گزارش پروژه اول رباتیک پیشرفته

مسیریابی ربات های KUKA و PUMA با استفاده از الگوریتم های RRT و SBL

توسط:

علی نورمحمدی اصل

استاد:

دکتر مرادی

بهار ۱۳۹۴

در این پروژه به مسیریابی ربات‌های با درجه آزادی بالا پرداخته شده است. برای این منظور از کتابخانه MPK (Motion Planning Kit) استفاده شده است. این کتابخانه به زبان C++ است و می‌توان برای ربات های مختلف مسیریابی را انجام داد. الگوریتم مورد استفاده برای مسیریابی ربات‌ها در این کتابخانه SBL است.

هدف در این پروژه انجام مسیریابی برای دو ربات PUMA و KUKA است که در شکل (۱) نشان داده شده‌اند. برای مسیریابی از الگوریتم موجود در کتابخانه، SBL، و الگوریتم دیگری به نام RRT استفاده شده است. الگوریتم RRT در MPK موجود نیست و باید برنامه آن نوشته و به کتابخانه MPK افزوده شود.

شکل۱ – ربات های PUMA(b) و KUKA(a)

پروژه را با توجه به اهداف موجود می توان به ۶ بخش زیر تقسیم کرد که در ادامه به توضیح هر یک خواهیم پرداخت.

۱. طراحی محیط شبیه سازی خواسته شده برای ربات KUKA (مدل ربات KUKA موجود است).
۲. طراحی ربات PUMA با توجه CAD داده شده و طراحی محیط شبیه سازی.
۳. پیاده سازی الگوریتم SBL برای دو محیط بالا.
۴. پیاده سازی الگوریتم RRT در MPK .
۵. پیاده سازی الگوریتم RRT برای دو محیط بالا.
۶. مقایسه عملکرد الگوریتم‌های SBL و RRT .

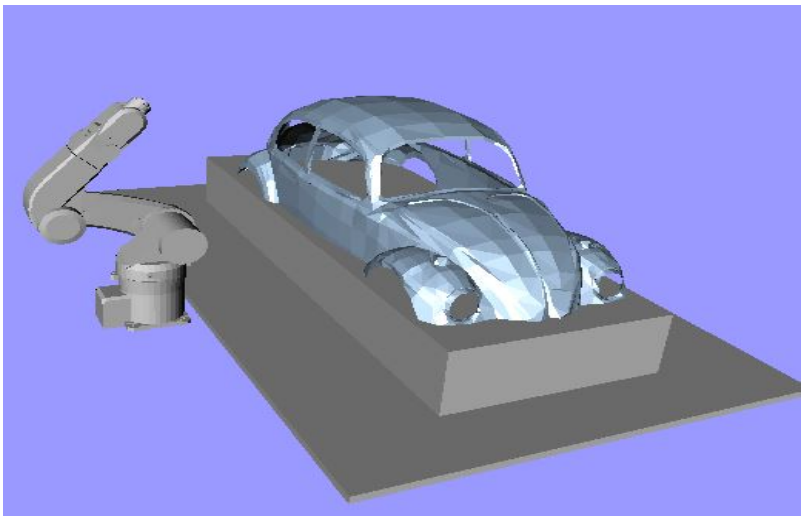
بخش اول: طراحی محیط شبیه سازی خواسته شده برای ربات KUKA

برای طراحی محیط با توجه به اینکه مدل ربات KUKA از پیش طراحی شده است و فایل rob. در اختیار ما گذاشته شده است کافیست تا ربات و موانعی که به شکل‌های بدنه یک اتوموبیل و دو مکعب مستطیل است با هم در یک محیط پیاده کنیم. برای این منظور از برنامه موجود در جدول(۱) استفاده می‌شود.

جدول ۱: برنامه طراحی محیط برای ربات KUKA

```
#Inventor V2.0 ascii

DEF robot mpkRobot {
        fileName "kuka.rob"
        translation 0 0 0
        rotation 1 0 0 -1.57
}

DEF box1 mpkObstacle {
        Separator {
                        Translation {translation 0 0 0}
                        Scale {scaleFactor 500 500 500}
                        DEF __triangulate__ Cube {
                        width 1.5
                        height .5
                        depth 4
                        }
        }
}

DEF box2 mpkObstacle {
        Separator {
                        Scale {scaleFactor 500 500 500}
                        Translation {translation 0 -.25 0}
                        DEF __triangulate__ Cube {
                        width 3
                        height .05
                        depth 5
                        }
        }
}

DEF car mpkObstacle {
        Separator {
                Scale {scaleFactor 500 500 500}
                Translation {translation 0 0 0}
                DEF __triangulate__ File {name "vwbeetle_body.iv"}
        }
}
```

این برنامه در Notepad نوشته و به فرمت iv. ذخیره می‌شود. بخش اول این برنامه ربات KUKA را به محیط وارد می‌کند. بخش دوم و سوم مکعب ها را با ابعاد خواسته شده به عنوان مانع وارد محیط می‌کند و در آخر مدل از پیش طراحی شده بدنه اتوموبیل به عنوان مانع وارد محیط می‌شود. شکل این محیط در شکل (۲) نشان داده شده است. مکان (Translation) و دوران (Rotation) طوری انتخاب شده‌اند که شکل مطلوب حاصل شود.

