UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fachgruppe Algorithmen und Komplexität
Projektgruppe Schlaue Schwärme

May 3, 2009

User's Guide

Alexander Klaas
Andreas Cord-Landwehr
Christoph Raupach
Christoph Weddemann
Daniel Warner
Daniel Wonisch
Kamil Swierkot
Marcus Märtens
Martina Hüllmann
Peter Kling
Sven Kurras

Contents

1. Beginning

1.1. Introduction

This document shall help you to use the RobotSwarmSimulator to simulate robot swarms and to get useful information by the visualization and statistics modules.

For this purpose the document is divided in several sections. It will be useful if you just start with the "Getting Started" section and play a little bit with the simulator and continue by reading the more detailed information on input parameters, input file specifications, generating simulations and getting statistics.

1.2. Getting Started

If you are using the RobotSwarmSimulator the first time, we will present you a simple way to create an example scenario to get a feeling about the usage of this simulator. All example files can be found in subdirectory `ProjectFiles` of the install directory. Here we will present an example on how to simulate the behavior of the Center of Gravity algorithm (COG).

1. Change to the directory that contains the RobotSwarmSimulator binary.
2. Run the following command: `RobotSwarmSimulator --generate --distr-pos 20 --add-pos-handler \\ --robots 1001 --algorithm COGRobot` This will generate a simulation specification in form of the following files:
   - `newrandom.swarm` – contains information about the simulation process
   - `newrandom.robot` – contains information about the robots
   - `newrandom.obstacle` – contains information about the obstacles

   The last two files are referenced in the `.swarm` file. Thus, to load the simulation, you only need to load the `.swarm` file. Note that all these files are simple text files that can be edited by hand.

3. Run the command:
   ```
   RobotSwarmSimulator --project-file newrandom --output mylogs
   ```
   This will start the simulation process. There are various keyboard shortcuts that can be used to control the simulation. Press `h` for an overview. You can quit the simulation by pressing `q`.
4. The previous step has generated various output files in the subdirectory `mylogs`, mainly produced by the statistics module of the RobotSwarmSimulator. You may directly use gnuplot to analyze these files.

This simple example can be used as a base for further test runs. Please look at the following chaptors to get specifications of the input files and the user inferface.

2. Setting up a Simulation

You can use the RobotSwarmSimulator to create simulations to test the behaviour of your algorithms. The algorithms can be written in C++ or in Lua. Also there are several mandatory files for the simulation. These are the main projectfile, the robot file and the obstacle file. All of these files can be generated automatically by using the `--generate` mode of the RobotSwarmSimulator. For finer settings please have a look at the following sections and Appendix B.

2.1. Create Simulation environments

All global settings of your simulation can be defined in the main projectile (ending: `*.swarm`). This includes the setup of the ASG, Vector Modifier, Robot Control and the according View. Please refer to the following sections for a detailed description.

Activation Sequence Generators

There are two Activation Sequence Generators (ASGs). A synchronous ASG and an asynchronous ASG.

To use the synchronous ASG one only needs to set the variable `ASG=SYNCHRONOUS`. No further variables need to be set.

To use the atomic semi synchronous ASG one needs to set `ASG=ATOMIC_SEMISYNCHRONOUS`. Furthermore one needs to set the following variables:

- `ATOMIC_SEMISYNC_ASG_SEED`: Seed for the ASG

To use the asynchronous ASG one needs to set `ASG=ASYNCHRONOUS`. Furthermore one needs to set the following variables:

- `ASYNC_ASG_SEED`: Seed for the ASG
- `ASYNC_ASG_PART_P`: The higher this is, the more robots are activated for each event. Goes from 0.0 to 1.0.
- `ASYNC_ASG_TIME_P`: The lower this is the smaller is the time difference between events. Be careful with very high values as buffer overflows might happen. Goes from 0.0 to 1.0.

Event Handler

The event handler is responsible for applying actual changes to the simulated world. In fact, this is the *only* place where the world may be changed. An event handler is mainly specified by so called *Request Handlers* that specify how the different requests robots can issue are handled. For example, to always apply position requests from the robots in exactly the way they were issued, you would set `POSITION_REQUEST_HANDLER_TYPE=VECTOR` and additionally set the following configuration variables:

- `VECTOR_POSITION_REQUEST_HANDLER_DISCARD_PROB="0.0"`: No event will be discarded.
- `VECTOR_POSITION_REQUEST_HANDLER_SEED="42"`: Seed for random number generator.
- `VECTOR_POSITION_REQUEST_HANDLER_MODIFIER=""`: We don't need any vector modifiers in this example

The following list provides a short overview over more advanced and non–standard request handlers. For further explanation of all possible values see section 2.1 and Appendix B.

