Masaryk University
Faculty of Informatics

Schpin – quadruped control software

Bachelor's Thesis

Jan Koniarik

Brno, Fall 2016

Masaryk University
Faculty of Informatics

Schpin – quadruped control software

Bachelor's Thesis

Jan Koniarik

Brno, Fall 2016

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jan Koniarik

Advisor: RNDr. Zdeněk Matěj Ph.D.

Acknowledgement

This is the acknowledgement for my thesis, which can span multiple paragraphs.

Abstract

This is the abstract of my thesis, which can span multiple paragraphs.

Keywords

Robot, Quadruped, ROS

Contents

List of Tables

List of Figures

1 Introduction

infill

PART I

LOCOMOTION ALGORITHM

2 Introduction

Locomotion algorithm uses legs of robot to execute motion based on input, which is path through which robot should move. Either user or another algorithms can define that path. Algorithm than commands legs of the robot to make proper movement.

Note that we don't assume that robot has only 4 legs. While this is actual configuration, algorithm is designed in a way that should allow for any number of legs. Making it into proper centipede would be really interesting, however, for purpose of this thesis, we stick to 4 legs only.

Task is split into two phases: planning and execution. Presence of modular system is crucial for proper development. (Advantages of such a system are noted in [1]) For planning, we use modular system to process received data. As for execution, modular system is periodically executed. These parts should implement system that allows for simple addition (or removal) of factors to consider - modules.

In planning phase, main path for robot is processed and sidepaths for each leg are generated. These sidepaths serves as potential surface to step on for each leg.

Execution phase than analyzes actual sensory data and generates paths for body and legs. These paths are then used for movement and starts at current location. Note that paths are actually re-generated each cycle, that harms performance, but will be solved in later development.

For entire process, one big data structure is created. Every module has access to it and is free to modify it at will. Data structure contains static data for the process of execution. Apart from that, place for storing dynamic data is created, to allow of sharing data between modules that is not defined in static part of structure. History is kept for this data structure for modules to use. But we don't expect to remember more than the last one for now.

Static part od data structure will be properly documented and got wide array of tools. Adequate visualisation and analysis is required to ease debugging. For dynamic part of data structure, we can only provide universal tools for such a parts. Keep in mind that universal tools won't perfectly fit the data and will be second rate.

Configuration of robot is required. Especially number of legs, and values relevant to them, such as preffered direction of lift. These values should be loaded from central definiton of robot shape. ROS gives us tools for this, so it's just question of proper usage.

Each module than should implement at least one of the following API:

Initial Module is executed over module once, at start of executing that group of modules.

Cyclical Modules are executed for each item in container. After all modules finish, next item in container is used.

Finish Module is executed over module once, at end of executing that group of modules.

After this intro, we print glossary of terms necessary for understanding the algorithm.

Locomotion glossary

point 3-axis position in cartesian coordinate system - [x,y,z].
orientation Orientation in 3D space represented by quaternion.
pose Combines orientation and point.
waypoint Goal for aglorithm, used as input from other sources.
- **goal pose** Goal pose of specific waypoint. To accept specific waypoint, real position has to be within treshold.
- **legs in the air limit** Maximal number of legs in the air at given moment.
- **accept tresholds** Tolerance to accept waypoint as passed.
- **speed limit** Maximal speed robot should move through the waypoint. Given the chaotic nature of movement and practicall unpredictabillity, we avoid using concrete speed settings entirely.

main path Ordered collection of waypoints which robot should move through. Robot does not use this path as movement, but it's used to generated paths for actual movement. All paths are generated "around" this main path.
body path Path of poses, which body should move through. Also contains estimate of acting forces. This is not the same as main path, which define path for robot as whole.
sidepath Path of points on the ground, on which leg can step on. Having defined explicit potential space to use for legs significantly eases locomotion.
leg path Path of points which should leg move through. Also should contain support estimate.

> Note: What exactly should support mean needs more elaboration. But we know that '0' means 'no support at all', which is crucial for us. Practically, we can just use 0/1 values until better solution is picked.

configurable variable Variables that are configurable by user at first phase, lately they should be modified by self-learning mechanism.

distance functions These functions should return relative distance on path for main path, body path, sidepath and leg path. Distances should be directly comparable. Values should reasonably detect order of legs and if leg is in front of body or behind body.