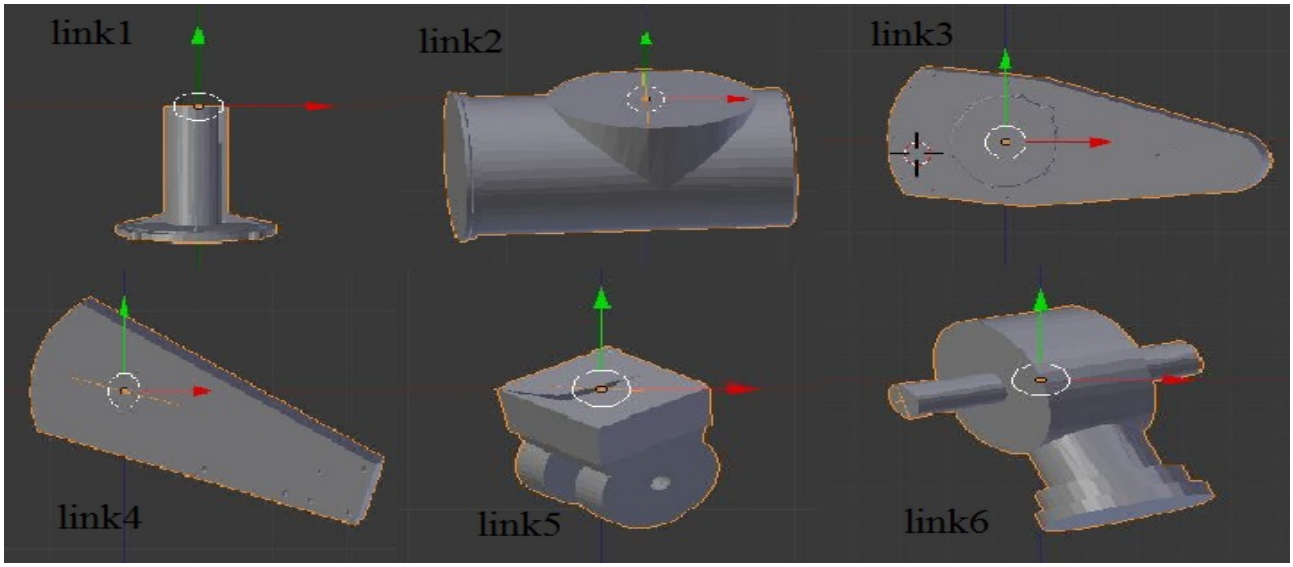


شکل۲- محیط شبیه سازی برای ربات KUKA

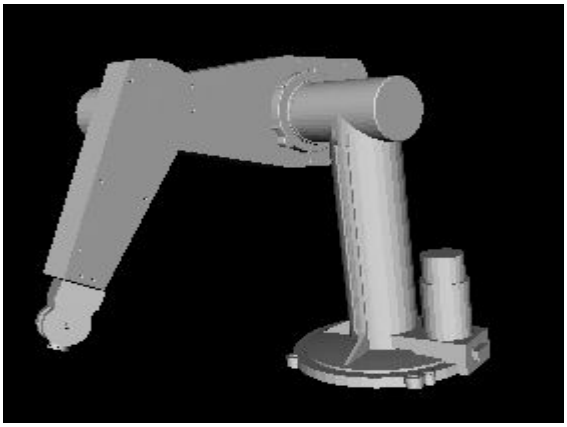
بخش دوم: طراحی ربات PUMA با توجه CAD داده شده و طراحی محیط شبیه سازی

برای طراحی ربات PUMA از مدل این ربات که به صورت یک فایل sldprt. داده شده است، استفاده می‌شود. این فایل توسط نرم افزار SolidWorks به صورت stl. ذخیره می‌شود. سپس با توجه به راهنمای داده شده این فایل را توسط نرم افزار Blender باز می کنیم و هر بازوی ربات را جدا کرده و در یک فایل blend. ذخیره می کنیم. سپس در هر یک از این فایل ها محور مختصات را به محلی که به Link قبلی باید وصل شود، انتقال می دهیم. شکل (۳) هر یک از این link ها را نشان می دهد.

سپس می بایست همه این لینک‌ها را به هم متصل نمود. برای این منظور باید هر لینک را به مختصاتی که باید به لینک دیگر وصل شود انتقال داد. جدول (۲) برنامه نوشته شده برای این منظور را نشان می دهد. این برنامه نیز به صورت iv. ذخیره می‌شود. شکل (۴) شکل ربات به دست آمده از برنامه را نشان می دهد.

