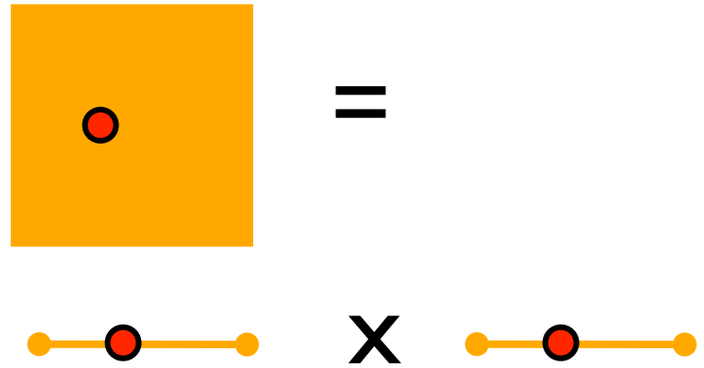
Primitives for probabilistic motion planning

- PRM, EST, and RRT
  - Sample a configuration
  - `CLEAR(q)`
  - `LINK(q,q')`

Sample positions

- Sample a unit interval
  Choose (uniformly) at random $x \in [0,1)$

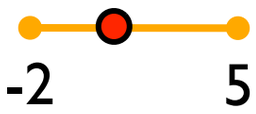
- Sample a unit square
  Sample $x \in [0,1)$ x $[0,1)$



- Sample a unit cube
  Sample $x \in [0,1)$ x $[0,1)$ x $[0,1)$

Q&A

- How do we sample the following?

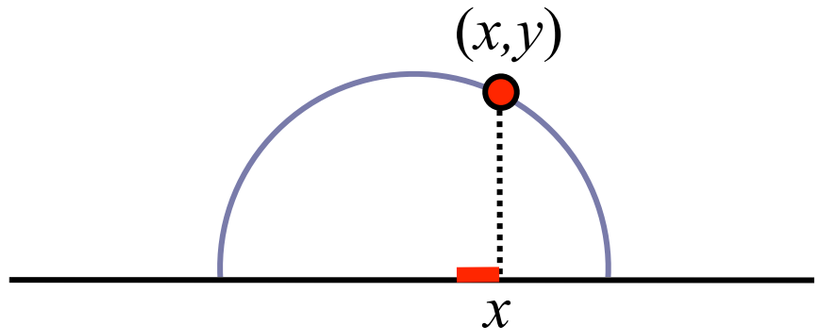


- Scale and shift $x \rightarrow 7x - 2$

"Scale" and "shift". In general, we think of a suitable mapping.

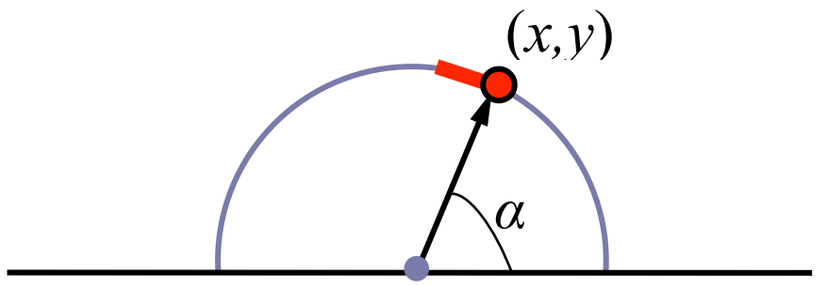
Sample 2-D orientations

- Sample $x \in [-1,1)$ and set $y = \sqrt{1 - x^2}$.
- Intervals of same widths are sampled with equal probabilities.



