SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE, FAKULTA ELEKTROTECHNIKY A INOFRMATIKY

Riadenie mobilného robota

Zadanie na predmet RMR

dokumentácia

Martin Dodek

Michaela Kolárska 7.5.2019

Obsah

Programátorská príručka

Lokalizácia

Lokalizácia a polohovanie funguje na viacerých princípoch. Napríklad pri odometrii sa využívajú údaje z enkóderov, ktorých naintegrovaný počet impulzov si prevedieme na metre a získame tak dĺžku prejdenej dráhy kolesa. Pri štarte programu sme si počiatočnú hodnotu enkóderov uložili, aby sme neskôr vedeli pracovať s novými aktuálnymi údajmi a určovať relatívnu polohu robota. V programe sa uvažuje s pretečením hodnoty premennej enkóderov po cca 65 000 impulzoch (16 bit unsigned short).

Odometria je v programe implementovaná ako zvláštna trieda a výpočet sa realizuje 4 spôsobmi. Konkrétne sú to tieto algoritmy postupnej integrácie:

- Forward Euler
- Backward Euler
- Trapezoidal rule

Ukážka implementácie triedy odometrie:

```
class Odometry
{
private:
	float wheel_distance_right = 0;
	float wheel_distance_left = 0;
	float wheel_last_distance_right = 0;
	float wheel_last_distance_left = 0;
	float delta_alfa = 0;
	float delta_l = 0;
	float delta_l_left = 0;
	float delta_l_right = 0;

public:
	RobotPosition position;

	Odometry()
	{
		odometry_init(RobotPosition(0,0,0));
	}
	Odometry(float x, float y,float alfa)
	{
		odometry_init(RobotPosition(x, y, alfa));
	}
	void odometry_backward_euler(Encoder encL, Encoder encR);
	void odometry_forward_euler(Encoder encL, Encoder encR);
	void odometry_trapezoidal_rule(Encoder encL, Encoder encR);
	void odometry_curved(Encoder encL, Encoder encR);
};
```

Ich presnosti sa vzájomne líšia v závislosti od tvaru a charakteru prejdenej dráhy a od „vzorkovacej periódy“ s akou sa odometria počíta, resp. ako často prichádzajú nové dáta. Primárne používame

odometriu, ktorá sa počíta vo funkcii *odometry_curved* ako pohyb po kružnicových oblúkoch. Počítaná bola týmito vzťahmi:

$$x_{k+1} = x_k + \frac{d(l_r + l_i)}{2(l_r - l_i)}(\sin\varphi_{k+1} - sin\varphi_k)$$

$$y_{k+1} = y_k + \frac{d(l_r + l_i)}{2(l_r - l_i)}(\cos\varphi_{k+1} - cos\varphi_k)$$

$$\varphi_{k+1} = \varphi_k + \Delta_\alpha$$

Pre ilustráciu – implementácia metódy výpočtu:

```
void Odometry::odometry_curved(Encoder encL, Encoder encR)
{

	wheel_distance_left = encL.encoder_real_value*tickToMeter;
	wheel_distance_right = encR.encoder_real_value*tickToMeter;
	delta_l_left = wheel_distance_left - wheel_last_distance_left;
	delta_l_right = wheel_distance_right - wheel_last_distance_right;
	delta_l = (delta_l_left + delta_l_right) / 2;
	delta_alfa = -(delta_l_left - delta_l_right) / d;
	if (abs(delta_l_right - delta_l_left) > arc_line_switch_treshold)
	{
		position.coordinates.X += (d*(delta_l_left + delta_l_right) / (delta_l_right -
	delta_l_left) / 2 * (sin(position.alfa + delta_alfa) - sin(position.alfa)));
	position.coordinates.Y -= (d*(delta_l_left + delta_l_right) / (delta_l_right -
	delta_l_left) / 2 * (cos(position.alfa + delta_alfa) - cos(position.alfa)));
	}
	else
	{
	position.coordinates.X += delta_l * cos(position.alfa);
	position.coordinates.Y += delta_l * sin(position.alfa);
	}
	position.alfa += delta_alfa;
	wheel_last_distance_left = wheel_distance_left;
	wheel_last_distance_right = wheel_distance_right;
}
```