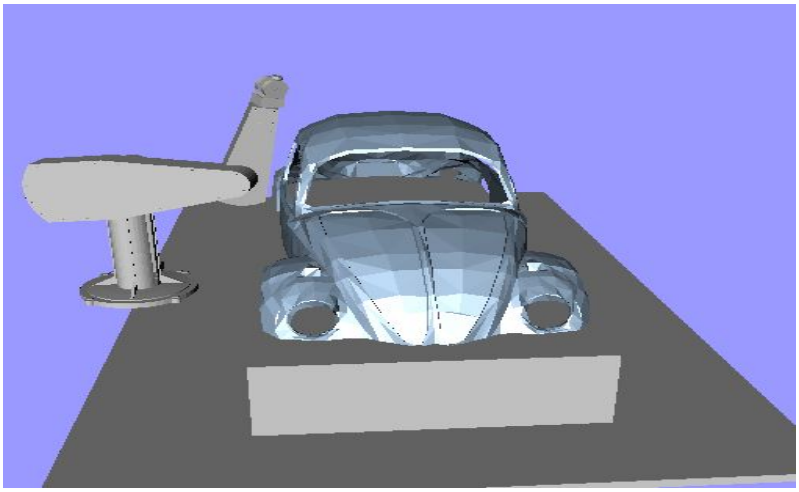


شکل۳- قطعات ربات PUMA با مختصات تغییر کرده در Blender



شکل۴ ربات PUMA تشکیل شده از لینک‌ها

قدم بعدی طراحی یک فایل با فرمت rob. است که در آن لینک‌های ربات به هم متصل شده‌اند و محدودیت حرکتی هر لینک مشخص شده است. جدول (۳) برنامه نوشته شده برای این منظور را نشان می دهد. محدودیت‌های حرکتی هر لینک با توجه به فایل راهنمای ربات PUMA تعیین شده است. در آخر برای طراحی محیط مانند بخش قبل که مربوط به ربات KUKA بود، عمل می کنیم. این برنامه نیز در جدول (۴) نشان داده شده است که به صورت iv. ذخیره می شود. شکل (۵) محیط ساخته شده را نشان می دهد.



شکل۵- محیط شبیه سازی برای ربات *PUMA*

جدول ۲: برنامه طراحی فایل *PUMA.iv* و اتصال لینک ها

```
#Inventor V2.1 ascii
Separator
{
        File {name "puma1.iv"}
        File {name "puma2.iv"}
        Separator
        {
                Translation {translation -0.024 0.078 .136} #offset puma_3 w.r.t puma_2
                File {name "puma3.iv"}
        }
        Separator
        {
                Translation {translation -0.024 0.078 .136} #offset puma_3 w.r.t puma_2
                Translation {translation 0.43 0.03 .11} #offset puma_4 w.r.t puma_3
                File {name "puma4.iv"}
        }
Separator
        {
                Translation {translation -0.024 0.078 .136} #offset puma_3 w.r.t puma_2
                Translation {translation 0.43 0.03 .11} #offset puma_4 w.r.t puma_3
                Translation {translation 0.185 -.307 -.015} #offset puma_5 w.r.t puma_4
                File {name "puma5.iv"}
        }
Separator
        {
                Translation {translation -0.024 0.078 .136} #offset puma_3 w.r.t puma_2
                Translation {translation 0.43 0.03 .11} #offset puma_4 w.r.t puma_3
                Translation {translation 0.185 -.307 -.015} #offset puma_5 w.r.t puma_4
                Translation {translation 0.038 -.067 .008} #offset puma_6 w.r.t puma_5
                File {name "puma6.iv"}
        }
}
```

