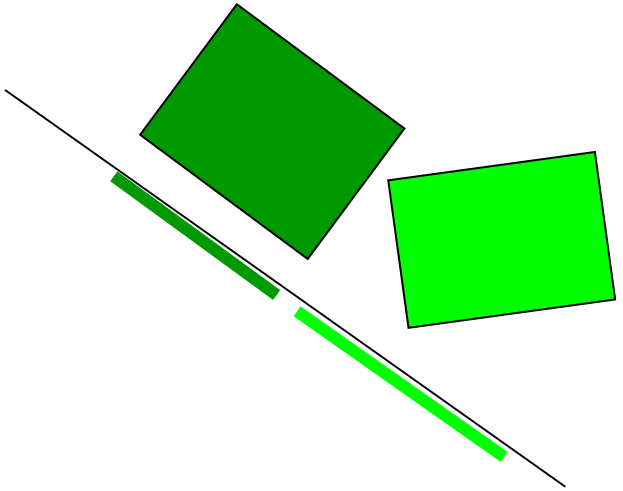


Collision Test Example OBB vs. OBB

- How do we determine if two oriented bounding boxes overlap?

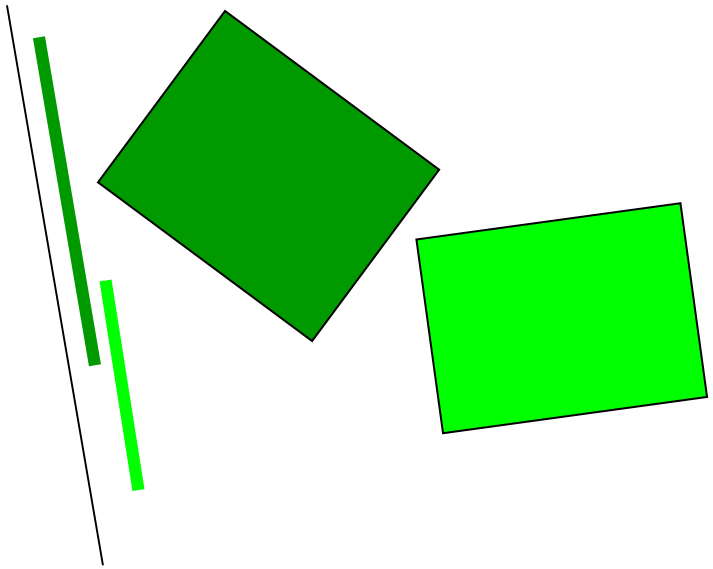


- Two convex shapes do not overlap if and only if there exists an axis such that the projections of the two shapes do not overlap
- **Separating Axis Theorem** (SAT)



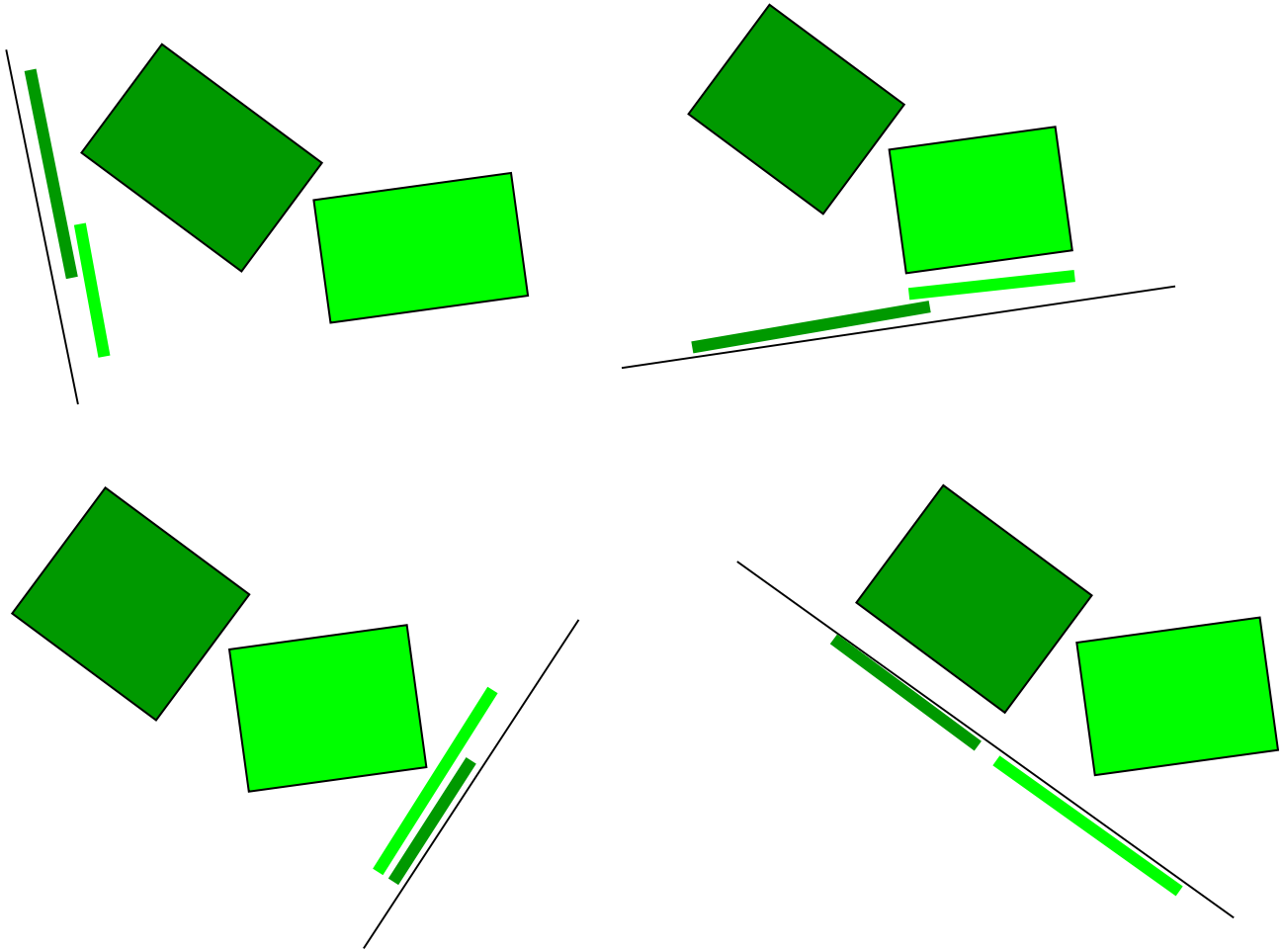
Enumerating Separating Axes

- 2D: check axis aligned with normal of each face
- 3D: check axis aligned with normals of each face and cross product of each pair of edges



Enumerating Separating Axes

- 2D: check axis aligned with normal of each face



Separating Axis Theorem (SAT) in 3D

- Two polytopes A and B are disjoint if and only if there exists a separating axis which is:

- perpendicular to a face from either (for vertex/face contact)

or

- perpendicular to an edge from each (for edge/edge contact)

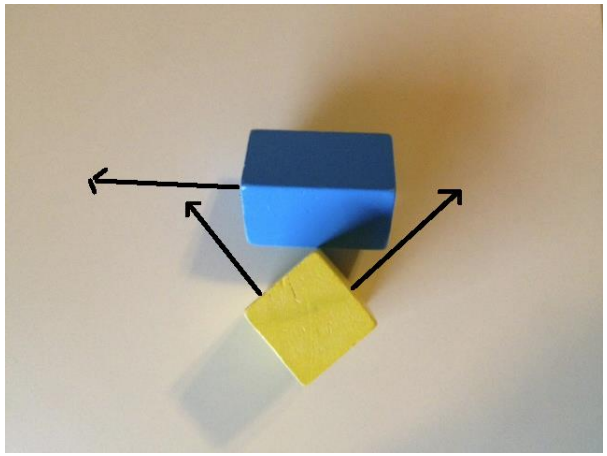
Implications:

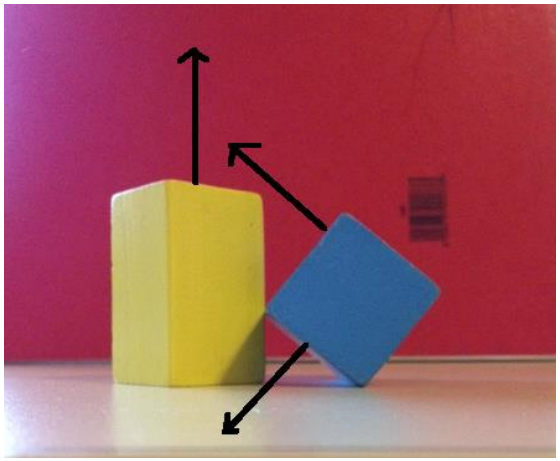
- Given two generic polytopes, each with E edges and F faces, number of candidate axes to test is:

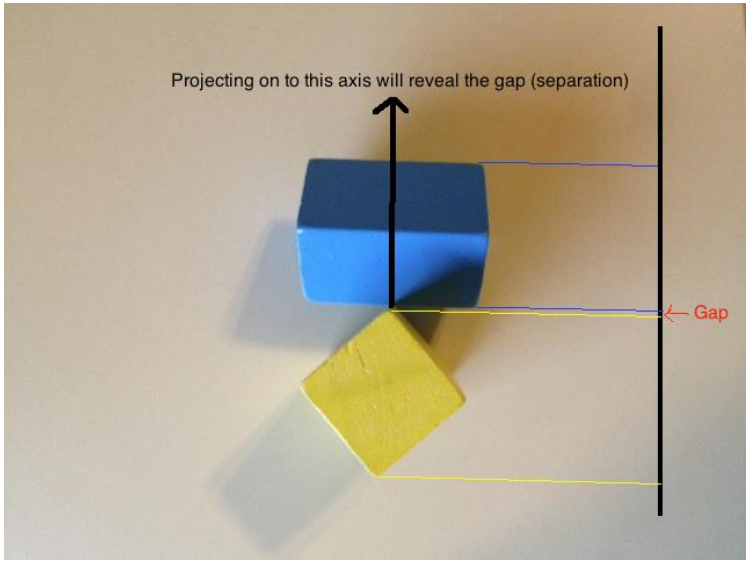
$$2F + E^2$$

- OBBs have only E = 3 distinct edge directions, and only F = 3 distinct face normals. OBBs need at most 15 axis tests.

Separating Axis Theorem (SAT) in 3D





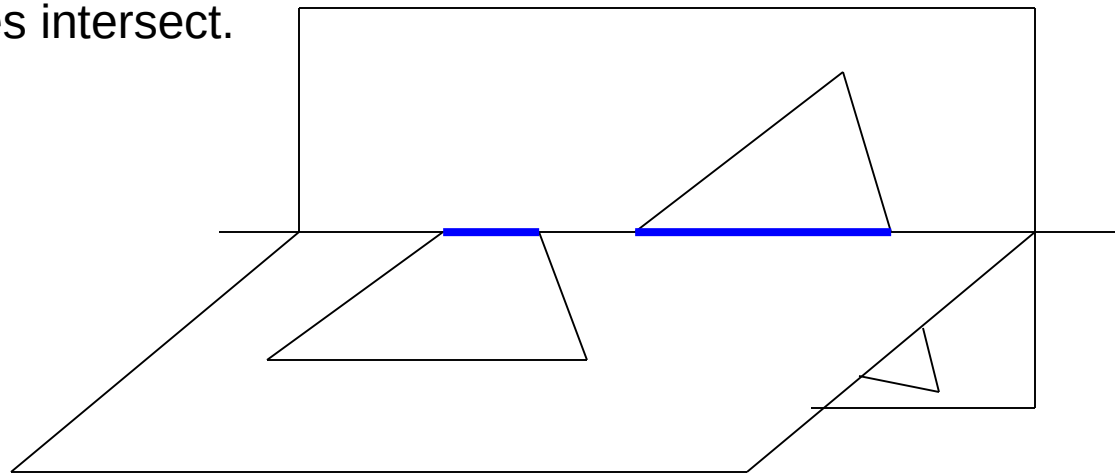


Collision Test Example Triangle-Triangle

- Many collision detection tests eventually reduce to this.
- Two common approaches. Both involve finding the plane a triangle lies in.
  - Cross product of edges to get triangle normal.
  - This is the plane normal [A B C] where plane is Ax+By+Cz+D=0
  - Solve for D by plugging in a triangle vertex

Triangle-Triangle Collision 1

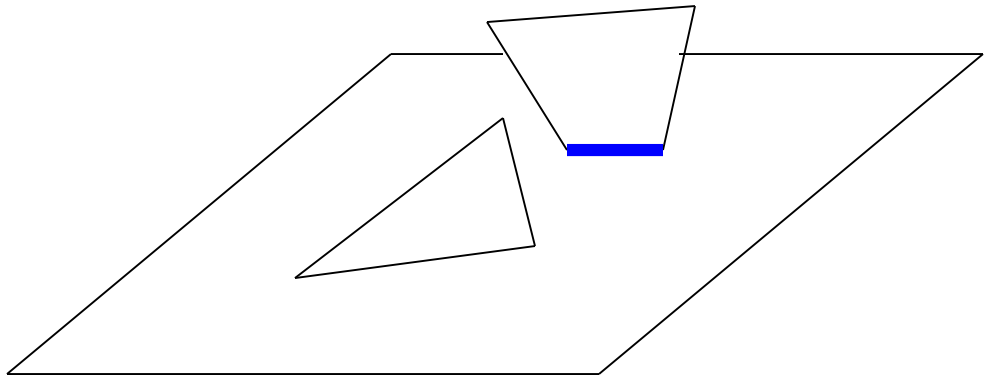
- Find line of intersection between triangle planes.
- Find extents of triangles along this line
- If extents overlap, triangles intersect.



Collision Test Example Triangle-Triangle

Triangle-Triangle Collision 2

- Intersect edges of one triangle with plane of the other triangle.
- 2 edges will intersect – form line segment in plane.
- Test that 2D line segment against triangle.

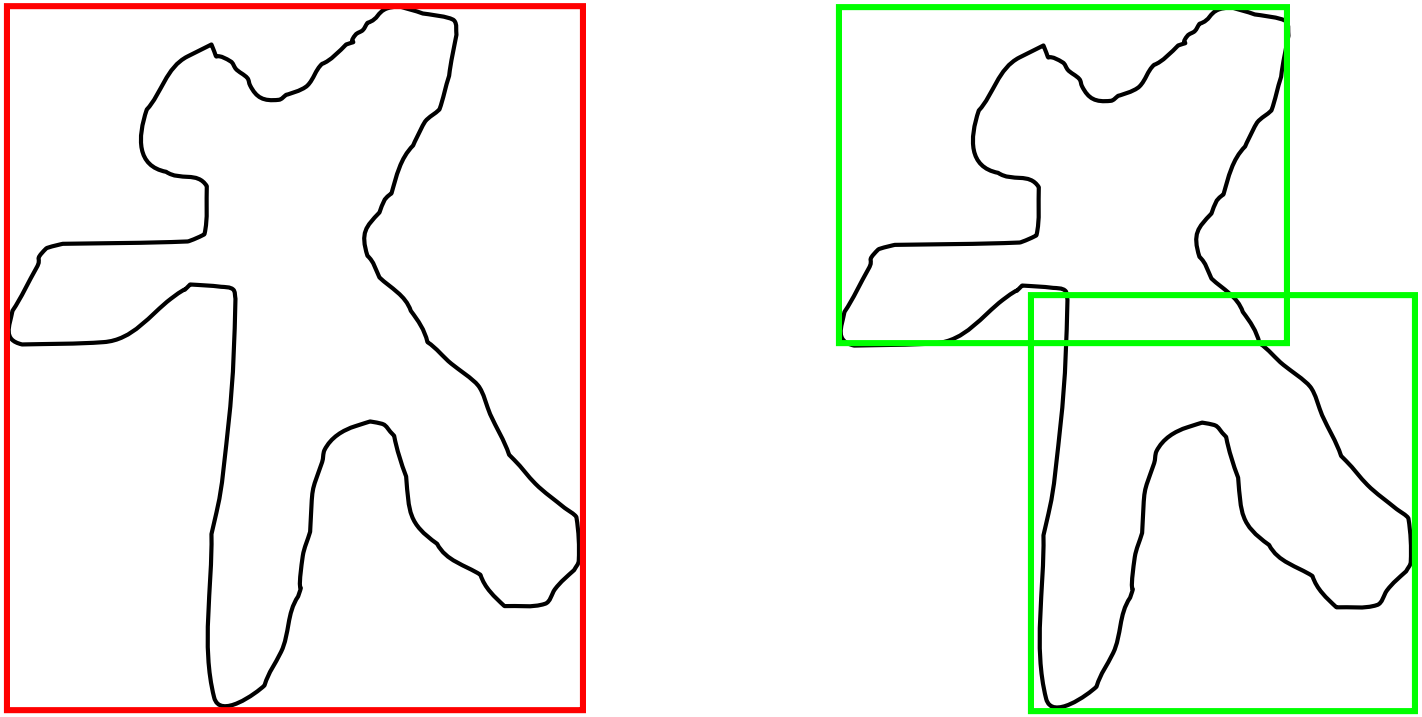


Bounding Volume Hierarchies (BVH)