Sample 2-D orientations

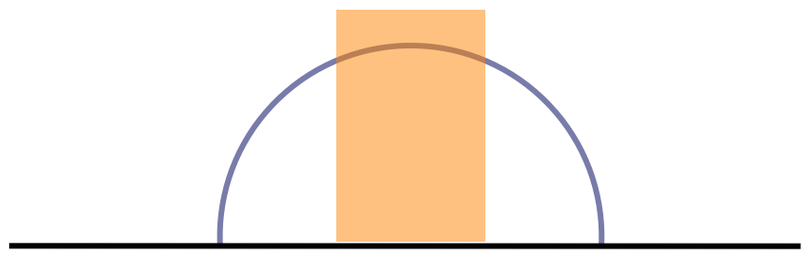
- Sample $\alpha \in [0, \pi)$, and set $x = \cos \alpha$ and $y = \sin \alpha$.



- Arcs spanned by same angles are sampled with equal probabilities.

Comparison

- Each method generates samples uniformly distributed over its own sample space.



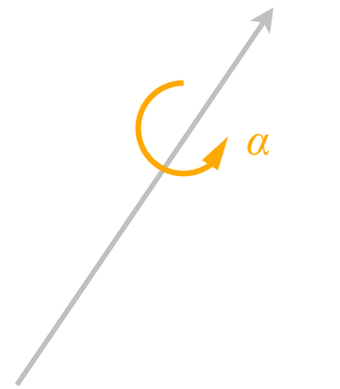
- The two sample spaces are different. For sampling orientations, the second method is usually more suitable.

- A suitable notion of uniform random sampling depends on the task at hand and not on the mathematics.

Sample 3-D orientations

- Sample a unit quaternion
  (n_x, n_y, n_z, α) with $n_x^2 + n_y^2 + n_z^2 = 1$.
- Sample $\mathbf{n}$ and α separately.
- Sample $\alpha \in [0, 2\pi)$.

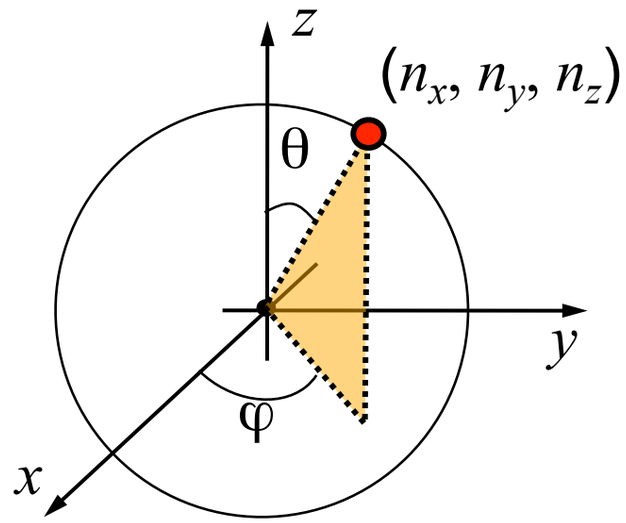
$\mathbf{n} = (n_x, n_y, n_z)$



Sample a point on the unit sphere

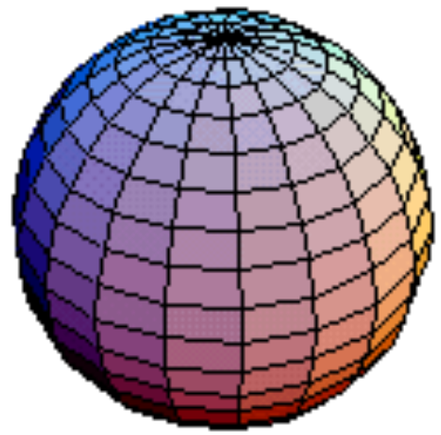
- Longitude $\varphi \in [0, 2\pi)$ and latitude $\theta \in [0, \pi)$.

$$n_x = \sin\theta\cos\varphi$$
$$n_y = \sin\theta\sin\varphi$$
$$n_z = \cos\theta$$