جدول ۳: برنامه طراحی فایل puma.rob و اعمال محدودیت های چرخشی ربات

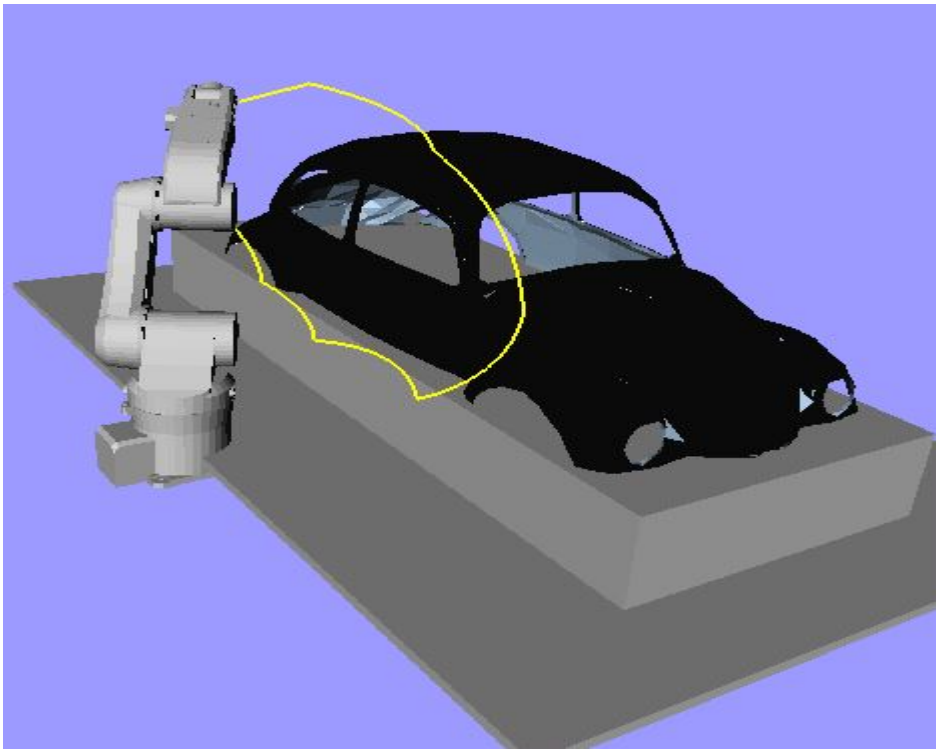
```
#MPK v1.0 robot
# Issue with self-collision of link 1 with link 2 to be checked
# self-coll model NOT complete
selfcoll {
  Link5 Link3
  Link6 Link3
  Link4 Link1
  Link5 Link1
  Link6 Link1
}

joint Link1 {
  model0 "PUMA/puma1.iv"
}
joint Link2 {
  parent Link1
  Rot1 0 1 0 -2.79 2.79
  param 0
  model0 "PUMA/puma2.iv"
}
joint Link3 {
  parent Link2
  ConstTransl_Rot1 -0.024 0.078 .136 0 0 1 -2.18 2.18
  param 1
  model0 "PUMA/puma3.iv"
}
joint Link4 {
  parent Link3
  ConstTransl_Rot1 0.43 0.03 .11 0 0 1 -2.35 2.35
  param 2
  model0 "PUMA/puma4.iv"
}
joint Link5 {
  parent Link4
  ConstTransl_Rot1 0.185 -.307 -.015 0 1 0 -2.61 2.61
  param 3
  model0 "PUMA/puma5.iv"
}
joint Link6 {
  parent Link5
  ConstTransl_Rot1 0.038 -.067 .008 0 0 1 -1.74 1.74
  param 4
  model0 "PUMA/puma6.iv"
}
joint TracePoint {
  parent Link6
  ConstTransl 0 0.05 0
  tracePoint
  coll FALSE
}
```

جدول ۴: برنامه طراحی فایل محیط شبیه سازی برای ربات *PUMA*

```
#Inventor V2.0 ascii
DEF robot mpkRobot {
        fileName "puma.rob"
        translation 0 .419 0
        rotation 1 0 0 0
}
DEF box1 mpkObstacle {
        Separator {
                        Translation {translation 0 0 0}
                        Scale {scaleFactor .6 .6 .6}
                        DEF __triangulate__ Cube {
                        width 1.5
                        height .5
                        depth 4
                        }
        }
}
DEF box2 mpkObstacle {
        Separator {
                        Scale {scaleFactor .6 .6 .6}
                        Translation {translation 0 -.25 0}
                        DEF __triangulate__ Cube {
                        width 3
                        height .05
                        depth 5
                        }
        }
}
DEF car mpkObstacle {
        Separator {
                Scale {scaleFactor .6 .6 .6}
                Translation {translation 0 0 0}
                DEF __triangulate__ File {name "vwbeetle_body.iv"}
        }
}
```

بخش سوم: پیاده سازی الگوریتم SBL برای دو محیط بالا

برای پیاده سازی الگوریتم SBL برای ربات KUKA ابتدا ربات را توسط کلید های ctrl+arrow به مکان مطلوب می‌بریم. کلید space گام های این انتقال را تنظیم می کند. فایل kukaplan.pln را توسط کلید L بارگذاری می‌کنیم. این فایل شامل ۱۰ نقطه پیکربندی ربات است. سپس توسط کلید F2 الگوریتم SBL را اجرا می کنیم. توسط کلید T مسیر طراحی شده را نمایش می دهیم و با کلید F5 ربات مسیر طی شده را طی می کند. با کلید A می‌توان سرعت انیمیشن را کنترل کرد. شکل (۶) مسیر طراحی شده را نشان می دهد.