Ďalšou alternatívou je využitie viacerých implementácii odometrie s priemerovaním výsledku, prípade s váhovaným priemerovaním, ktoré je implementované pomocou preťažených operátorov. Ukážka použitia priemerovania:

```
	odometria_1.odometry_forward_euler(encL, encR);
	odometria_2.odometry_backward_euler(encL, encR);
	odometria_3.odometry_trapezoidal_rule(encL, encR);
	odometria_4.odometry_curved(encL, encR);
	odometry_position = (odometria_4.position+odometria_3.position)/2;
```

Polohovanie

Vytvorili sme si špeciálny nelineárny regulátor pre diferenciálny podvozok za účelom dosiahnutia želanej polohy robota akčnými zásahmi. Za stavové premenné systému, ktoré chceme riadiť (stabilizovať), považujeme len súradnice X a Y.

Uhol natočenia theta nie je potrebné nijako špecificky riadiť, jeho hodnota môže vo výsledku ľubovoľná. Samozrejme musí byť taká, aby celý systém bol stabilný a konvergoval do rovnovážneho stavu (poloha musí konvergovať k 0,0).

Vstupom do systému sú signály vt a omega – translačná rýchlosť a rýchlosť rotácie. Výstup X aj Y sú vlastne integrátory nelineárne závislé od uhla theta.

$$\begin{pmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{pmatrix} = \begin{pmatrix}\cos(\theta) & 0\\ \sin(\theta) & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}v_t\\ \omega\end{pmatrix}$$

Cieľom riadenia je stabilizovať tento systém do polohy (0,0) vychádzajúc z ľubovoľných počiatočných podmienok, a to riadením vstupov vt a omega. S použitím Lyapunovovej teórie stability sme navrhli nelineárne riadenie. K dispozícii máme samozrejme spätnú väzbu od všetkých stavových veličín (poloha,uhol).

Pre splnenie podmienok použitia Lyapunovovej teórie stability je nutné uvažovať stabilizáciu do bodu (0,0). Všeobecná stabilizácia do iného ľubovoľného bodu sa dá dosiahnuť rovnakým spôsobom ale s použitím transformačných vzťahov pre súradnice, ktoré posúvajú stred súradnicového systému do žiadaného bodu.

Aby sa to ešte trochu skomplikovalo vstupy budú musieť byť mierne modifikované. Nemáme totiž k dispozícii priamo možnosť meniť rýchlosť otáčania omega ale iba polomer zatáčania R.

Rýchlosť rotácie potom závisí aj od translačnej rýchlosti ω=v_t/R. Translačnú rýchlosť vt vieme nastavovať priamo a môže byť ako kladná, tak aj záporná.

Potom sa sústava diferenciálnych rovníc pozmení (vt a R sú signály akčného zásahu)

$$\begin{pmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{pmatrix} = \begin{pmatrix}\cos(\theta) & 0\\ \sin(\theta) & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}v_t\\ v_t/R\end{pmatrix}$$

Stabilizácia vzdialenosti robota:

$$z = \sqrt{x^2 + y^2}$$

Potom časová derivácia – diferenciálna rovnica vzdialenosti robota:

$$\dot{z} = v_t \frac{x\cos(\theta) + y\,sin(\theta)}{z}$$

Zvolím Lyapunovovu funkciu v kvadratickom tvare:

$$V(z) = \frac{1}{2}z^2$$

Potom:

$$\frac{dV}{dz} = z$$

Derivácia LF pozdĺž trajektórie systému:

$$\dot{V} = v_t(x\cos(\theta) + y\,sin(\theta))$$

Ak chcem aby bolo $\dot{V}$ záporne semidefinitné a tým pádom bola vzdialenosť *z* stabilná, musím akčným zásahom v_t zabezpečiť aby výraz bol záporný. Najjednoduchšie to spravím takto:

$$v_t = -m(x\cos(\theta) + y\,sin(\theta))$$

Potom:

$$\dot{V} = -m(x\cos(\theta) + y\,sin(\theta))^2$$

Kde *m* je konštanta zosilnenia.