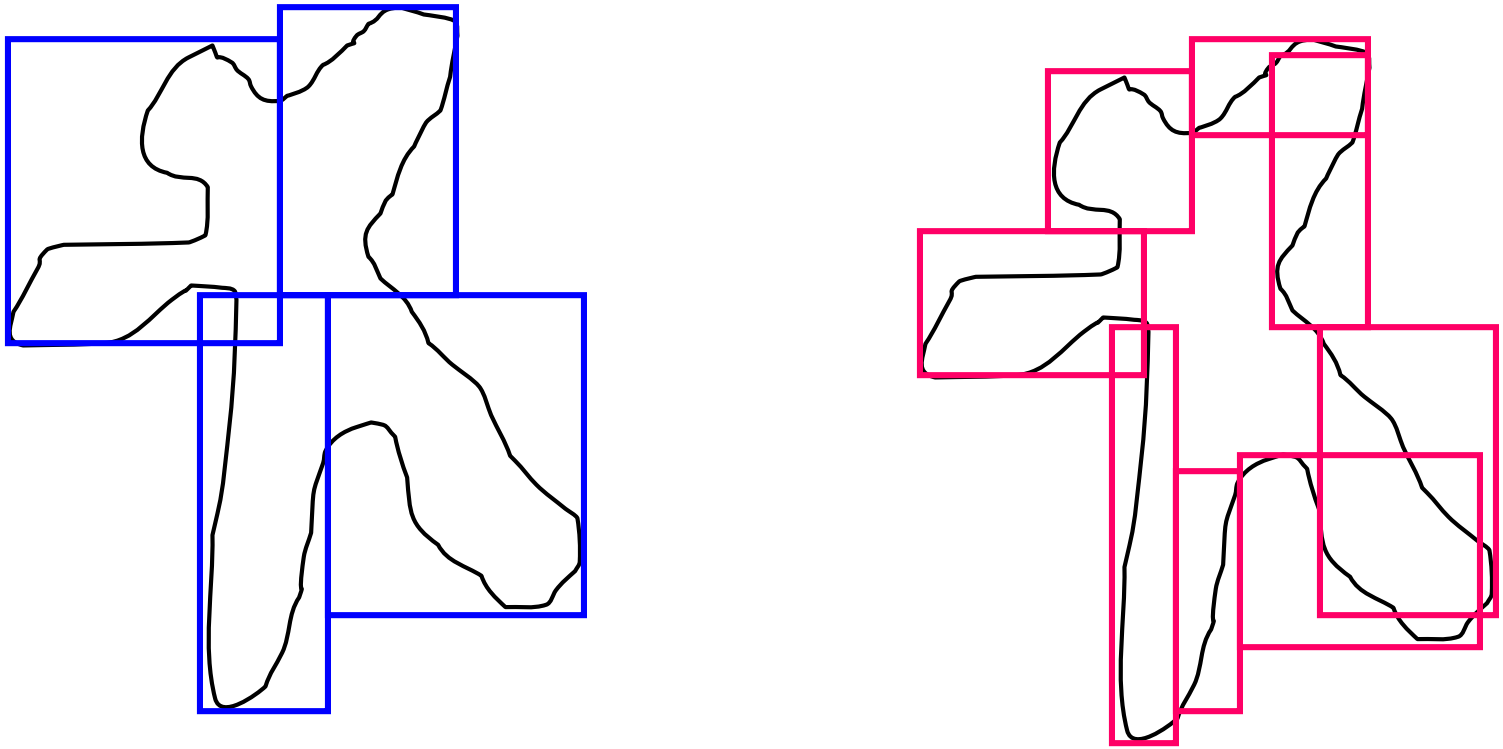
- What happens when the bounding volumes do intersect?
- We must test whether the actual objects underneath intersect.
- For an object made from lots of polygons, this is complicated.
- So, we will use a bounding volume hierarchy

- Highest level of hierarchy – single BV around whole object
- Next level – subdivide the object into two (or maybe more) parts.
  - Each part gets its own BV
- Continue recursively until only one triangle remains