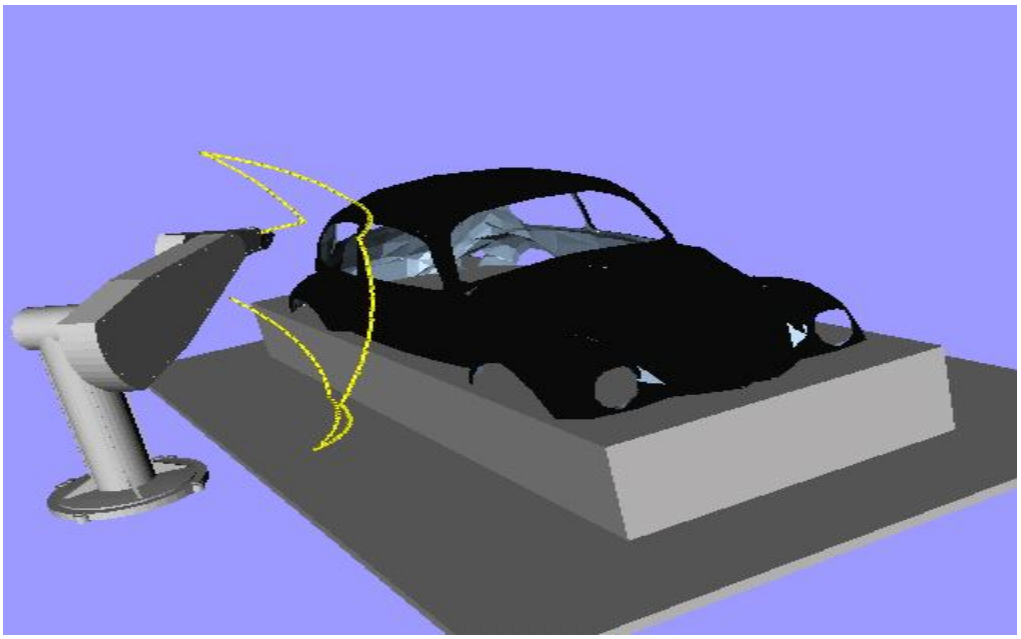


شکل۶- مسیر یافت شده برای ربات KUKA

حال محیط شامل ربات PUMA را اجرا می کنیم. ابتدا ربات را توسط کلید های ctrl+arrow به مکان مطلوب می‌بریم. برای تهیه فایلی مانند kukaplan.pln باید با کلیک کردن بر روی لینک مورد نظر و استفاده از کلیدهای arrow ربات را در حالت مورد نظر قرار دهیم سپس با کلید insert این پیکربندی را به نقاط اضافه می‌کنیم. این کار را برای تعداد نقاط مطلوب انجام می‌دهیم. در آخر با استفاده از کلید S این مجموعه پیکربندی را با نام مورد نظر ذخیره می‌کنیم. حال می توانیم مراحل بالا را برای این ربات نیز در نظر بگیریم. جدول ۵ نمونه ای از فایل ذخیره شده به فرمت pln. را نشان می‌دهد که هر سطر یک پیکربندی ربات و هر ستون موقعیت لینک را در آن ساختار نشان می‌دهد. شکل (۷) مسیر طراحی شده برای این ربات توسط الگوریتم SBL را نشان می دهد.

جدول ۵:یک نمونه مجموعه پیکربندی برای ربات *PUMA*

*	0.67 0.55 0.65 0.5 0.5
*	0.62 0.53 0.71 0.5 0.5
*	0.54 0.53 0.77 0.5 0.5
*	0.54 0.53 0.77 0.5 0.5
*	0.54 0.53 0.77 0.5 0.5
*	0.43 0.53 0.74 0.5 0.5
*	0.38 0.46 0.62 0.5 0.5
*	0.47 0.41 0.5 0.5 0.5
*	0.59 0.41 0.55 0.5 0.5
*	0.7 0.5 0.6 0.5 0.5



شکل۷- مسیر یافت شده برای ربات*PUMA*

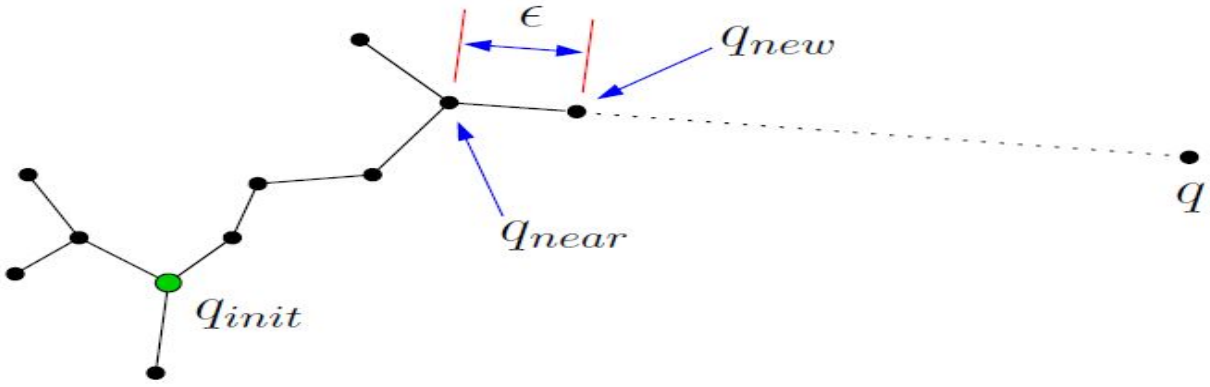
بخش چهارم: پیاده سازی الگوریتم RRT در MPK

در روش RRT یک درخت جست وجو از مکان اولیه شروع به رشد می‌کند و با استفاده از داده‌های کنترلی به یک موقعیت جدید می‌رسد. هر رأس بیانگر یک موقعیت جدید است و هر بردار یک ورودی است که برای رسیدن به موقعیت جدید مورد استفاده قرار می‌گیرد. ایده اصلی این روش در ایجاد نقاطی تصادفی در فضا و کشیدن درخت جست‌وجو به سمت آنها است. نحوه انتخاب تصادفی نقاط موجب می‌شود فضای جواب در این روش به خوبی پوشش داده شده و شانس وجود هر نقطه از فضای آزاد با یکدیگر برابر باشد. الگوریتم این روش در جدول ۶ نشان داده شده است.