- **Collision Position Request Handler** This is a special position request handler that provides simple collision handling functionallity. To use it, you have to set `POSITION_REQUEST_HANDLER_TYPE=COLLISION`. The possible options are listed in Appendix B. The two most important (new) configuration options include:
  - `COLLISION_POSITION_REQUEST_HANDLER_STRATEGY` You may choose between the values `STOP` and `TOUCH`. While the strategy `STOP` will cause the position request to be discarded if it would cause a collision, the latter strategy `TOUCH` will move the robot from the requested position (where a collision occured) backwards towards it's original position until the collision is resolved (i.e., the robot will be touching at least one other robot after the collision was handled).
  - `COLLISION_POSITION_REQUEST_HANDLER_CLEARANCE` Use this value to choose the distance from that on two robots will be considered colliding. For example, if you have spheric robots of radius r, choosing a clearance of $2r$ will consider two robots colliding as soon as their spheres intersect.

Vector Modifiers

Vector modifiers are objects that change a given vector in a specific way. In addition to the to be modified vector, they may be given a so called *reference vector*. How exactly this reference vector influences the modification of the given vector is due to the respective implementation.

Such vector modifiers are used by vector request handlers (e.g. position request handler). Requests to these handlers consist of a three dimensional vector (e.g. position the robot wants to be "beamed" to). To model different aspects like inaccuracy or certain restrictions, vector request handlers may be given a list of vector modifiers. Before granting a request, every vector modifier in this list will be applied to the requested vector (e.g. the new position a robot requested) using the current value of the corresponding robot attribute (e.g. the robot's current position) as the reference vector.

The list of vector modifiers is a (not necessarily nonempty) list, i. e.

```
(VECTOR_MODIFIER_1);(VECTOR_MODIFIER_2);...
```

The order of the elements of this list is important. If no Vector Modifier shall be used for the corresponding Request Handler, then use `VECTOR_MODIFIERS=""`.

An element `VECTOR_MODIFIER_k` of the Vector Modifier list is a tuple, defined as follows:

```
VECTOR_MODIFIER_k=(VECTOR_MODIFIER_TYPE,VECTOR_MODIFIER_PARAM_1,VECTOR_MODIFIER_PARAM_2,..)
```

The number and types of paramters like `VECTOR_MODIFIER_PARAM_1, VECTOR_MODIFIER_PARAM_2,...` depends on the corresponding type of the Vector Modifier. Currently there are the following types of Vector Modifiers:

- VectorDifferenceTrimmer
- VectorTrimmer
- VectorRandomizer

I. e. the value of `VECTOR_MODIFIER_TYPE` needs to be `VECTOR_DIFFERENCE_TRIMMER`, `VECTOR_TRIMMER` or `VECTOR_RANDOMIZER`.

VectorDifferenceTrimmmer If `VECTOR_MODIFIER_TYPE` is equal to `VECTOR_DIFFERENCE_TRIMMER`, then the following parameters are expected:

1. `length` of type double: Vector request handlers that use vector difference trimmers will ensure that the robot's request does not differ too much from the current value of the robot's corresponding attribute. If the difference is larger than `length`, the robot's request is changed such that the difference's norm matches exactly `length` afterwards. For example, a position request handler using a vector difference trimmer with parameter `length` = 1 will ensure that the robot can move only inside the unit sphere around it's current position. If the robot requested a position outside of this unit sphere, it would be moved on the intersection of the unit sphere's boundary and the line from the robot's current position to the requested position.

I. e. an element of the `VECTOR_MODIFIERS`-list of type `VECTOR_DIFFERENCE_TRIMMER` may look like: `(VECTOR_DIFFERENCE_TRIMMER,5.2)`.

VectorTrimmer If vector modifier type equals `VECTOR_TRIMMER`, then the following parameters are expected:

1. `length` of type double: Every vector this vector modifier is applied to will be scaled such that it has length at most `length`. May be used for example by position request handlers to ensure that robots can not move out of the sphere of radius `length` around the origin of the global coordinate system.