Q&A

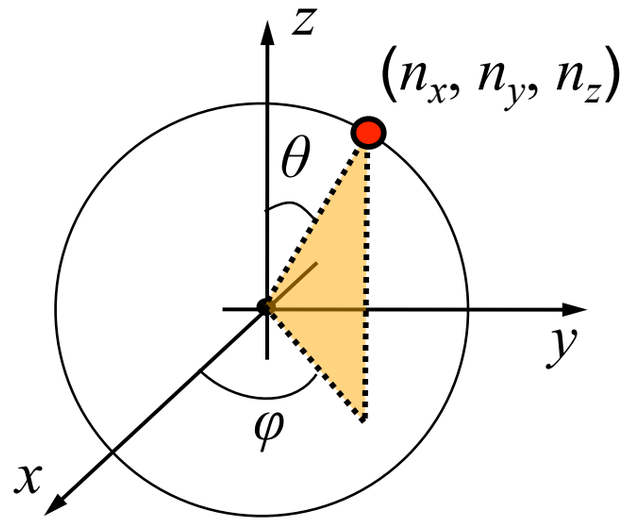
- Choose $\varphi \in [0, 2\pi)$ and $\theta \in [0, \pi)$ uniformly at random.
- Is this a suitable notion of uniform sampling over a sphere?



Choose a point uniformly at random on the unit sphere

- Sample $U_1 \in [-1, 1)$ and $U_2 \in [0, 2\pi)$, and set

$$
\begin{aligned}
n_z &= U_1 \\
n_x &= \sqrt{1 - U_1^2} \cos U_2 \\
n_y &= \sqrt{1 - U_1^2} \sin U_2
\end{aligned}
$$



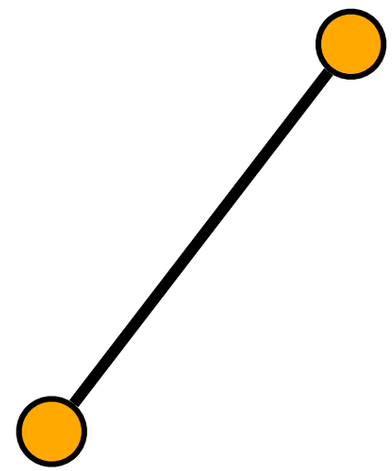
Geometric primitives in 2-D

- 0-dimensional

```
class Point {

  double x;
  double y;
}
```

- 1-dimensional

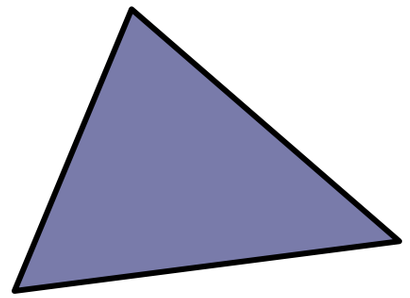
```
class Segment {

  Point p;
  Point q;
}
```