2.0.1 Synchronization

For all 4 types of paths and time estimates, we require synchronizaiton. This algorithm works on presumption that for every item on any path. You can find relevant item on all of other paths. With exception of body path and leg paths, which are generated only from actual location of robot.

Given that all paths are either directly or undirectly generated from main path. We use that connection as synchronization element.

Because we can synchronize all paths and because they are linear spaces in nature. We can base distance function on this property of paths.

Basic requirement for distance function is comparability of paths. For any given items from any two paths that we want to compare. We can replace item from one path, for adequate item on second path (due to synchronization).

Items from same path are then compared. Given that we just need to figure out their order on path and distance, it's trivial matter then.

2.0.2 Configuration

Given that we target this system for ROS, for developing phase we will use provided configuration system. All parameters of algorithm will be configured by this internal system.

As for longterm development. There is space for replacing that configuration system with neuron network. This needs means of rating the algorithm, so neuron can evaulate the process.

But that is meaningfull only after we can setup working manual configuration.

3 Input

When new data arrives, modular system is executed. Input is received as sequence of waypoints for main path. As robot moves through waypoints, they are reported as done and dropped from memory. There is interface to drop already given waypoints, so algorithm must take this into account between movement cycles. (For now everything is only asynchronous and not parallel, so it's just proper algorithmization)

Waypoints should have sequence number defined from originator of path. Some ID of originator should also be stored for each waypoint. Mechanism for locking locomotion to only one originator at time is necessary.

Interface for new data should work with two actions:

Add New items are added at the end of the container.

Drop All items after specified sequence are droped.

> Note: Reason for the "drop" phase is for cases when sudden change of movement is required.

> Note: Number of waypoints should be carefully considered, each waypoint means potentially high amount of calculations. (although it's still $\mathcal{O}(n)$ for now) But, we still need enough of them for proper movement.

3.1 Input modular system

This modular system is executed on both Add and Drop actions. Adequate steps for each action are allready performed.

mod. 1 – Start In case actual path is empty. Inserts actual pose at start of the path. Configuration is copied from next waypoint.

mod. 2 – Process main path In case it's needed, main path is processed.

> Note: In our implementation, we infill another poses into main path.

mod. 3 – Generate Sidepaths Sidepaths is generated for each leg. These are used as surface on which leg can step on.

For now we use only lines to define such a surface. But in long run it could make sense to move to more complex structure. Such as area with density function. For now line should work just good enough.

While generally we assume that -Z axis is ground, so we always want to align in that direction. We should also think about more exotic use cases. Consider for example robot that wants to climb on glass of your window. (seen that somewhere) For that robot, relevant direction of putting legs, is perpendicular to plane of that glass.

Therefore it would be bad idea to hardcode that we always want to land legs in -Z direction.

Note that in case 3D map of surrounding area changes. This algorithm should be notified and sidepaths should be recalculated.

> Note: Practically, notification of changes will require reporting what areas relevant for this algorithm should be watched. Reporting everything that changs in radius of 'r' from main path should work. Where 'r' is big enough constant based on size of the robot.

Each module should modify same container with data. But, optimalisation modules requires sharing limits to ammount of modification they do. For that purpose, separate container with sum of just optimalisation changes should exist.

All modules are executed in initial phase.

mod. 3.1 – Copy main path Main path is copied as base dataset.

mod. 3.2 – Static offset Each leg has configured offset from body position where it should be in stable position. (Using configurable variables) This offset is than applied to each pose on main path, rotated based on pose orientation.

mod. 3.3 – Speed optimalisation Speed limit makes areas with small speed limit wide and fast areas narrow. Effect of this should be configured by configurable variables.

mod. 3.4 – Shape optimalisation Straight areas are more narrow and curvy areas are wider.

> Note: Could be detect by second derivative of main path: $\sum_{x=a}^{b} P_{main}''(x)$ Where a,b are area boundaries.

mod. 3.5 – Vector alignment Final point is aligned by defined direction vector to nearest rigid thing. ($[0, 0, 1]$ - gravity direction as vector for now)

> Note:

mod. 3.6 – 3D Map correction In case 3D map exists, legs are shifted on the ground to more stable positions. This module should be subject of later development.

> Note: This module won't be implemented.

4 Execution

Execution itself is represented with loop running at specific frequency. For development phase the frequency is 50Hz, but for fast movement higher will be required. Loop itself is modular system. Each module has ability to stop the cycle.