Akčný zásah v_t má pre akúkoľvek polohu robota a uhol natočenia to správne znamienko a veľkosť

Stabilizácia uhla natočenia robota:

Žiadanú hodnotu uhla natočenia robota nemám explicitne definovanú.

Jediný zmysel dáva určiť si žiadanú hodnotu uhla natočenia tak, že to bude vždy opačná hodnota k uhlu (akoby fáza komplexného čísla) ktorý dostaneme transformáciou súradníc polohy robota X,Y do polárnych súradníc (atan(y/x)). Chcem teda smerovať robota vždy opačným smerom aký smer má jeho polohový vektor.

Uhol polohového vektora (fáza polohového vektora) robota označím ako alfa:

$$\alpha = \mathrm{atan}\left(\frac{y}{x}\right)$$

Označím potom chybu natočenia robota (výchylka uhla natočenia) ako rozdiel aktuálneho natočenia a fázy polohového vektora.

$$\theta_e = \theta - \alpha$$

Pre dynamiku regulačnej odchýlky uhla natočenia robota bude platiť nasledujúca rovnica:

$$\dot{\theta}_e = \dot{\theta} - \dot{\alpha}$$

Túto diferenciálnu rovnicu môžem teraz chcieť stabilizovať k nule, teda požadujem nulový rozdiel uhlov (natočenia robota a smerového uhla).

Za $\dot{\theta}$ ešte dosadím známy výraz z pôvodného systému.

$$\dot{\theta}_e = \frac{v_t}{R} - \dot{\alpha}$$

Následne viem ešte dosadiť už známu translačnú rýchlosť robota, ktorá vyplynula z riešenia stabilizácie vzdialenosti Lyapunovovou rovnicou.

$$\dot{\theta}_e = \frac{-(x\cos(\theta) + y\,sin(\theta))}{R} - \dot{\alpha}$$

Ostáva určiť časovú deriváciu uhla polohového vektora:

$$\dot{\alpha} = \frac{\mathrm{d}\left(\mathrm{atan}\left(\frac{y}{x}\right)\right)}{dt}$$

X a Y sú časovo premenlivé signály a preto musíme derivovať nasledovne:

$$\dot{\alpha} = \frac{1}{1 + \left(\frac{y}{x}\right)^2}\left(-\frac{y}{x^2}\dot{x} + \frac{1}{x}\dot{y}\right)$$

$$\dot{\alpha} = \frac{v_t}{z^2}(-y\cos(\theta) + x\,\sin(\theta))$$

Dynamika odchýlky uhla natočenia bude:

$$\dot{\theta}_e = v_t\left(\frac{1}{R} + \frac{1}{z^2}(y\cos(\theta) - x\,\sin(\theta))\right)$$

Zvolíme kandidáta na Lyapunovovu funkciu v tvare:

$$V(z) = \frac{1}{2}{\theta_e}^2$$

Potom:

$$\frac{dV}{d\theta_e} = \theta_e$$

Derivácia LF pozdĺž trajektórie systému bude:

$$\dot{V} = \frac{dV}{dt} = \frac{dV}{d\theta_e}\frac{d\theta_e}{dt}$$

$$\dot{V} = \theta_e v_t\left(\frac{1}{R} + \frac{1}{z^2}(y\cos(\theta) - x\,\sin(\theta))\right)$$

Potrebujem dosiahnuť zápornú semidefinitnosť derivácie Lyapunovovej funkcie pozdĺž trajektórie systému pre zaručenie stability.

$$\dot{V} = -(\theta - \mathrm{atan}\left(\frac{y}{x}\right))(x\cos(\theta) + y\sin(\theta))\left(\frac{1}{R} + \frac{1}{z^2}(y\cos(\theta) - x\,\sin(\theta))\right)$$

Ako ale akčným zásahom R zabezpečiť zápornú semidefinitnosť tohto výrazu ?