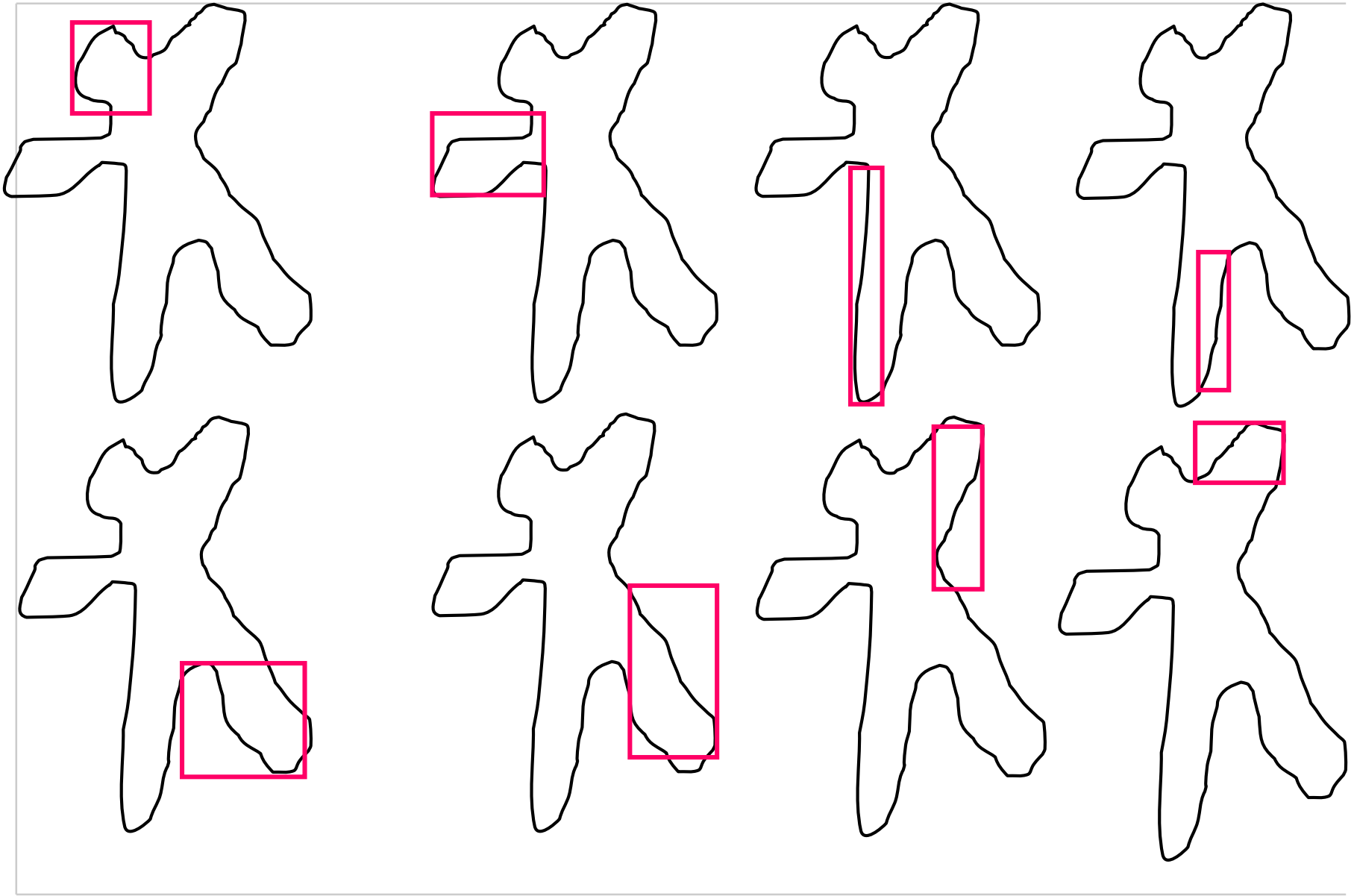
Bounding Volume Hierarchy Example



Bounding Volume Hierarchy Example

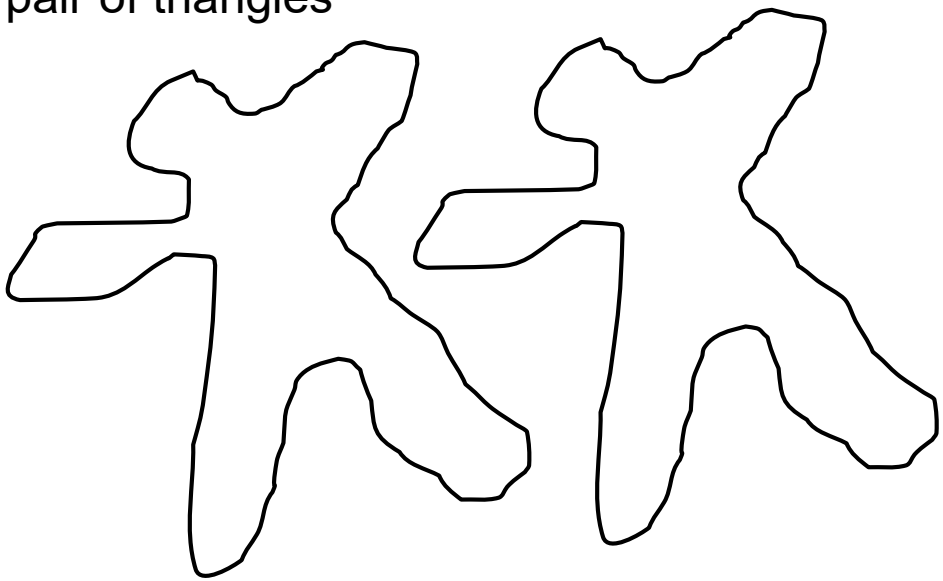


Bounding Volume Hierarchy Example

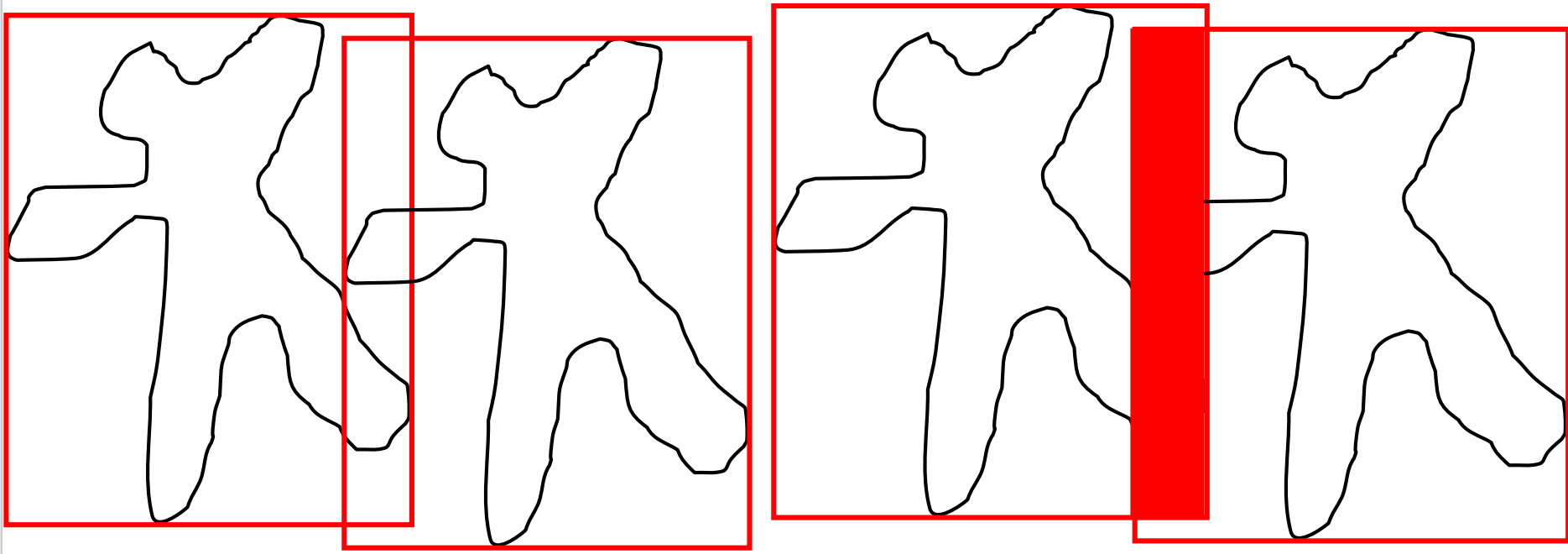


Intersecting Bounding Volume Hierarcies

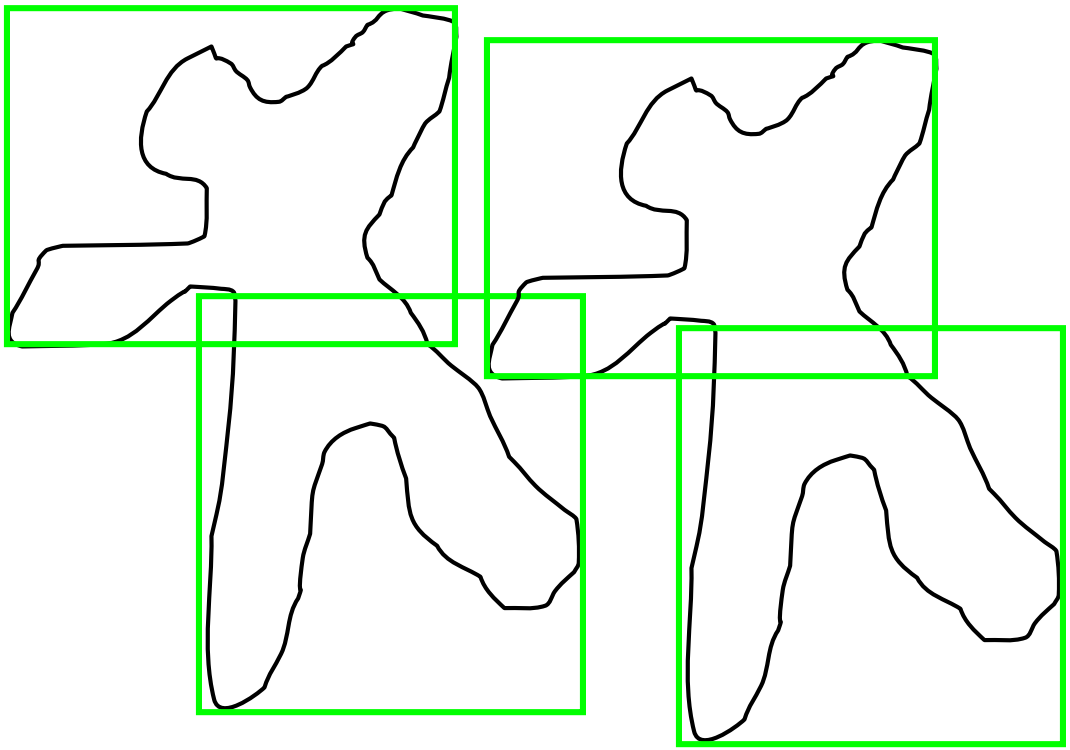
- For object-object collision detection
- Keep a queue of potentially intersecting BVs
  - Initialize with main BV for each object
- Repeatedly pull next potential pair off queue and test for intersection.
  - If that pair intersects, put pairs of children into queue.
  - If no child for both BVs, test triangles inside
- Stop when we either run out of pairs (thus no intersection) or we find an intersecting pair of triangles
- Example:



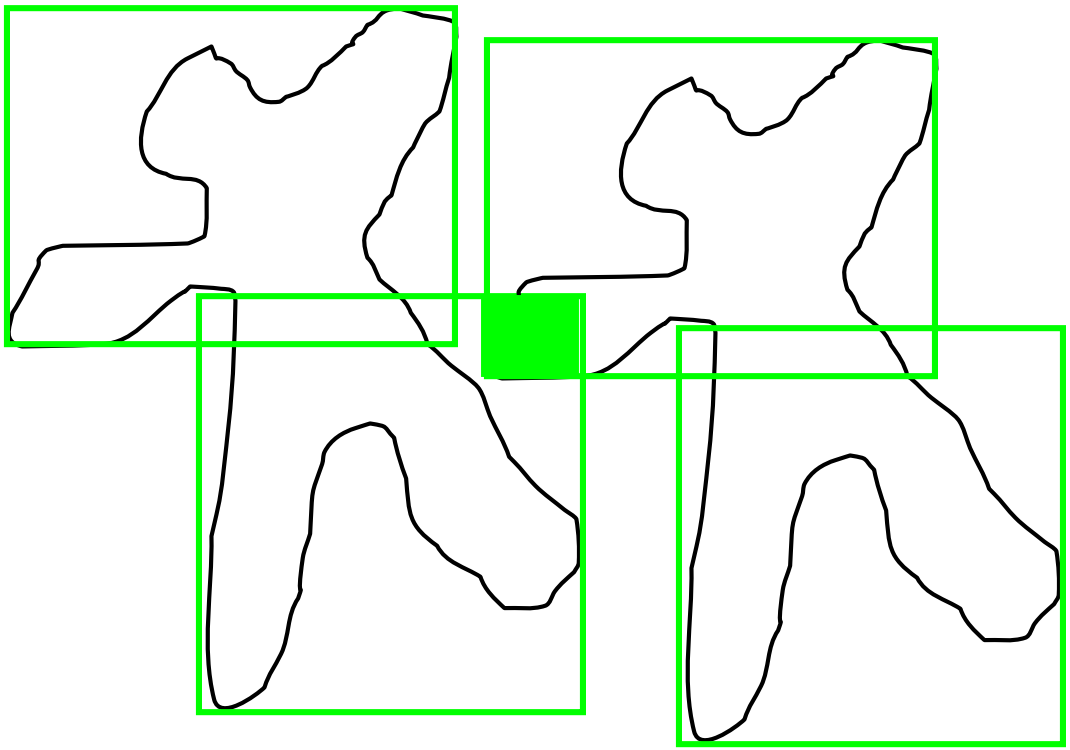
BVH Collision Test example



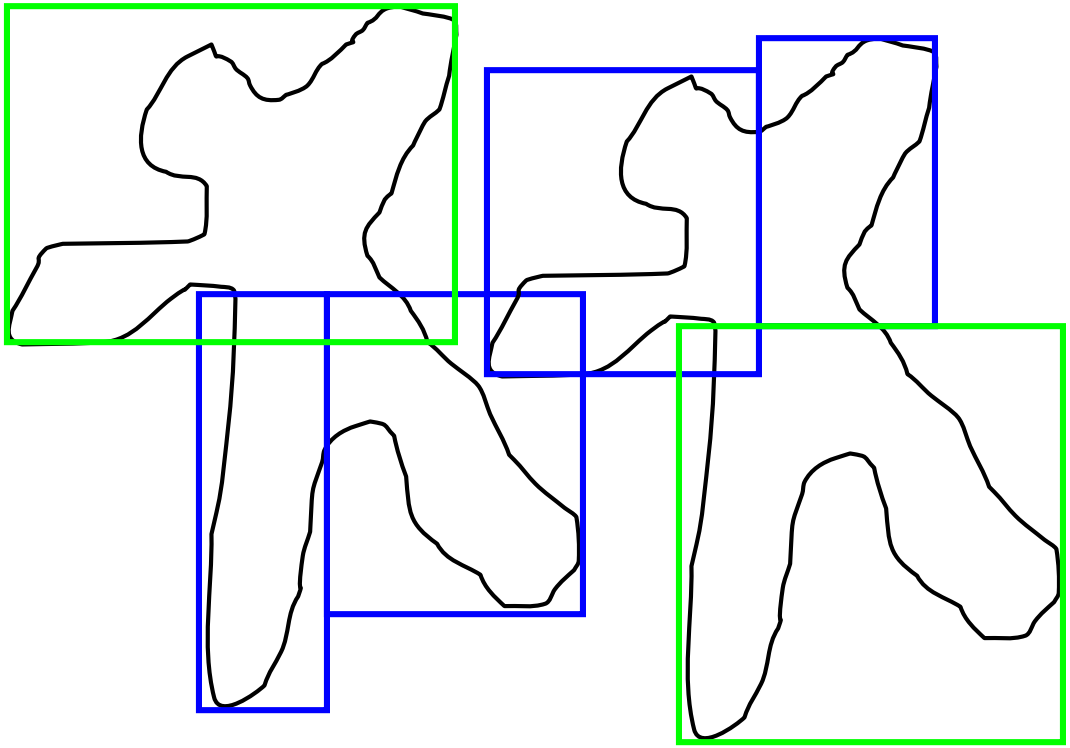
BVH Collision Test example



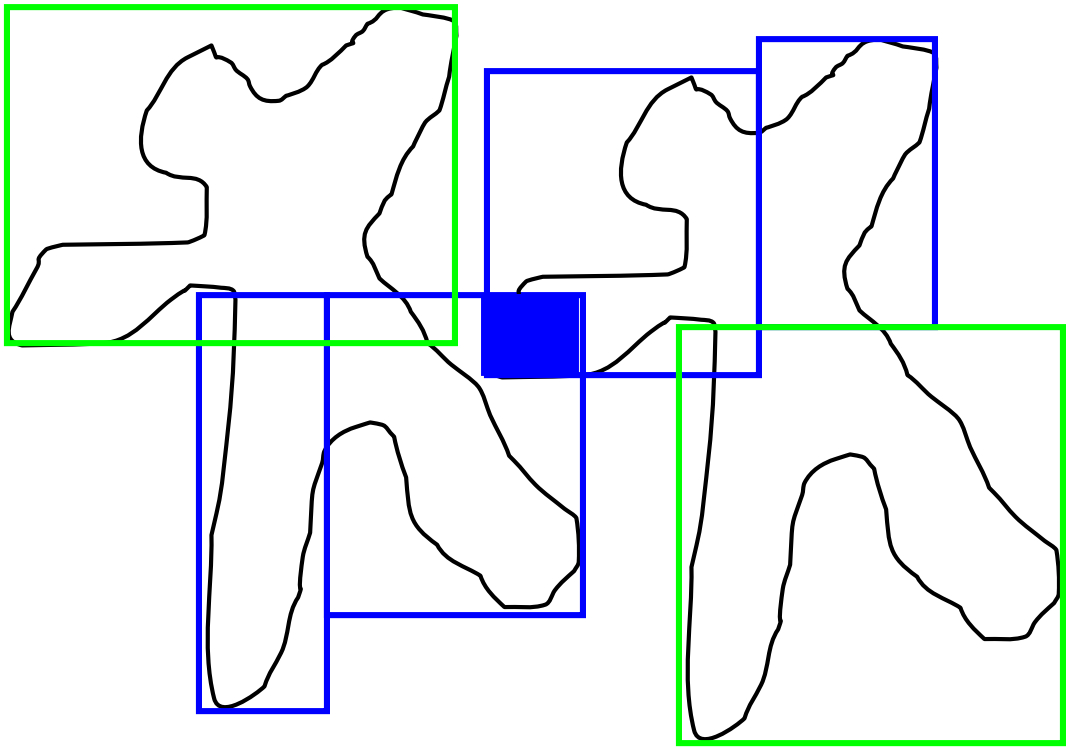
BVH Collision Test example



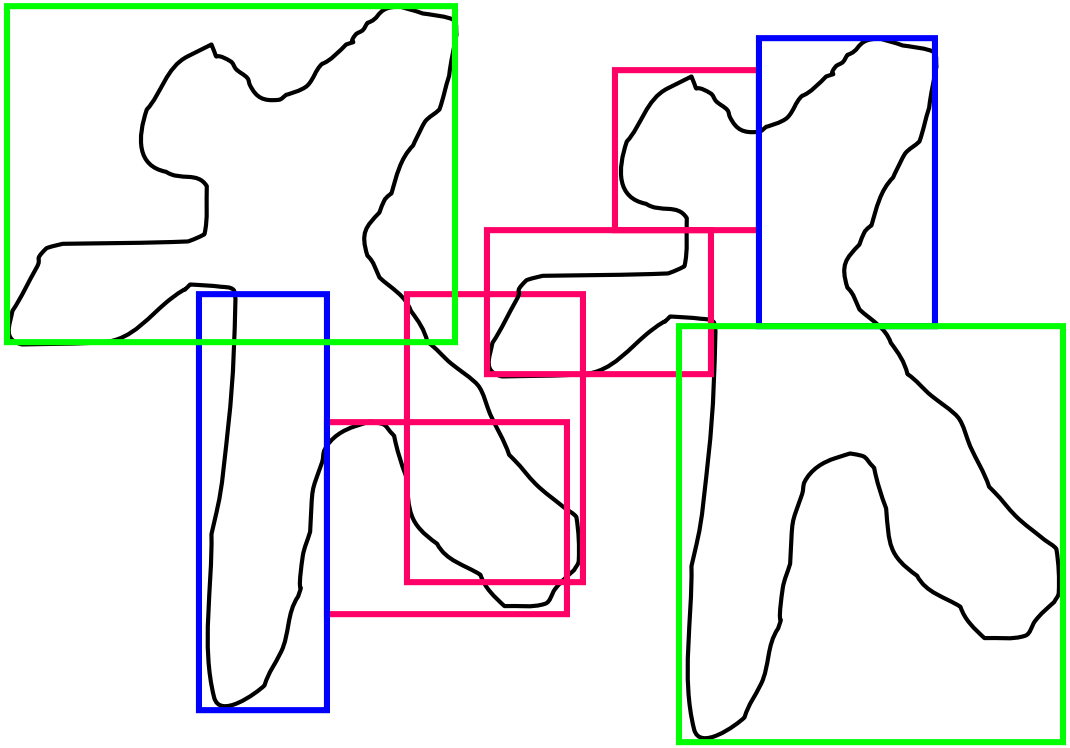
BVH Collision Test example



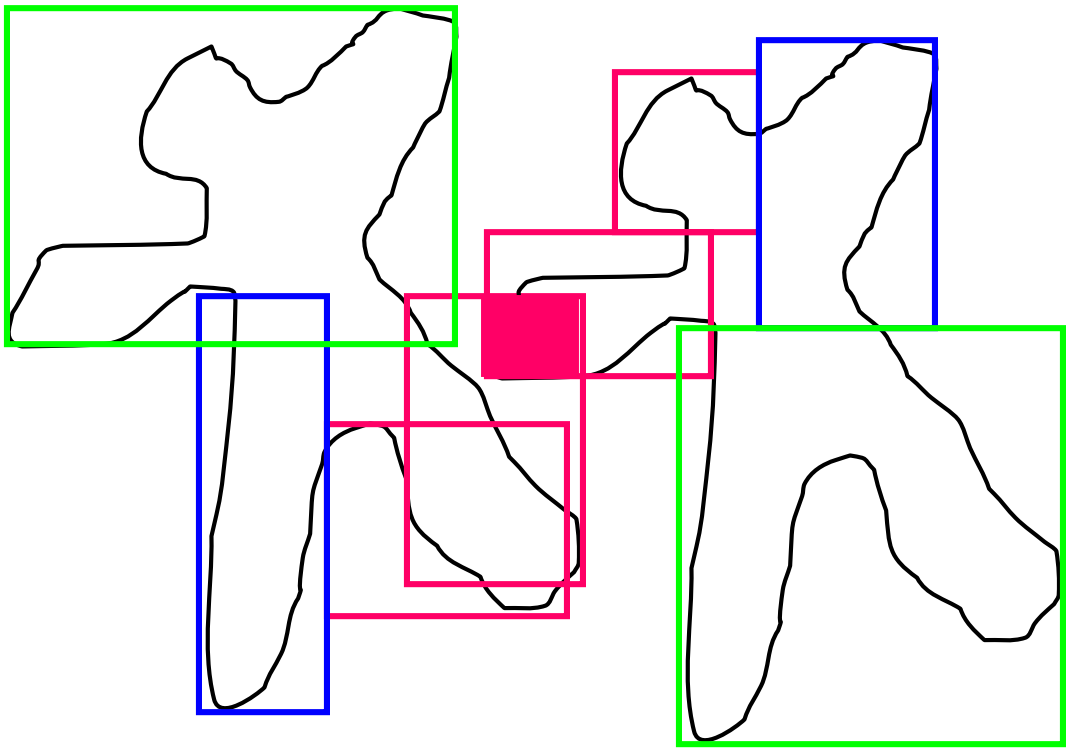
BVH Collision Test example



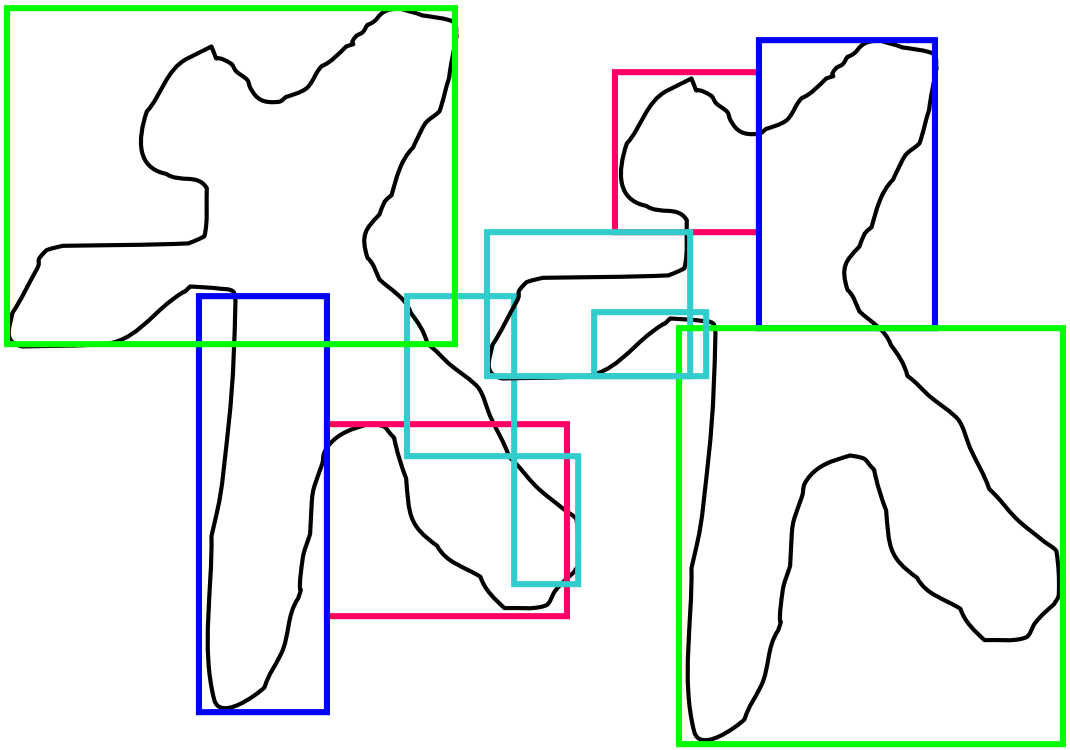
BVH Collision Test example



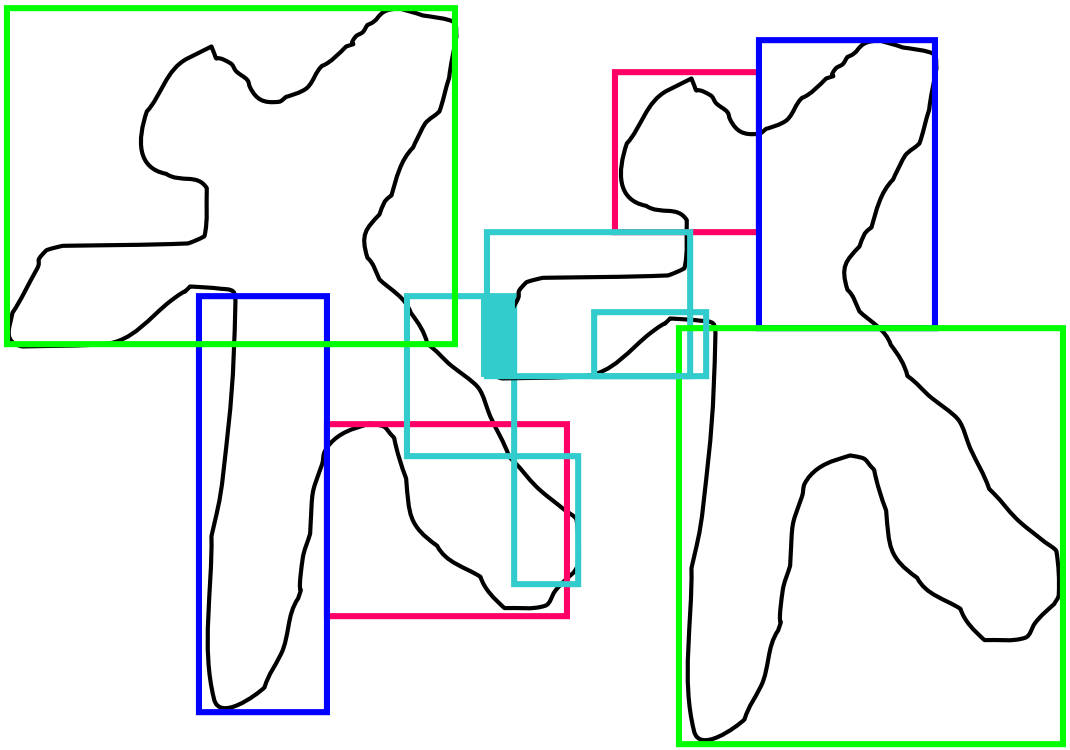
BVH Collision Test example



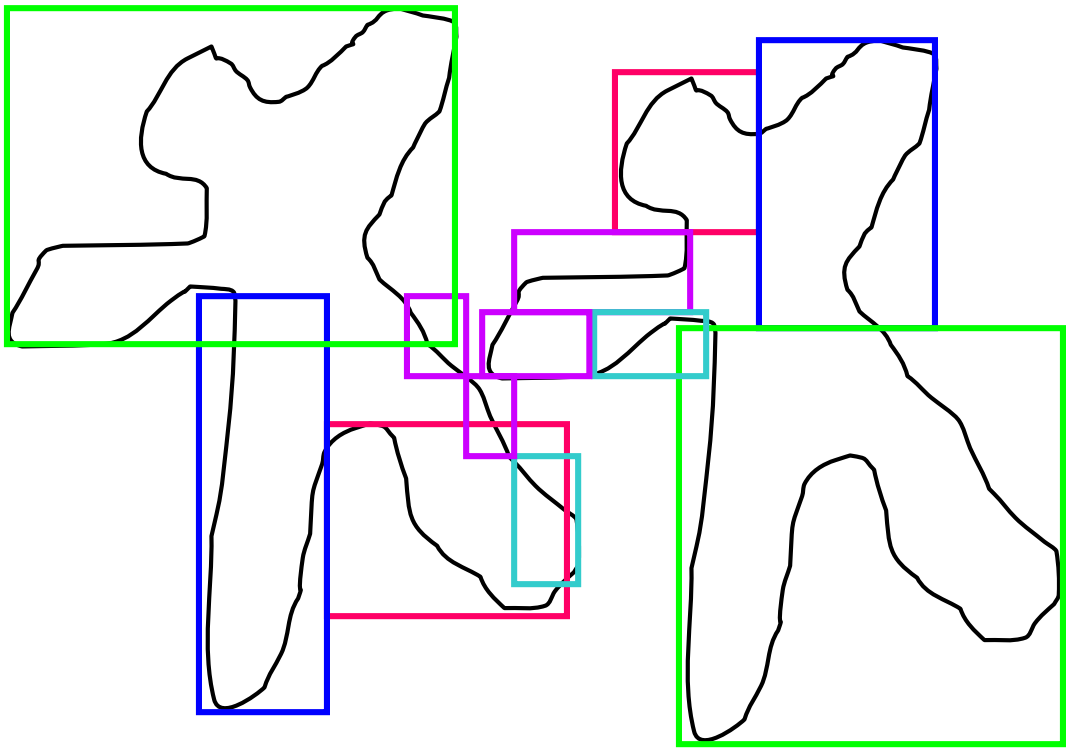
BVH Collision Test example