At the beginning of cycle, actual state is analyzed together with previous step. Body path and leg paths are then generated for rest of the path. Based on generated data, movement is executed.

In many cases, we found out chicken-egg problem. Leg paths are generated based on body path. But that itself should be generated based on leg paths.

This problem is solved by giving body path priority in generation. Body path is generated first and leg paths are based on body path. In second cycle, body path uses leg paths from previous cycle to balance itself. We assume that there should be small changes in paths between cycles in case environment doesn't change.

Also, even with 50Hz, we should get quickly to state where paths change only slightly.

4.1 Execution modular system

Main execution body works only in Initial phase of modules for now.

4.1.1 Load Sensors

Module takes care of receieving data from sensors. These data should be than copied into central data structure when this module is executed.

- Body pose
- Body velocity
- Body momentum
- Legs position
- Legs load

> Note: all data also contains covariance matrix

4.1.2 Pick path items

Picks actual relevant path points. (In case that timestamp exists)

Body path pose picks closest pose. Distance is calculated as weighted sum of position distance and angular distance.

Main path pose picks pose based on body path pose. There is direct relation between body path poses and main path poses.

Sidepath point picks point based on position distance.

Leg path point picks closest point based on position distance. Uses point on sidepath in case leg path itself is missing.

4.1.3 Mark as passed

We start from previously passed items on main path. Iteration checks each item, and if we are within it's accept range, it's marked as passed. At first item that is not marked as passed, this marking ends.

4.1.4 Drop old data

Old parts of paths are dropped. However parts should remain in memory for specific amount of time. That amount should be managed by configurable variable.

4.1.5 Calculate Errors

Calculate errors between state of the robot and generated paths.
Time differences are compared based on time estimates.

Value	Path item	Error type
Body position	Body path pose	Position distance
Body orientation	Body path pose	Angular distance
Body time error	Body path pose	Time difference
Leg position	Leg path point	Position distance
Leg time error	Leg path point	Time difference

4.1.6 Error watchdog

In case body got too big position error, main path is dropped and callback is issued to 'requestor' of given path.

It's than regenerated based on actual data, and this modular system is restarted.

4.1.7 Setup actual location

Based on picked pose on main path, actual location is picked.

4.1.8 Generate time estimates

Time estimates should use analysis of previous cycles to assert time aspects of next movement. We assume that next paths should be similar enough to previous parts. But for first phase, we use simple generation:

Setup time stamps, so it represents max speed for waypoints on main path. For development phase, these max speed should be set on values, that are definetly within capabilities of the robot.

TODO. . . this has to be relevant against actual main path pose.

4.1.9 Generate body path

Modular system is used for body path generation.

This modula system works over dataset, where poses can be marked as immutable. Anything that is marked immutable is not modified by all modules and can be ignored in calculations.

Initial modules are:

mod. 1 – Copy main path Main path is copied as basic dataset. Note that we copy only from actual location up to the end.

mod. 2 – Setup start First body path pose is moved to actual position.

mod. 3 – Process feedback Data from accelerometer (if present) are incorporated into start pose acting forces.

mod. 4 – Lock start Start pose is marked as immutable.

mod. 5 – Add Gravity Gravity effect is added to acting forces of items.

mod. 6 – Leg paths If leg paths are not in previous data set, this module skips.

Weighted average of leg positions is calculated for each pose on basic dataset. Leg 'support' capability is used as weight of each leg position. (We will consider support capability to be 1 for this thesis.)

Existing acting forces vector is used. We project weighted leg positions average on plane that contains actual body pose. Weight for those position is actual load for each leg. Projection is made in direction of forces vector. Plane is picked as plane that is perpendicular to force vector and contains body pose. // draw this

That average is than stored as magnet point for that pose. For each magnetic point, we also remember next following magnetic point.

For each pose, we store sequence number of that pose for that magnetic point.

> Note: Based on actual forces and speed, we could add another force estimation for entire path. But that is subject of later development, so it was avoided for now.

After init phase is done, cyclical phase begins, that phase actually sets the pose itself:

mod. 1 – Calculate goal Goal is calculated as weighted average of actual magnetic point and next magnetic point. Where weight for that average is actual pose sequence index for actual magnetic point, and it's reverse for next magnetic point. So at start of the part relevant for specifig magnetic point, goal is the magnetic point itself. Goal than lineary shifts to next magnetic point and at the end of part, it's actually next magnetic point.