Aproximácia chyby(výchylky) natočenia robota:

Ak poznám uhlovú rýchlosť (rotácie) polohového vektora robota (už som ju odvodil).

$$\dot{\alpha} = \frac{v_t}{z^2}(-y\cos(\theta) + x\,\sin(\theta))$$

Môžem tvrdiť, že jej znamienko, a v princípe aj veľkosť (bez prenásobenia vt) je úmerná práve výchylke natočenia (chybe natočenia robota).

To prakticky znamená, že keď polohový vektor (v zmysle X,Y) rotuje okolo žiadanej polohy, tak sme nesprávne natočení (kolmo k cieľu). Naopak ak robot smeruje správne k cieľu, polohový vektor robota nerotuje ale iba sa zmenšuje alebo zväčšuje (ideme priamo na cieľ).

Preto je znamienko rotácie polohového vektora rovnaké ako znamienko chyby uhla natočenia a ich veľkosti sú si úmerné, samozrejme bez uvažovania absolútnej veľkosti člena v_t, ten sa správa ako zosilnenie rotácie.

$$sign(v_t)\frac{1}{z^2}(-y\cos(\theta) + x\,\sin(\theta)) \approx \theta_e$$

Potom výraz pre Lyapunovou rovnicu

$$\dot{V} = \theta_e v_t\left(\frac{1}{R} + \frac{1}{z^2}(y\cos(\theta) - x\,\sin(\theta))\right)$$

prejde to tohto tvaru:

$$\dot{V} = -(sign(v_t)\frac{1}{z^2}(-y\cos(\theta) + x\,\sin(\theta)))(v_t)\left(\frac{1}{R} + \frac{1}{z^2}(y\cos(\theta) - x\,\sin(\theta))\right)$$

Čo po úprave bude:

$$\dot{V} = -v_t\ sign(v_t)\left(\frac{\theta_e}{R} + \left(\frac{1}{z^2}(y\cos(\theta) - x\,\sin(\theta))\right)^2\right)$$

A platí:

$$-v_t\ sign(v_t) = -|v_t|$$

Kde výraz

$$\left(\frac{1}{z^2}(y\cos(\theta) - x\,\sin(\theta))\right)^2 \approx {\theta_e}^2$$

Je vždy kladný a v podstate zastupuje druhú mocninu výchylky

Potom ostáva zabezpečiť, aby aj výraz

$$-|v_t|\left(\frac{\theta_e}{R}\right)$$

Bol vždy záporný

Najjednoduchšie to realizujem:

$$R = k\frac{|v_t|}{\theta_e}$$

Kde θ_e viem ľahko vypočítať aj bez použitia atan2, ako bolo už ukázané:

$$\theta_e \approx sign(v_t)\frac{1}{z^2}(-y\cos(\theta) + x\sin(\theta))$$

Kde *k* je konštanta zosilnenia

V programe riadenia robota sa primárne využíva metóda s názvom *regulate* v súbore *regulator.cpp*. Jej programová implementácia sa nachádza na nasledujúcom úryvku:

```
void RobotRegulator::regulate(RobotPosition& current_position, RobotPosition& desired_position)
{
	float alfa;
	output.translation_speed = -translation_gain * ((current_position.coordinates.X - 
desired_position.coordinates.X)*cos(current_position.alfa) + (current_position.coordinates.Y - 
desired_position.coordinates.Y)*sin(current_position.alfa));
	alfa = Point_angle(current_position.coordinates - desired_position.coordinates);
	error.X = -cos(alfa - current_position.alfa)*sign(output.translation_speed);
	error.Y = -sin(alfa - current_position.alfa)*sign(output.translation_speed);
	delta = Point_angle(error);
	symmetry_correct(current_position.coordinates,desired_position.coordinates);
	output.radius = rotation_gain / delta * output.translation_speed;
	saturate_radius();
	nonlinear_power_function();
	saturate_speed();
}
```

Obchádzanie prekážok

Výsledkom tejto úlohy je bezkolízny pohyb robota z bodu A do bodu B. To znamená, vedieť detegovať v prostredí dynamické prekážky, ktoré sú v kolíznom kurze s dráhou robota a vhodne pozmeniť trajektóriu tak, aby sme sa kolízii vyhli.