BVH Collision Test example



BVH Collision Test example



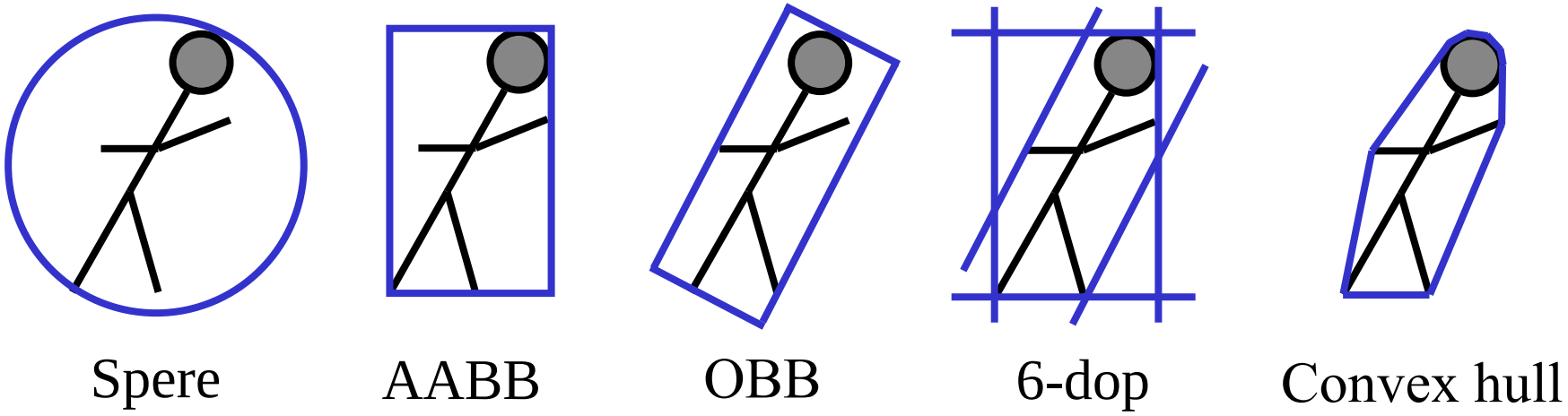
Broad Phase vs. Narrow Phase

- What we have talked about so far is the “narrow phase” of collision detection.
  - Testing whether two particular objects collide
- The “broad phase” assumes we have a number of objects, and we want to find out all pairs that collide.
- Testing every pair is inefficient

Broad Phase

- Form an AABB for each object
- Pick an axis
  - Sort objects along that axis
  - Find overlapping pairs along that axis