I. e. an element of the `VECTOR_MODIFIERS`-list of type `VECTOR_TRIMMER` may look like: `(VECTOR_TRIMMER,10.0)`.

VectorRandomizer If vector modifier type equals `VECTOR_RANDOMIZER`, then the following parameters are expected:

1. `seed` of type unsigned int: Seed used by randomization.
2. `standard derivation` of type double: Derivation used by the vector randomizer. Applying this vector modifier to a vector will add a random vector distributed according to the multidimensional normal distribution with derivation `standard derivation` (i.e. every coordinate of the random vector is distributed according to N(0,`standard derivation`)).

I. e. an element of the `VECTOR_MODIFIERS`-list of type `VECTOR_DIFFERENCE_TRIMMER` may look like: `(VECTOR_DIFFERENCE_TRIMMER,1,0.5)`.

RobotControl

The `RobotControl` variable defines the class which should be used to control the robots (and in particular to control the views of the robots). Currently one of the following classes has be chosen:

1. `UNIFORM_ROBOT_CONTROL`
2. `ROBOT_TYPE_ROBOT_CONTROL`

Each class is explained in detail below. Note that each class expects certain class specific parameters.

UniformRobotControl This class assigns each robot the same view type. The concrete view type needs to be defined using a `VIEW` variable. The possible values for this variable (view types) are definied below (see 2.1). E. g. you may to assign each robot global view to the world using `ROBOT_TYPE_ROBOT_CONTROL="GLOBAL_VIEW"`.

RobotTypeRobotControl This class assigns each robottype the same view type. Therefore robots with different robot types may have different view types. Currently there are two robot types:

1. `MASTER`
2. `SLAVE`

To specify which view type should be used by each robot type, there must be variables of the form *RobotType*`_VIEW`.
The value of each variable has to be a view type (see 2.1). Note that the view type parameters are also distinguished using the *RobotType* prefix. E. g. you may specify

```
MASTER_VIEW="CHAIN_VIEW"
MASTER_CHAIN_VIEW_NUM_ROBOTS="5"
```

to set the view for master robots to a chain view allowing the robots to see five neighbor robots. Note that exactly one view type should be defined for each robot type.

ViewTypes

The view type of a robot defines its vision model. Whenever a view type is expected you may use one of the following values:

1. `GLOBAL_VIEW`
2. `COG_VIEW`
3. `CHAIN_VIEW`
4. `ONE_POINT_FORMATION_VIEW`
5. `SELF_VIEW`
6. `CLONE_VIEW`

Each view type is explained in detail below.

GLOBAL_VIEW Allows robots to see literally everything. There are no parameters expected.

COG_VIEW View model meant to be used for center of gravity algorithms, i. e. every robot can see every other robots position, velocity and acceleration. The coordinate-system and id of each robot is not visible. There are no parameters expected.

SELF_VIEW View model which allows robots to access every self-related information while disallowing to access any other information. There are no parameters expected.

CHAIN_VIEW View model meant to be used for robot chain related algorithms, i. e. every robot can see k neighbor robots position. Besides this no more information is visible. When using this view type you have to specify the variable $k \in \mathbb{N}$ using the parameter variable `CHAIN_VIEW_NUM_ROBOTS`.

ONE_POINT_FORMATION_VIEW View model meant to be used for one point formation algorithms, i. e. every robot can see every other robots position, velocity and acceleration only in a limited view radius r. The coordinate-system and id of each robot is not visible. When using this view type you have to specify the variable $r \in \mathbb{R}$ using the parameter variable `ONE_POINT_FORMATION_VIEW_RADIUS`.

CLONE_VIEW View model designed for the use with the clone movement algorithm. In this view model every robot can see every other robots position, velocity and id only in a limited view radius r. Also in this model robots can query for the time of their last look event. When using this view type you have to specify the variable $r \in \mathbb{R}$ using the parameter variable `CLONE_VIEW_RADIUS`.

2.2. Running the RobotSwarmSimulator with Parameters

Each execution of the RobotSwarmSimulator needs specific parameters that are mandatory. By running the RobotSwarmSimulator you have to specify at least, which kind of execution you want. There are two main options:

- By adding the parameter `--generate` the generation mode is started to create new input files. This will generate three different files having the suffixes `.swarm`, `.robot` and `.obstacle`. The most important of these is the `.swarm` file, which references the other two files.
- Using the parameter `--project-file` ⟨`your_input_file`⟩, the simulation mode is started. In this case, `<your_input_file>` should be the name of a file having the suffix `.swarm` (e. g. generated with generation mode).