جدول۶: الگوریتم RRT

```
Algorithm BuildRRT
  Input: Initial configuration qinit, number of vertices in RRT K, incremental distance Δq)
  Output: RRT graph G

  G.init(qinit)
  for k = 1 to K
    qrand ← RAND_CONF()
    qnear ← NEAREST_VERTEX(qrand, G)
    qnew ← NEW_CONF(qnear, qrand, Δq)
    G.add_vertex(qnew)
    G.add_edge(qnear, qnew)
  return G
```

در این الگوریتم برای جلوگیری از بررسی مسیر های طولانی بعد از تولید مکان تصادفی نقطه جدید را در راستای آن ولی در گامی نزدیک‌تر در نظر می گیریم. شکل (۸) این موضوع را نشان می دهد.



شکل۸- روند تشکیل درخت در RRT

در برنامه نویسی الگوریتم RRT مانند شبه کد داده شده ابتدا توسط دستور زیر یک پیکربندی تصادفی تولید می کنیم.

```
randConfig.LocalBoxSample(qs,1,&qRand);
```

سپس بررسی می‌شود که این نقطه در فضای آزاد باشد.

```
point_checker->collision(&qRand);
```

سپس نزدیک ترین پیکربندی به این پیکربندی را می‌یابیم و اندیس آن را ذخیره می کنیم.

```
for (j = genPath.begin(); j != genPath.end(); ++j){
		dist = (*j).dist(qRand);
		if (jj == 0 || dist < minDist){
			minDist = dist;
			indMin = jj;
			prevState = *j;
		}
		jj++;
}
```

در مرحله بعد فاصله این دو پیکربندی را محاسبه می‌کنیم. اگر این فاصله کمتر از مقدار تعیین شده توسط کاربر بود آن را به عنوان پیکربندی جدید انتخاب می‌کنیم ولی در غیر این صورت یک پیکربندی جدید در همان راستا ولی با طول مشخص را انتخاب می کنیم.

```
if (minDist>id){
	a = id / minDist;
	for (int j = 0; j < qRand.size(); j++){
		valn = (prevState[j])*(1 - a) + (qRand[j]) * a;
		qNew.push_back(valn);
	}
	minDist = (prevState).dist(qNew);
}
else
	qNew = qRand;
```

سپس توسط دستور زیر در فضای آزاد بودن مسیر بررسی می شود.

```
edgeCollCheck(prevState, qNew, minDist);
```

تابع edgeCollCheck به صورت زیر تعریف شده است.

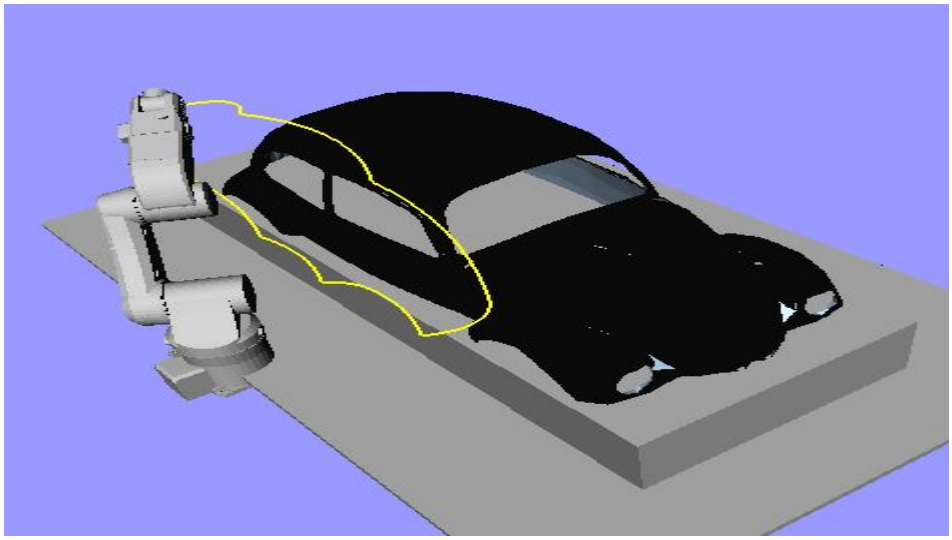
```
bool rrtPlanner::edgeCollCheck(const mpkConfig& prevState, const mpkConfig& qNew, double minDist){
	int m = minDist / EPSILON;
	mpkConfig ql;
	bool fail = false;
	int i = 1;
	while (i < m && !fail){
```

```
                double val;
                double t;
                ql.clear();
                t = double(i) / m;
                for (int j = 0; j < qNew.size(); j++){
                        val = (prevState[j])*(1-t) + (qNew[j]) * t;
                        ql.push_back(val);
                }
                fail = point_checker->collision(&ql);
                i++;
        }
return fail;
}
```