Pôvodne definované algoritmy zo zadania (hmyzie algoritmy) sa pri praktickej implementácii javili zložité. Preto som vymyslel kombináciu niektorých vlastností pôvodných algoritmov spolu s použitím záplavového algoritmu z úlohy 4 – plánovanie trajektórie.

V princípe sa využívajú dáta z aktuálneho (live) laserového scanu prostredia. Z množiny meraných bodov sa abstrahujú objekty (prekážky). Následne sa analytickou geometriou priamok (parametrické a všeobecné vyjadrenie priamky v rovine) a výpočtom vzdialenosti bod – priamka určí, či sa nasnímaný bod nachádza v zóne kolíznej dráhy s robotom. Takto vyselektované objekty (nielen kolízne body, ale celý súvislý objekt) sa zaznamenajú do mapy prostredia, kde sa lokálne aplikuje záplavový algoritmus pre nájdenie bezkolíznej trajektórie.

Ukážka kľúčovej metódy rozlišovania, či sa bod nachádza v ceste robota.

```
bool RobotControll::is_point_in_way(Point m)
{	//parametricke vyjadrenie priamky
	Point A = actual_position.coordinates;
	Point B = wanted_position.coordinates;
	float k_x = (B - A).X;
	float k_y = (B - A).Y;
	//vseobecna rovnica priamky
	float a, b, c;
	a = -k_y;
	b = k_x;
	c = -k_x * A.Y + k_y * A.X;

	float dist = abs(a*m.X + b * m.Y + c) / PointLength(Point{ a,b });
	if (dist < zone_width && is_triangle_sharp(A,B,m))
	{
			return true;
	}
	return false;
}
```

Mapovanie

Práca s mapou a všetky metódy objektu mapy sú implementované v súbore *map.cpp.*

To zahŕňa klasické konštruktory (podľa zadaných fyzických rozmerov mapy a veľkosti mriežky – rozlíšenia mapy), copy konštruktor pre vytváranie lokálnych kópii objektu, dynamickú alokáciu pamäte pre bunky mapy a samozrejme deštruktory.

Okrem toho objekt mapy využíva preťažených operátorov a operátor prístupu do poľa [] so špecifickou indexačnou štruktúrou.

Ukážka triedy Mapa:

```
class Mapa
{
public:
	Mapa()
	Mapa(int rows,int cols, float  x_low,float x_high,float y_low,float y_high,std::string
filename="")
	Mapa&  operator=(const Mapa& source)
	int& operator[](Matrix_position position)
	Mapa operator && (Mapa& other)
	long sum_elements()
	int get_rows() { return rows; }
	int get_cols() { return cols; }
	Mapa (const Mapa& source,bool copy=true)
	~Mapa()
	void FloodFill_fill(Point start, Point target,bool diagonal);
	bool FloodFill_find_path(Mapa& map_with_path,Point start, Point target, floodfill_priority
priority, std::queue <RobotPosition>& path,bool diagonal);

	std::list<Point> get_obstacles_points();
	int assert_matrix_indices(Matrix_position XY);
	void enhance_obstacles(int window_size);
	int addPoint(Point P, cell_content content);
	void addPointToHistogram(Point P);
	void buildFromHistogram(Mapa& histogram, int treshold);
	void fill_with_objects(map_loader::TMapArea objects);
	void saveMap(std::string filename);
```

```
        void clearMap();
        void loadMap(std::string filename);

private:

        int cols = 0;
        int rows = 0;
        float  x_lim[2], y_lim[2];
        int** cells_data=NULL;
        void allocate()
        void deallocate()
```

Metódy objektu mapa implementujú zaznačovanie bodov do mapy, budovanie histogramu . Následné vytvorenie mapy z takéhoto histogramu meraní pre účely štatistickej separácie šumu a štatisticky nevýznamných údajov. Okrem tohto spôsobu vytvárania mapy, je tu ešte možnosť jej načítania známej mapy zo súboru, ktoré je implemetované vo funkcii *load_map*.