There are other command line options that may be used to show help or about messages and to customize the generation or simulation mode. Start the RobotSwarmSimulator with the parameter `--help` to get an overview. A typical session may look like this:

```
./RobotSwarmSimulator --help
./RobotSwarmSimulator --project-file <path_to_testdata>/testfile_2 \\
                      --output <path_to_output_files> \\
                      --history-length 10
```

All options listed in Listing 2.1 can be used. Further information for this parameters can be found in the following sections. The definition of most of the parameters can be found in Table B.1.

Listing 2.1: RobotSwarmSimulator Helpline

```
localhost:~$ ./RobotSwarmSimulator --help

General options:
 --help              shows this help message
 --version           shows version of RobotSwarmSimulator
 --about             tells you who developed this awesome piece of software

Generator options:
 --generate                   switch to generator mode
 --seed arg (=1)              seed for random number generator
 --robots arg (=100)          number of robots
 --algorithm arg (=NONE)      name of algorithm or lua-file
 --swarmfile arg (=newrandom) swarm-file for output
 --robotfile arg (=newrandom) robot-file for output
 --obstaclefile arg (=newrandom) obstacle-file for output
 --add-pos-handler            add position request handler for testing
 --add-vel-handler            add velocity request handler for testing
 --add-acc-handler            add acceleration request handler for testing
 --distr-pos arg (=0)         distribute position in cube [0;distr-pos]^3
 --min-vel arg (=0)           distribute velocity in sphere with minimal
                              absolut value min-vel
 --max-vel arg (=0)           distribute velocity in sphere with maximal
                              absolute value max-vel
 --min-acc arg (=0)           distribute acceleration in sphere with minimal
                              absolut value min-acc
 --max-acc arg (=0)           distribute acceleration in sphere with maximal
                              absolute value max-acc
 --distr-coord                distribute robot coordinate-systems uniformly

Simulation options:
 --project-file arg      Project file to load
 --output arg            Path to directory for output
 --history-length arg (=25) history length
 --dry                   disables statistics output
 --blind                 disables visual output
 --steps                 if set this terminates the simulation after a given amount
     of steps
```

2.2.1. General options

--help Lists all possible options including a short description.

--version This option shows the version information of your RobotSwarmSimulator.

--about Get information about the developer team, contact information and more.

2.2.2. Generator options

--generate Switch to generator mode. This is necessary for the further options of this section.

--seed arg Sets the seed for the random number generator for robot generation. If not set the seed is `1`. An unsigned integer value is expected.

--robots arg The number of robots to be generated. The default number is 100. An unsigned integer value is expected.

--algorithm arg The name of the algorithm the robots should use. If not set the algorithm `SimpleRobot` is used. This is only a stub without any functionality. Also the name of a `Lua`-file can be given. The extension `.lua` is mandatory for lua-files.

--swarmfile arg The name of the swarmfile that shall be generated. Default is `newrandom`. Filename without extension is expected.

--robotfile arg The name of the robotfile that shall be generated. Default is `newrandom`. Filename without extension is expected.

--obstaclefile arg The name of the obstaclefile that shall be generated. Default is `newrandom`. Filename without extension is expected.

--add-pos-handler Causes the generated files to contain a position request handler with reasonable default values. If you need a more sophisticated position request handler, you have to edit the generated `.swarm` file yourself.

--add-vel-handler Causes the generated files to contain a velocity request handler with reasonable default values. If you need a more sophisticated velocity request handler, you have to edit the generated `.swarm` file yourself.

--add-acc-handler Causes the generated files to contain a acceleration request handler with reasonable default values. If you need a more acceleration request handler, you have to edit the generated `.swarm` file yourself.

--distr-pos arg Distributes the position of robots uniformly at random in the cube $[-arg/2, +arg/2]^3$ If not set, all robots are at position zero.

--min-vel arg The robots will be generated with a velocity distributed uniformly in a sphere with the given minimum and maximum absolute value (see `---max-vel`). This parameter defaults to 0.

--max-vel arg The robots will be generated with a velocity distributed uniformly in a sphere with the given minimum and maximum absolute value (see `---min-vel`). This parameter defaults to 0.