$\frac{iM_{actual}+(i_{max}-i)M_{next}}{2i_{max}}$

mod. 2 – Direction PID PID will be aplied between actual direction and direction towards goal (Goal - actualposition).

This pid than affects final pose where body should move.

This should end up with body path, that shifts between stable zones which are defined by legs.

4.1.10 Calculate instability

Instability is determined by one value. There are several factors which should be implemented as modular system:

mod. 1 – Sum of leg position distance errors
$\sum_{i=1}^{leg_n} E_{dist}(L_i)$

mod. 2 – Body position distance error
$E_{dist}(B)$

mod. 3 – Body angular distance error
$E_{angle}(B)$

mod. 4 – Body path shape
Sum of of second derivatives for local body path $\sum_{x=-l}^{l} P_{body}''(x)$

mod. 5 – Sidepath shapes

> Note: If leg paths are not present, this module is ommited

Sum of sums of second derivatives of local leg paths $\sum_{i=1}^{sidepath_n} \sum_{x=-l}^{l} P_{sidepath_i}''(x)$

> Note: Leg path shapes and body path shape, can't be replaced by main path shape. Leg path shape can eventually be affect by 3D map of terrain, which is more relevan than actual main path shape.

$$I = a_0 \sum_{i=1}^{leg_n} E_{dist}(L_i) + a_1 E_{dist}(B) + a_2 E_{angle}(B) + a_3 \sum_{x=-l}^{l} P_{body}''(x) + a_4 \sum_{i=1}^{sidepath_n} \sum_{x=-l}^{l} P_{sidepath_i}''(x) \quad (4.1)$$

Where:

I final instability

a_{0-4} configurable coeficients for each component

l configurable variable that defines distance range of 'local' areas of paths.

> Note: distance used on paths are from distance functions

4.1.11 Generate leg paths

Leg paths uses modular system for generation.

All module share key data structures. Especially important are groups of legs that are to be considered in same phase.

For each leg, we track direction in which it moved from last time. (There should be history between cycles)

Swing of leg is achieved via this direction. Leg next position is defined based on actual direction and variables add to that direction. Practical effect is that leg should move from sidepath and slowly change it's path back to it. There are three relevant configurable variables for this concept.

forward coeficient How far should leg move forward in case it's in the air in each step

start lift coeficient Defines how far should leg move from the sidetrack when move begins

lift change coeficient Defines how should the distance leg moves from sidetrack change between each step.

Idea is, that at start leg is moving big distances from sidepath, until that distance became negative and leg starts to move towards the sidepath.

Modules should be executed in cycle over body path.

mod. 1 – Sidepath distance Sidepath distance for each leg is calculated and stored alongside.

mod. 2 – Select groups Legs are split into groups based on variances between distance. We want to keep variance in each group low, but variance between groups high. (proper algorithm has yet to be designed)

Only legs that are touching ground are splitted into groups.

mod. 3 – Calculate maximal number of legs Smallest value from all modules is picked as maximal number of legs in the air.

> Note: 0 is valid value too

- **mod. 3.1 – Number of legs** 'n' as number of legs on the robot.
- **mod. 3.2 – Instability limit** 0 or 'n' if instability is below certain configurable thresholds.
- **mod. 3.3 – Error limit** 1 if body and leg errors are above certain tresholds. (this may be merged with module above)
- **mod. 3.4 – Waypoint limit** 'n' from maximal number of legs in the air for actual waypoint configuration.
- **mod. 3.5 – Body position** 0 in case actual average of leg distances is bigger than body distance.

  > Note: Note that in case robot is standing and all distances are 0. And there is no body path that results in actual movement. Robot will be stuck, basic way of getting around this is to introduce tolerance for this module.

> Note: Using this as 'float' version could have various implication, 1.5 -> one leg is 50

mod. 4 – Pick desired leg order and lift group Leg order defines in which order legs should be lifted in case instability is low enough. (details later) Lift groups defines legs that should be lifted together.

This chapter of algorithm is undeveloped for now. We got prepared leg order that should be good enough for beginning and fast bount leg order that is able to stress algorithm to it's limits.

Modules are:

- If desired speed is high enough and there can be two legs at a time, switch to bount leg order.
- Use manualy picked leg order.

mod. 5 – Pulldown extra legs In case there are more legs in the air that is actual limit. Pick leg that will be the most easier to put down and set it's destination vector, so it arrives at sidetrack ASAP.

mod. 6 – Continue leg movement For each leg that is in the air, create new direction vector.