  - For overlapping pairs, check along other axes.
- Limits the number of object/object tests
- Overlapping pairs then sent to narrow phase

Summary of Bounding Volumes



increasing complexity & better approximation

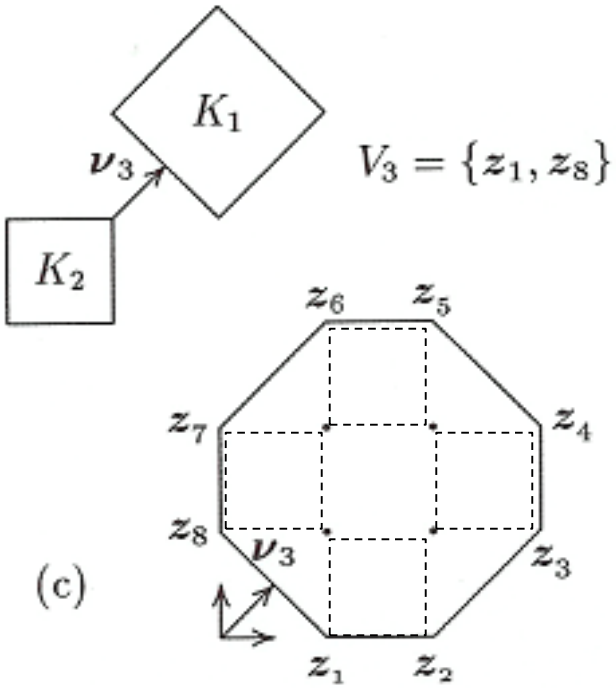
lower costs for collision tests

Collision Detection – Distance calculation

- Calculation of the minimal distance between two geometries

Assumptions:

- Convex geometries, here convex polyhedrons
- Algorithm of Gilbert, Johnson and Keerthi (GJK)
- Based on the Minkowski Difference of two convex geometries (here: polyhedrons