Mapovanie priestoru sa realizuje s využitím údajov z laserového skenera. Cieľom je získať mapu priestoru so zaznačenými prekážkami. V prvom rade sa laserové merania filtrujú na základe kvality a definovanej maximálnej/minimálnej meranej vzdialenosti. Tieto dáta sa prepočítavajú transformáciou do globálnych súradníc. To sa realizuje vo funkcii *lidar_measure_2_point.*

```
Point lidar_measure_2_point(LaserData lidar_measurement, RobotPosition robot_position)
{
        Point P;
        P.X = robot_position.coordinates.X + lidar_measurement.scanDistance / 1000 * cos(-
deg2rad(lidar_measurement.scanAngle) + robot_position.alfa);
        P.Y = robot_position.coordinates.Y + lidar_measurement.scanDistance / 1000 * sin(-
deg2rad(lidar_measurement.scanAngle) + robot_position.alfa);
        return P;
}
```

Generovanie trajektórie v známej mape - bludisku

Výsledkom tejto úlohy je získanie množiny bodov trajektórie (z bodu štartu do bodu cieľa) bezkolízne, v známej mape s prekážkami, a to čo s najkratšou výslednou dĺžkou dráhy.

Využívame takzvaný záplavový algoritmus, ten pracuje nad našou mriežkovou „vybudovanou“ mapou. V princípe algoritmus funguje tak, že vybudovanú, prípadne načítanú mapu ohodnotí. Štart algoritmu - prekážky v mriežkach číslo 1, cieľová bunka má číslo 2 a voľný priestor 0.

Ako ďalšie sa iteratívne ohodnotia neohodnotení susedia aktuálnej pracovnej bunky - inkrementálne. Začína sa od cieľa. Bunky zaznačené ako prekážka sa neohodnocujú a ponechávajú si hodnotu 1. Postupné ohodnocovanie všetkých buniek môže byť podľa implementácie realizované stackom, alebo v našom prípade frontou. Ďalej je možné zvoliť si susednosť : 4 (do kríža) alebo 8 susednosť (aj diagonálne).

Na takto ohodnotenej mriežkovej mape algoritmus pokračuje výberom vhodnej trajektórie.

Tentokrát sa začína od štartu. Možností je opäť viacero: V prvom rade sa uprednostňujú bunky s najmenšou hodnotou (najväčší gradient), v prípade zhodnosti hodnôt sa vyberá buď podľa priority v smere X alebo v smere Y.

Keďže nechceme, aby robot zastal v každej bunke vybranej trajektórie, tak sa do výslednej trajektórie zapíšu iba jej zlomové body (algoritmus využíva vlastnosti prvej a druhej diferencie indexu trajektórie)- teda také body, kde robot mení smer.

Problematika tesného obchádzania prekážok:

Nakoľko pôvodný záplavový algoritmus plánuje optimálnu trajektóriu v tesnom okolí – susednosti s prekážkami, je ale z praktických dôvodov vhodné obchádzať prekážky s určitou rezervou vzdialenosti. Riešenie tohto problému je implementované rozšírením buniek s prekážkami o určitú šírku (rozšírenie prekážok) – jedná sa tak o formu predspracovania mapy pred samotným záplavovým algoritmom (metóda *enhance_obstacles*).