--min-acc arg The robots will be generated with a acceleration distributed uniformly in a sphere with the given minimum and maximum absolute value (see `--max-acc`). This parameter defaults to 0.

--max-acc arg The robots will be generated with a acceleration distributed uniformly in a sphere with the given minimum and maximum absolute value (see `--min-acc`). This parameter defaults to 0.

--distr-coord arg Generates uniformly distributed coordinate–systems for the robots. If this option is not given, all robots will have the same global coordinate system (defined by a unit matrix). If not set, all velocities are zero.

2.2.3. Simulation options

--project-file arg Specifies the project file use. Use this parameter to load a simulation specified by a `.swarm` file. Not that you may ommit the file extension `.swarm`. Mandatory for simulation.

--output arg Specifies a directory to be used for files generated by the simulation (e. g. statistics files for gnuplot). The given name is interpreted relative to the current directory and will create the directory if necessary. If ommitted, all generated files will be stored in the current directory.

--history-length arg Sets the history length, i. e. the length of the ringbuffer that stores past simulation states. Standard is 25 (which should be a good choice in most cases). An unsigned integer is expected.

--dry No statistic files are beeing generated.

2.3. Using the Simulator-Interface

During the simulation it is possible to interact with the simulation in different ways. The following hot-keys are supported while simulating:

`Space` Start/ Stop.

`q` Quit the RobotSwarmSimulator.

`F1` Help!

`g` Show the center of gravity of the swarm.

`v` Show velocity vectors.

`b` Show acceleration vectors.

`k` Show global coordinates system.

`w`,`s` In the corresponding camera mode use `w` for up and `s` for down.

Arrow-Keys Moves the view: left, right, before, behind.

[m] Use [m] to switch to mouse spinning mode and use your mouse to rotate the view. Note that this is not supported in every camera mode.

[+], [-] Increase/ decrease simulation-speed by constant.

[*],[/] Double/ half simulation-speed.

[c] Change camera mode.

[t] Switch skybox

[z] Show visibility graph

2.3.1. Information from Vizualisation

At the beginning of a simulation the camera view is directed to point `(0,0,0)`, if not explicitely specified. The axis, displayed when activated by pressing [K] are scaled with each unit equals to `2`. All presented robots always have diameter `0.15`, where the ball is defined with center equals to the roboter position.

2.4. Create Robot Algorithms for the RobotSwarmSimulator

There are two ways to define a new robot. One way is to write a subclass of `Robot` and add a new condition in `factories.cc`. The other way is to define the robot algorithm by the Lua scripting language and to load the algorithm at run-time. We want to stress, that a definition of robot algorithms by Lua scripts only scales for small numbers of robots. Thus, for robot swarms of sizes greater than 500 robots you will (on standard computers) recognize a lack of performance.

2.4.1. Create Robot Algorithms by Lua Scripts

For information on how to write Lua scripts please visit `http://www.lua.org` and use the documentation presented there. For interacting the environment Lua scripts may access the following functions and constants (if allowed by the current view and if the according request handlers are set):

Lua Functions

`get_visible_robots()` Returns the array of visible robots.

`get_visible_obstacles()` Returns the array of visible obstacles.

`get_visible_markers()` Returns the array of visible markers.

`get_position(<robot>)` Returns the position of the calling robot as a Vector3d.

`get_marker_information(<robot>)` Returns the MarkerInformation of the calling robot.

`get_id(<robot>)` Returns the identifier of the calling robot.

`get_robot_acceleration(<robot>)` Returns the acceleration of the calling robot as Vector3d.

`get_robot_coordinate_system_axis(<robot>)` Returns the coordinate system of the calling robot as a CoordinateSystem.

`get_robot_type(<robot>)` Returns the type of the calling robot as a RobotType.

`get_robot_status(<robot>)` Returns the status of the calling robot as a RobotStatus.

`is_point_in_obstacle(<obstacle>, <point>)` Returns true iff the given point of type Vector3d is within the given obstacle.

`get_box_depth(<box>)` Returns the depth of the given box as a double.

`get_box_width(<box>)` Returns the width of the given box as a double.

`get_box_height(<box>)` Returns the height of the given box as a double.

`get_sphere_radius(<sphere>)` Returns the radius of the given sphere as a double.

`is_box_identifier(<identifier>)` Returns true iff the given identifier is an identifier of a box.