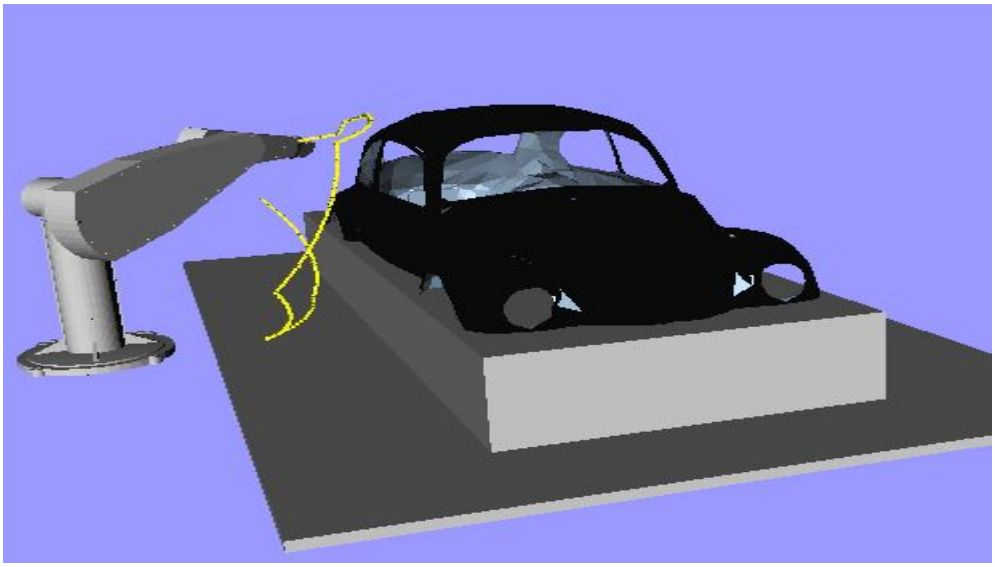
در هر مرحله فاصله نقطه پیکربندی جدید تا هدف بررسی می‌شود، اگر فاصله کمتر از یک حد مشخص شده توسط کاربر باشد، مسیر بین آن پیکربندی تا هدف بررسی می‌شود. این فرآیند تا زمانی که نقطه هدف برسیم، ادامه می‌یابد. در انتها مسیر یافت شده به عنوان پاسخ معین می‌شود.

بخش پنجم: پیاده سازی الگوریتم RRT برای دو محیط بالا

مانند بخش سوم یک بار محیط شامل ربات KUKA را اجرا می‌کنیم و فایل kukaplan.pln را بار گذاری می‌کنیم و با کلید F7 مسیر را توسط الگوریتم RRT می یابیم. یک بار نیز محیط شامل ربات PUMA را اجرا می‌کنیم و فایل pumaplan.pln را بار گذاری می‌کنیم و مانند حالت قبل مسیر را می یابیم. شکل ۹ و ۱۰ به ترتیب مسیر یافت شده برای ربات های KUKA و PUMA را نشان می دهد.



شکل۹- مسیر یافت شده برای رباتKUKA



شکل۱۰- مسیر یافت شده برای ربات PUMA

بخش ششم: مقایسه عملکرد الگوریتم‌های SBL و RRT

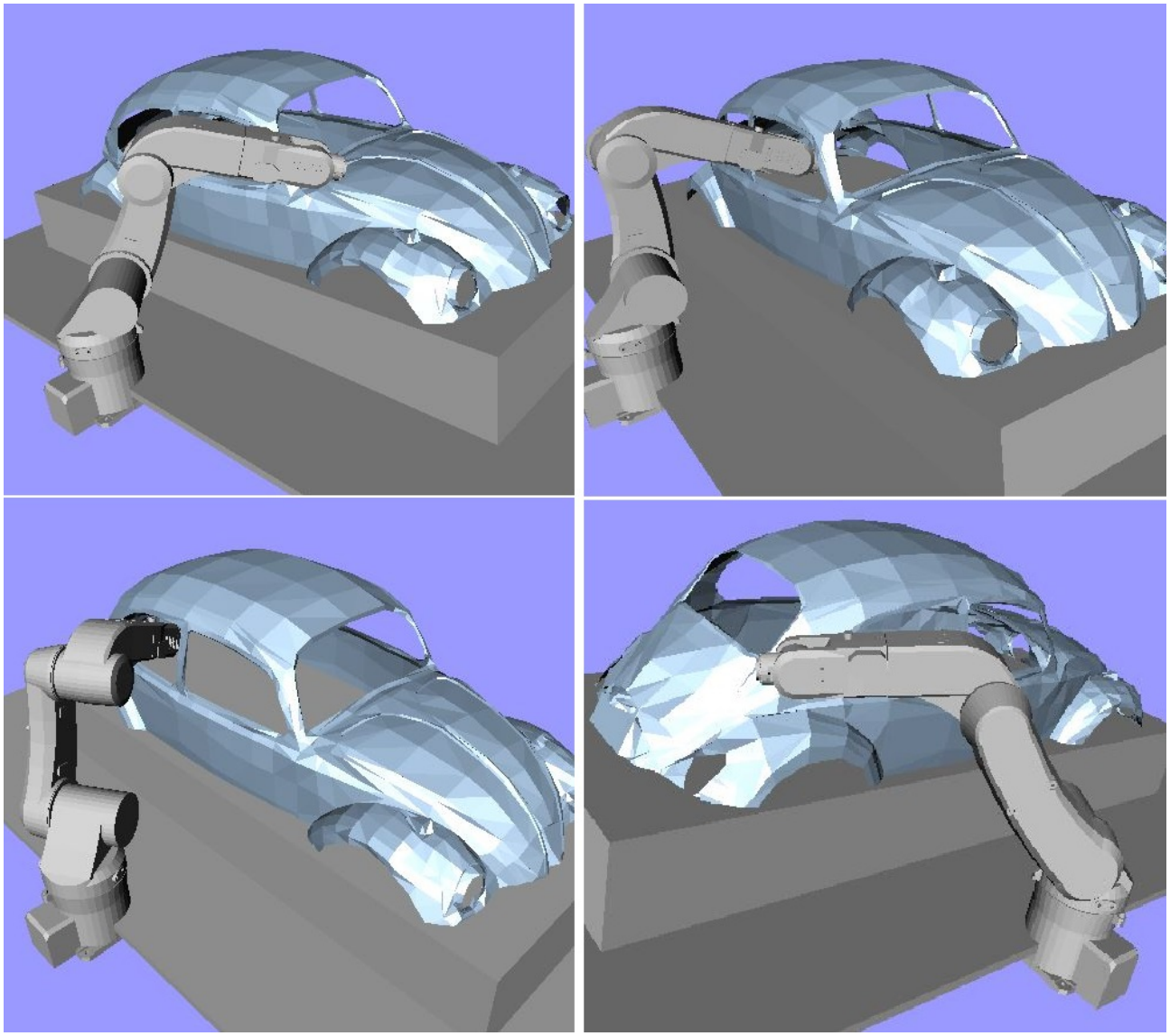
برای مقایسه الگوریتم‌های RRT و SBL یک مسیر نسبتاً دشوار را برای ربات KUKA در نظر گرفتیم. این مسیر شامل چهار نقطه است که دو نقطه از آن داخل ماشین است. از آنجایی که در الگوریتم SBL ابتدا وجود مسیر مستقیم بین نقطه شروع و پایان را بررسی می‌شود(توسط کد زیر)، با یک مسیر ساده نمی توان این دو الگوریتم را مقایسه نمود و نیاز به یک مسیر با وجود مانع است. شکل ۱۱ این چهار نقطه را که ربات باید از آن‌ها رد شود را نشان می‌دهد.

```
mpkSimpleSegmentChecker seg_check(point_checker, &qs, &qg, EPSILON);
```