Samotné použitie spomínaných metód vyzerá nasledovne:

```
start = actual_position.coordinates;
        int window_size = 5;
        bool result = false;
        do
        {
                clear_path();
                Mapa mapa_flood_fill(working_map, true);
                mapa_flood_fill.enhance_obstacles(window_size);
                mapa_flood_fill.FloodFill_fill(start, target, true);
                map_with_path = Mapa(working_map, true);
                result = mapa_flood_fill.FloodFill_find_path(map_with_path, start, target,
floodfill_priority_Y, path, true);
                window_size--;


        } while (result == false&&window_size>=0);
```

Užívateľská príručka

Aplikácia pre riadenie mobilného robota bola vytvorená pre použitie s operačným systémom Windows a využíva knižnicu pre grafické zohranie - Qt. Po spustení programu sa nám zobrazí užívateľské prostredie, ktoré je znázornené na obrázku nižšie. Význam jednotlivých prvkov si vysvetlíme postupne v ďalších častiach dokumentácie.

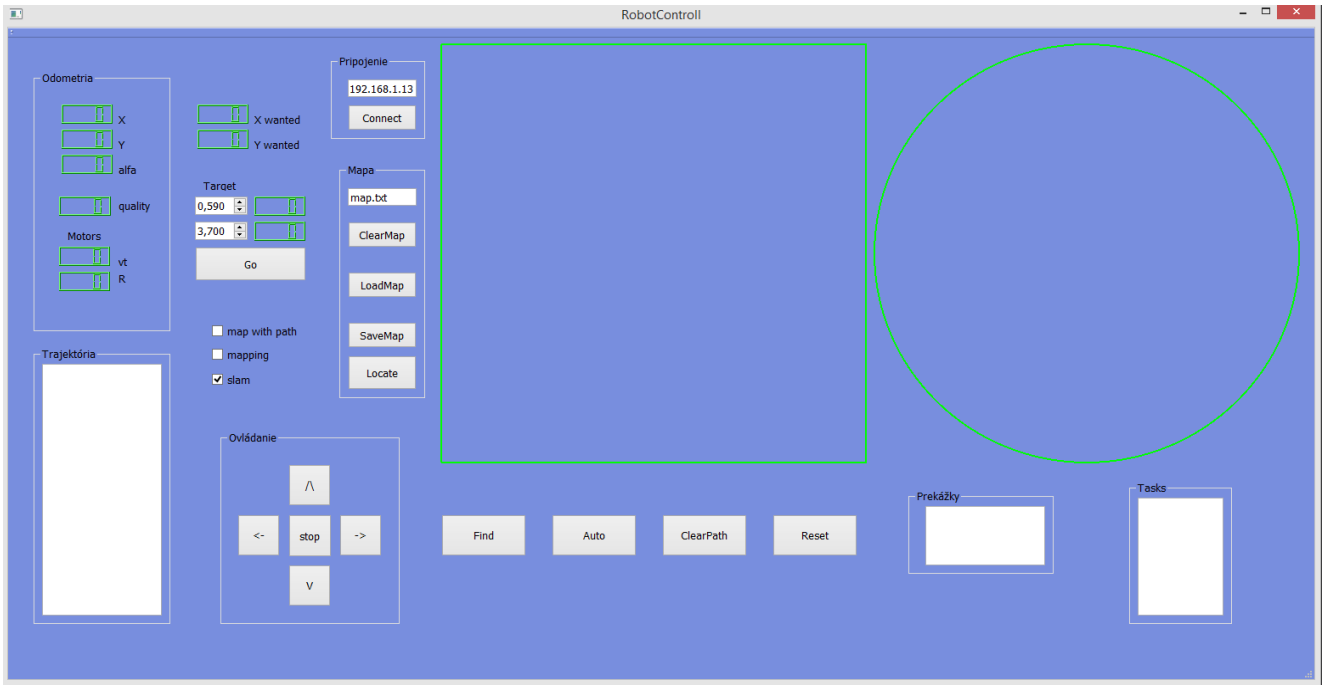

Figure 1 Užívateľské prostredie

Všeobecné bloky

Medzi takzvané všeobecné bloky sme zaradili bloky na ovládanie robota. Patrí medzi ne napríklad pripojenie aplikácie k robotu na základe zadanej IP adresy. Adresu, ku ktorej sa chceme pripojiť zadáme do okienka a stlačíme *connect*.