`is_sphere_identifier(<identifier>)` Returns true iff the given identifier is an identifier of a sphere.

`add_acceleration_request(<Vector3d>)`

`add_position_request(<Vector3d>)`

`add_velocity_request(<Vector3d>)`

`add_marker_request(<marker>)`

`add_type_change_request(<type>)`

`get_own_identifier()`

Geometry Package

`Geometry.is_in_smallest_bbox(<vector of Vector3d>,<Vector3d>)`

`Geometry.compute_distance(<Vector3d, Vector3d>)`

`Geometry.compute_cog(<vector of Vector3d>)`

`Geometry.compute_cminiball(<vector of Vector3d>)` Computes the center of the miniball around the given points.

`Geometry.sort_vectors_by_length(<vector of Vector3d>)` Sorts Vector3d points by distance to origin

Lua Constants

`RobotType SLAVE, MASTER`

`RobotStatus SLEEPING, READY`

Special Variable Types

`Vector3d` This type is the Lua equivalent to Vector3d in RobotSwarmSimulator

- Operators: `+, *, /`
- Dimensions: `x, y, z`

`DistributionGenerator` This type is the Lua equivalent to the class `DistributionGenerator` in RobotSwarmSimulator

- Constructor: takes the seed as parameter
- Methods: `set_seed, init_uniform, get_value_uniform, ...` (everything that the `DistributionGenerator` class provides at public scope)

`MarkerInformation` This is the data type for marker information. The information can be accessed by the according operators:

- Operators: `add_data, get_data`

`CoordinateSystem` This type consists of three Vector3d objects. The axes can be accessed by the following methods:

- Operators: `x_axis, y_axis, z_axis`

Example Algorithm

Listing 2.2 shows you how to formulate the COG-algorithm in Lua.

Listing 2.2: COG algorithm in Lua

```
function main()
    robots = get_visible_robots();
    center = get_position(get_own_identifier());
    for i = 1, #robots do
        center = center + get_position(robots[i]);
    end
    center = center / (#robots+1);
    add_position_request(center);
end
```

2.4.2. Create Robot Algorithms in C++

For creating a new robot algorithm inside the simulator you need to do the following steps:

1. Create a subclass of `Robot`.
2. Put the algorithm into `src/RobotImplementations/`.
3. Write the method `compute()`.
4. Write the method `get_algorithm_id()`.
5. Add an include of the header of your new robot class in file `src/SimulationKernel/factories.cc` and also add here the algorithm identifier as option in method `robot_factory(...)`.

3. Utilize the output files

3.1. Statistics

A simulation run results in three output files of statistic data:

- `gnuplot_20091224_184129_ALL.plt` (GNUPlot-configuration file)
- `output_20091224_184129_ALL.plt` (according statistic data)
- `output_20091224_184129_DATADUMP_FULL.plt` (complete data dump)

The filenames results from current date (year, month, day), the current time (hour, minute, second), followed by description of observed object subset (e. g. `ALL`, `MASTERS`,...).

A. gnuplot

A.1. Visualization with gnuplot

Display with gnuplot To display statistical information about a finished simulation run the correct gnuplot configuration file needs to be opened with gnuplot. On Unix systems (with installed gnuplot) the interactive gnuplot shell can be used ((`> load` "`gnuplotfile.plt`"). More commands can be displayed using `> help`. Also it is possible to directly display the file with `$ gnuplot -persist` "`gnuplotfile.plt`". The additional option `-persist` causes the window to stay open after the file has been opened. For more informaion on gnuplot read the manual page `$ man gnuplot`.

Display configuration A file `output_20091224_184129_ALL.plt` might look like this:

# time	avg_spd	minball_x	(...)
0	9.2	20.5	
1	11.3	21.5	
4	9.1	21.9	
6	7.5	22.3	

Each line contains extremly valuable statistical data for a certain time. The start of a new column is marked by one (or more) whitespaces. comment lines are started with a '#' sign.