Minkowski Difference:

$$K_1 - K_2 = \{x_1 - x_2 : x_1 \in K_1, x_2 \in K_2\}$$

iterative calculation of closest point v_3

Collision Detection – Distance calculation

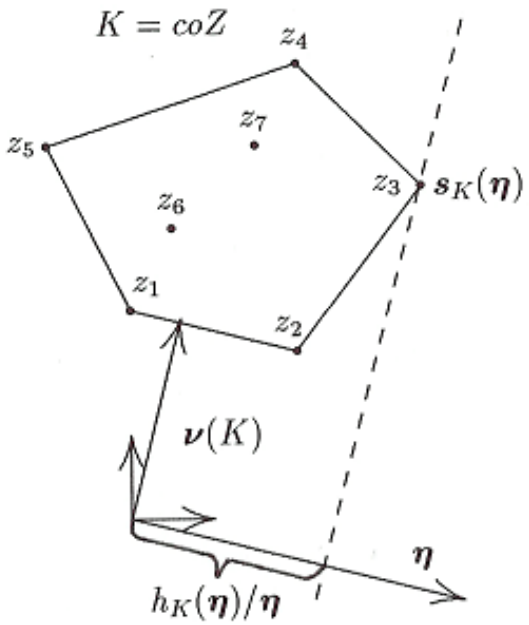
Helper Functions:

- maximal scalar product of points from K in direction η:

$$h_K(\eta) = \max\{x^T \eta : x \in K\}$$

- Point for which the scalar product is maximal:

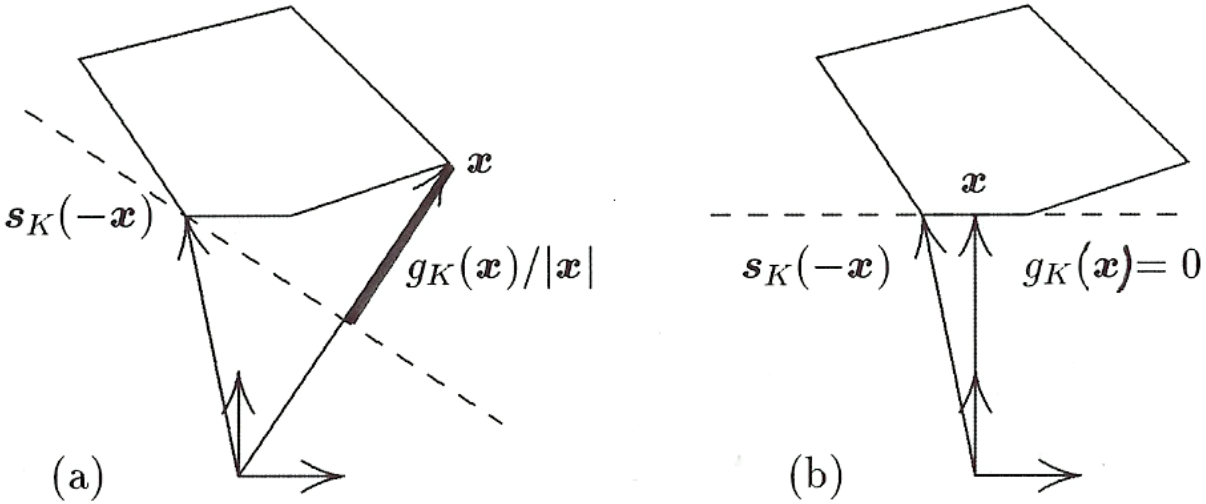
$$h_K(\eta) = s_K(\eta)^T \eta$$



Collision Detection – Distance calculation

Error function $g_K(x)$ is a measure, how far the current candidate is away from the closest point:

$$g_K(x) = |x|^2 + h_K(-x) = |x|^2 + s_K(-x)^T x, \quad x \in K$$



Collision Detection – GJK Algorithm

Step 1: Initialization: choose subset of vertices $V_1 = \{v_1,..,v_v\}$, k=1

Step 2: Iterative calculation of distance between subset co V_k to origin
$v_k = v(\text{co } V_k)$

Step 3: If $g_K(v_k) = 0$, set $v(K) = v_k$ and terminate algorithm (termination criterion)

Step 4: Modification of subset:
$V_{k+1} = \widehat{V_k} \cup \{s_K(-v_k)\}$, where $\widehat{V_k} \subset V_k$ consists of all vertices from V_k, whose convex combination builds the current v_k, so $v_k \in co\widehat{V_k}$

k = k+1 and proceed with step 2

Kollisionserkennung - abstandsbasiert

Das Verfahren:

Schritt 1: Initialisierung durch Auswahl einer Teilmenge der Ecken

$V_1 = \{v_1,..,v_v\}$, k=1

Schritt 2: Iterative Abstandsberechnung der Teilmenge co V_k zum Ursprung

$v_k = v(\text{co } V_k)$

Schritt 3: falls $g_k(v_k) = 0$, setze $v(K) = v_k$ und beende Algorithmus (Abbruchkriterium)

Schritt 4: Modifikation der Teilmenge

$$V_{k+1} = \hat{V}_k \cup \{s_K(-v_k)\}, \text{ wobei } \hat{V}_k \subset V_k \text{ diejenigen}$$

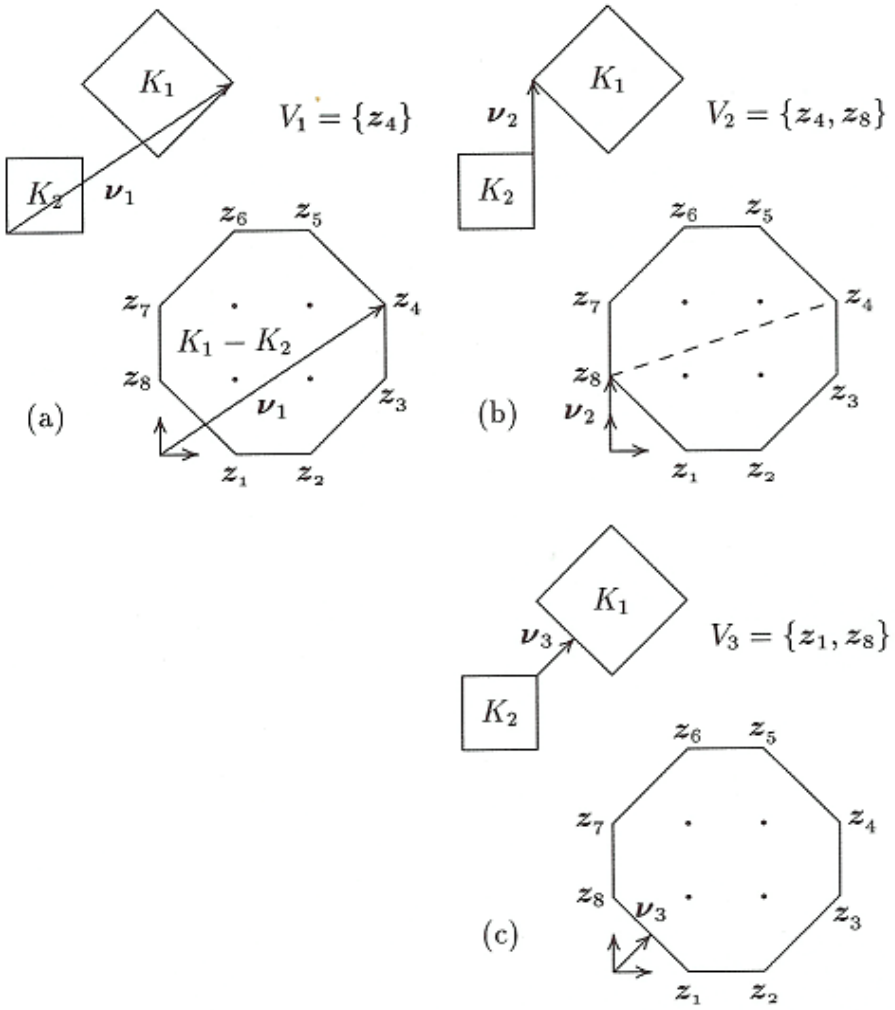
$$\text{Eckpunkte aus } V_k \text{ beinhaltet, deren Konvexkombination}$$

$$\text{gerade das aktuelle } v_k \text{ bildet, } v_k \in co\hat{V}_k;$$

$$k = k+1, \text{ weiter mit Schritt } 2$$

Collision Detection – GJK Algorithm Example

Example: Initialization k=1, $v_1=z_4$ so $V_1 = \{z_4\}$ (Step 1)



$$k=1:\ g_K(v_1)>0 \quad \hat{V}_1=\{z_4\}$$
$$V_2=\hat{V}_1\cup\{z_8\}=\{z_4,z_8\}$$

$$k=2:\ g_K(v_2)>0 \quad \hat{V}_2=\{z_8\}$$
$$V_3=\hat{V}_2\cup\{z_1\}=\{z_1,z_8\}$$

$$k=3:\ g_K(v_3)=0 \ \Rightarrow\ v(K)=v_3$$