Robot môže byť v ručnom režime ovládaný šípkami. Ak si ho želáme zastaviť z ľubovoľného režimu, môžeme tak urobiť stredným tlačidlom stop. Takto robota vieme ovládať manuálne. Ak by sme ho chceli ovládať automaticky využijeme blok *target*. Doň si vieme šípkami navoliť cieľové súradnice X a Y v priestore, kam sa má robot dostať.

Lokalizácia

Lokalizáciu robota môžeme vidieť v bloku nazvanom *odometria*. V ňom samotnom sú indikátory aktuálneho stavu robota a teda jeho aktuálne súradnice v priestore a uhol natočenia robota. Hneď vedľa sa nachádzajú ukazovatele žiadanej hodnoty robota v priestore. V ďalšom bloku môžeme nájsť ukazovateľ kvality odhadu polohy (SLAM) *quality*, rýchlosť *vr* a polomer otáčania *R* generovaný regulátorom.

Vykresľovanie mapy

Grafické rozhranie implementuje vizualizáciu dvoch máp:

globálnej a lokálnej.

Globálna mapa (štvorec) obsahuje celý priestor zaznamenanej mapy spolu s polohou robota a prekážkami.

Lokálna mapa (kruh) obsahuje živý náhľad aktuálneho laserového scanu prostredia v okolí robota.

Práca s mapou je možná v bloku *Mapa*. Do prvého políčka vložíme názov súboru, z ktorého chceme mapu načítať alebo do ktorého chceme zapisovať. Tlačidlom *ClearMap* mapu vyčistíme, *LoadMap* načítame a *SaveMap* uložíme. Tlačidlom *Locate* vieme ručne spustiť lokalizáciu robota v známej mape (SLAM). Check box *mapping* povoľuje samotné mapovanie – teda budovanie mapy. Check box *map with path* prepína medzi zobrazením klasickej mapy a mapy obsahujúcej plánovanú trajektóriu a jej body.

Generovanie ciest v bludisku

Pri generovaní ciest v bludisku využívame viacero blokov. Tlačidlom *Find* vieme spustiť záplavový algoritmus a nájsť trajektóriu so štartom v aktuálnej polohe a cieľom nastaveným v spin boxoch *Target*. Samotnú realizáciu prechádzania naplánovaných bodov spustíme tlačidlom Auto a kedykoľvek prerušíme zatlačením *Stop.*

Do zoznamu *Trajektória* sú po úspešnom naplánovaní vypisované body trajektórie, ktoré zostávajú neprejdené (fronta bodov). Vizualizácia takto získanej trajektórie je možná check boxom *map with path.* Kedykoľvek je ale možné naplánovanú dráhu vymazať stlačením *ClearPath*.

Prekážky

Prekážky detegované laserovým skenerom sú automaticky v systéme aktualizované a ich počet je vypísaný v okne *Prekážky*. V prípade, že je robot v autonómnom režime a vykonáva svoje úlohy, sú prekážky v kolíznom kurze automaticky obídené úpravou trajektórie. V prípade ručného riadenia sa prekážky rovnako detegujú ale k úprave trajektórie ani k zastaveniu nedochádza.

SLAM

Možnosť využitia presnejšej lokalizácie robota pomocou údajov z laserového skenera je taktiež implementovaná.

Pre zapnutie tejto funkcie treba označiť CheckBox SLAM. V tomto prípade sa na indikátore *quality* zobrazí kvalita aktuálneho odhadu polohy (hodnoty 0-1). Ak je táto funkcie zapnutá, v každom bode trajektórie dôjde k zastaveniu a dolokalizovaniu polohy robota. Lokalizácia prebieha, až pokiaľ kvalita nepresiahne akceptovateľnú úroveň.

Rovnako je mapovanie aktívne iba ak je kvalita odhadu dostatočná.