از آنجایی که برای مقایسه دو الگوریتم نیاز به مقایسه طول مسیر طراحی شده نیز است، تابعی جهت محاسبه طول مسیر به صورت زیر نوشته شد.

```
void pathLength(vector<config>& plan, double &len){
	len = 0;
	for (int i = 0; i < plan.size() - 1; i++){
		len = len + plan[i].q.dist(plan[i + 1].q);
	}
}
```

هر الگوریتم بیست مرتبه تکرار شده است و زمان سپری شده برای طراحی مسیر، طول مسیر طراحی شده و تعداد نقاط موجود در این مسیر در جدول آمده است. همچنین انحراف از معیار و میانگین موارد گفته شده در بالا نیز جهت مقایسه در جدول ۷ بیان شده است.



شکل۱۱- چهار نقطه پیکربندی ربات جهت انجام مسیریابی

جدول۷: عملکرد الگوریتم های RRT و SBL

#	SBL			RRT		
	Time(s)	Path Length	#Vertex	Time(s)	Path Length	#Vertex
1	3.632	3.506	15	1.815	2.570	14
2	3.344	2.616	12	3.151	3.602	17
3	3.075	3.725	16	0.734	2.555	12
4	3.266	3.539	14	0.446	2.172	10
5	3.406	3.688	15	1.037	2.392	11
3	3.652	4.927	22	1.436	2.934	14
7	3.604	3.770	17	0.381	2.460	11
8	3.433	3.414	15	4.111	3.278	16
9	3.571	4.930	19	0.482	3.235	15
10	3.565	6.272	22	1.317	3.060	15
11	3.187	5.175	21	1.764	4.582	22
12	3.43	4.040	18	3.098	3.072	15
13	3.47	5.238	22	5.850	2.717	13
14	3.61	4.947	21	3.743	4.657	23
15	4.17	3.331	17	2.448	4.416	21
16	4.17	5.010	20	0.348	2.376	11
17	3.08	4.339	21	3.240	5.240	21
18	3.465	3.026	13	0.774	5.180	22
19	3.392	6.095	25	0.326	3.301	18
20	3.175	4.104	16	0.474	3.270	18
Avg	3.4848	4.2846	18.050	1.8488	3.3535	15.950
StD	0.2936	0.9987	3.5759	1.560	0.9569	4.1482

همان طور که از مقادیر میانگین و انحراف از معیار مشخص است الگوریتم RRT دارای سرعت بیشتری نسبت به الگوریتم SBL است زیرا نیازی به نمونه برداری از محیط و ساختن Roadmap ندارد، به عبارتی شامل مرحله یادگیری نیست. اما سرعت آن دارای انحراف از معیار بیشتری است که ناشی از تصادفی بودن این روش است. همچنین با توجه به جدول می‌توان گفت در این شبیه سازی RRT مسیرهای کوتاهتری را پیدا کرده است.