These modules executes in Initial phase.

mod. 6.1 – Touchdown In case leg is close to sidepath, initiate touchdown process. Leg will go down until it supports robot.

If that happens, this module system is stoped.

mod. 6.2 – Copy direction Copies vector from previous step.

mod. 6.3 – Group landing point Based on actual leg groups, picks group that is closest to actual direction of leg.

Landing point is than center of that group.

mod. 6.4 – Shift to landing point Based on actual history of change, we should be able to guess where leg should land. Based on that we should be able to correct it so it lands in picked landing point.

mod. 6.5 – Shift to leg reach Based on actual leg reach, direction is changed, so it aims towards center of leg.

For this to work correctly, size of this correction, has to be exponentialy relative to distance of leg from it's center.

mod. 7 – Move leg up In case leg can be pulled up. We choose group with smallest ditance. Within the group, we pick next leg based on desired leg order.

That leg is than moved based on start lift coeficient up.

For now, we pick direction, as perpendicular vector to sidepath. That leaves us with 'n' vectors rotated around direction of sidepath.

Precise direction of vector should be picked based on prefered direction for each leg.

mod. 8 – Keep legs Each leg that did not changed it's position during generation is set to stay on actual point.

4.1.12 Prepare movements

Based on cycle frequency, prepares where exactly should each leg and body move. So it corresponds to time estimate.

4.1.13 Execute movements

Movements are executed.

5 Implementation

For simplicty of claculations, we use containers of items as storage for all paths. These countainers should always have same number of items. We can than work with assumption that i-th item in one container corresponds to i-th item in second container.

If two waypoints on body path are too far away from each other, that space is problem for fluent movement. Because of that we lineary interpolate another poses between waypoints. Rule that no two poses on robot path can be 'x' distance from each other and 'y' angular distance from each other, than servers as definiton about how many poses we need to interpolate.

This creates a lot of data, but makes any math that will be calculated over that dataset really simple. Given that this part should only have data about what is necessary to know. We can assume that body path won't have more than 2 000 items in long run.

For instance, if we know 20cm of path and have maximal distance of point 1mm. This will create list of 200 poses, which is small amount of data.

5.1 Main interface

Idea is that main.cpp should be easily modifiable for addition of new modules. // rewrite this bullshit

```
#include "ros/ros.h"
#include "schpin/locomotion.h"

using namespace schpin::modules;

int main(int argc, char **argv) {

  ros::init(argc, argv, "locomotion");

  // NOTE: all modules are configured via ROS conf
  // managment which is completely separate from
  // actual code here

  // setups modules to be used for generation of sidepath
  setupSidepathModules(
      sidepath::Module1(),
      sidepath::Module2(),
      sidepath::Module3(),
      sidepath::Module4(),
      sidepath::Module5()

      );

  // setups ROS-topic communication
  // for receieving data
  setupDataInput();

  // setups frequency of main cycle
  setupCycle(50);

  // setup modules used in main cycle
  setupCycleModules(

      cycle::Module1(),
      cycle::Module2(),
      cycle::SpecificModule(
          specific_module::Module1(),
          specific_module::Module2(),
          specific_module::Module3(),
          specific_module::Module4(),
          specific_module::Something(
              something::Module1(),
              something::Module2(),
              something::Module3(),
            ),
        ),
      cycle::Module4(),
      cycle::Module5(),

      );
  // run infinite cycle, managet by ros
  runCycle();
}
```

5.2 Debugging

For debugging purpose, we need means of analyzing central data structure. Most of the data will be 3D visualized.

During 'debug' state. Central data structure will be send over ROS network after execution of each module.

Given that ROS can record any communication and replay it. We can than step by step see how modules change central data structure and what is going on.

That is crucial for development of this mechanics. Also proper logging inside all of modules will be used. Given that we got full logging system with multiple levels of priority. We can use approach: better log everything rather than to miss something. With proper distribution into level hiearchy, this should endup with sane output.

Bibliography

[1] Christian Gehring et al. “Control of dynamic gaits for a quadrupedal robot”. In: *Proceedings - IEEE International Conference on Robotics and Automation* (2013), pp. 3287–3292. ISSN: 10504729. DOI: `10.1109/ICRA.2013.6631035`.

A An appendix

Here you can insert the appendices of your thesis.