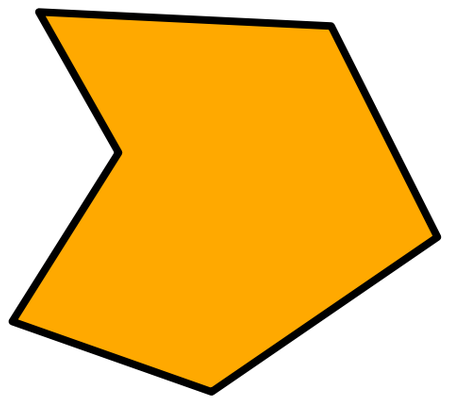
Geometric primitives in 2-D

- 2-dimensional

```
class Triangle {

  Point p;
  Point q;
  Point r;
}
```

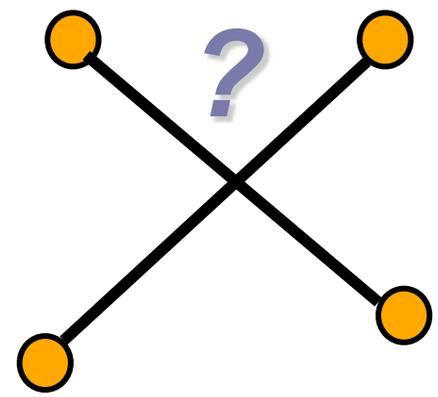


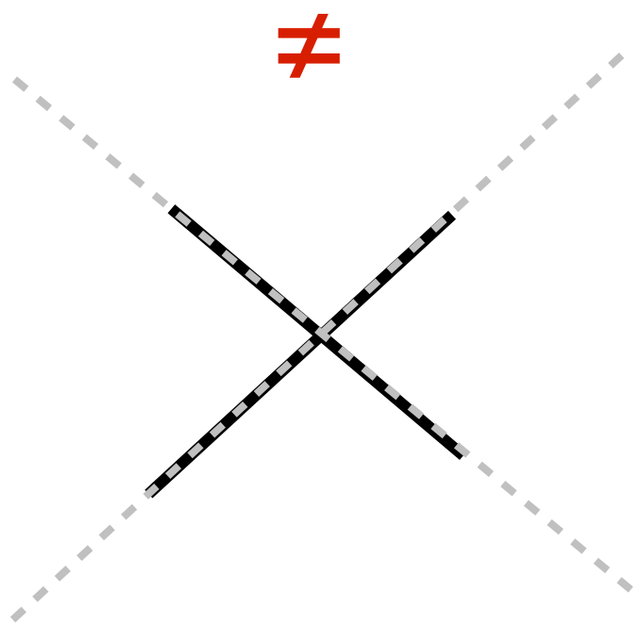
```
class Polygon {

  list<Point> vertices;
  int numVertices;
}
```



Intersecting primitives

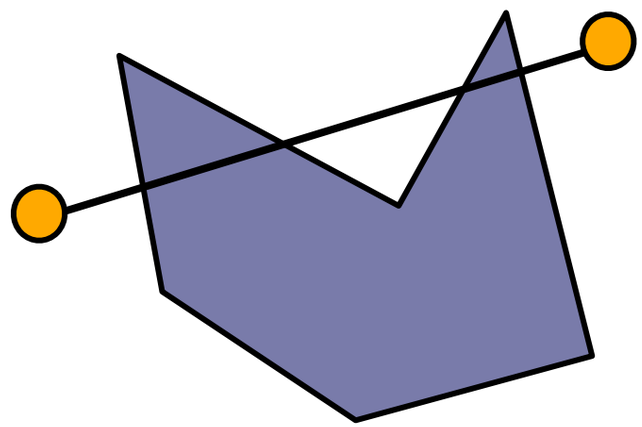
- Determine whether two line segments intersect.





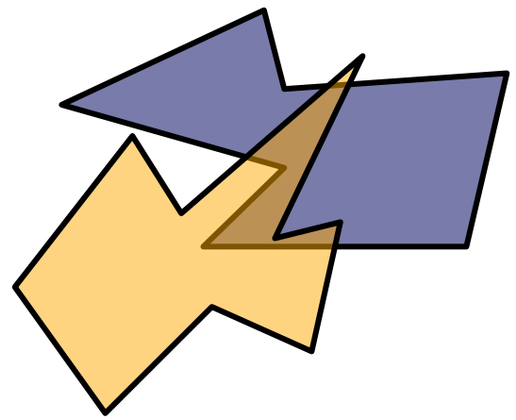
Intersecting a line segment and a polygon

- Intersect the line segment with every edge of the polygon.
- It takes $O(n)$ time for a polygon with n edges.



Intersecting two polygons

- Intersect every edge of one polygon with every edge of the other polygon.
- It takes $O(mn)$ time if the two polygons have m and n edges, respectively.



- The quadratic running time is not acceptable for complex geometric objects.

Key concepts.

- Sampling 2-D and 3-D orientations
- Collision detection, proximity query

Additional information. The survey below is a little outdated, but still provides good coverage of the main ideas for collision detection.

Collision detection between geometric models: a survey. M. Lin and S. Gottschalk.

There are many established collision detection libraries. Google for "collision detection code". It is highly recommended to use one of them instead of code your own, unless it becomes really necessary. It is challenging to implement geometric computation code correctly for various reason discussed earlier.