The file `gnuplot_20091224_184129_ALL.plt` also contains the formatting of the extremly valuable statistical data. The formatting can be changed if so desired.

```
# statistics of the simulation
#==================================
set title ' SCHLAUE SCHWÄRME '
set xrange []
set yrange []
set grid
set pointsize 0.5
set xlabel 'time'
set ylabel ''
```

```
plot 'output_(...)_ALL.plt' using 1:2 title 'avg_spd' with linespoints,\
     'output_(...)_ALL.plt' using 1:3 title 'minball_x' with linespoints
```

The lines have the following meaning:

- `set title 'Titel'` generates a title within the upper part of the graphical display.
- `set xrange [min:max]` limits the visible area (horizontal) to the area between *min* ans *max*, `set xrange []` scales the extremly valuable statistical data based on the datapoints.
- `set grid` displays a beautiful grid.
- `set pointsize multiplikator` scales the size of the points (if there are any) according to *multiplikator*.
- `xlabel 'Label'` labels the x-axis with useful information.

These choices only concern the graphical display. The configuration of the displayed data is done later.

- `plot 'Dateiname.plt'` starts the display process. Data is read from the input file. The parameter `using 1:2` uses the first column as x-axis and the second column as y-axis. `title 'avg_speed'` adds a title for the function defined by this. The addition `with linespoints` causes the data points to be displayed with lines between them. `linespoints` can be replaced by:
  - `lines` connects each data point with a line
  - `dots`displays datapoints as dots (for many datapoints)
  - `points` displays datapoints as points
  - `linespoints` displays a combination of *points* and *lines*
  - `impulses` displays a vertical line for each data point

More about gnuplot

[1] Homepage: `http://www.gnuplot.info/`
[2] Introduction course: `http://userpage.fu-berlin.de/völker/gnuplotkurs/gnuplotkurs.html`

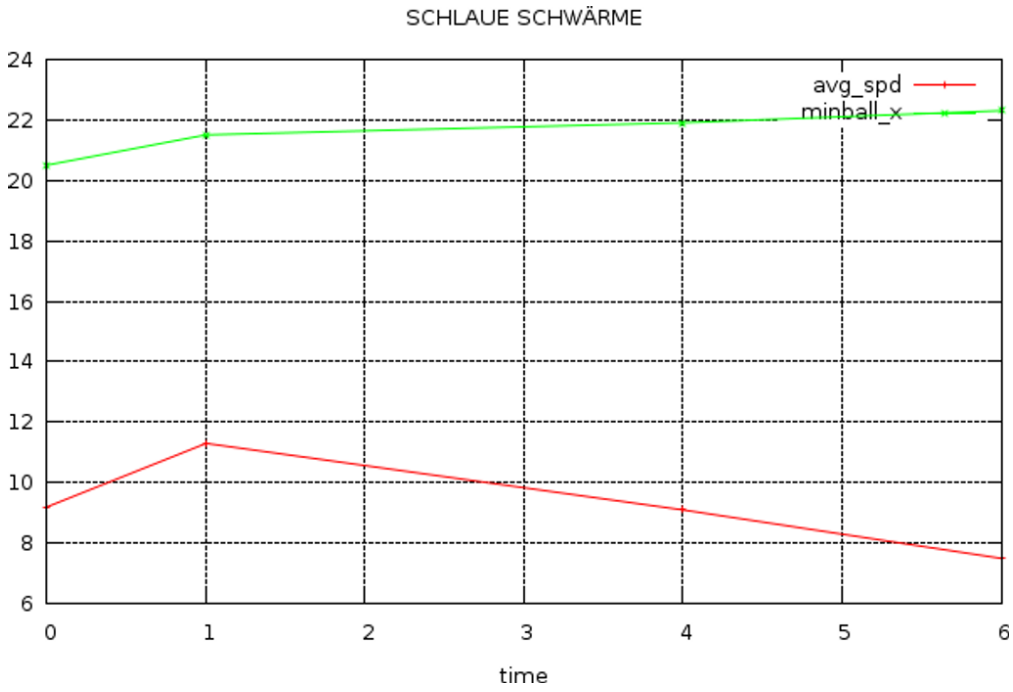

Figure A.1.: Example result output from gnuplot

B. Input-file Specification

B.1. Input-file Specifications

There are exactly four kinds of input files for the RobotSwarmSimulator. This includes the project specification files and also the Lua-script-files that define the robot behavior.

1. The main projectfile containing information about the model. The extension of this type of file is "'.swarm"'.

2. A file containing robot information. The extension of this file is "'.robot"'.

3. A file containing obstacle information. The extension of this file is "'.obstacle"'.

4. Lua file that describes the robot behavouir. The extension of this file is "'.lua"'.

B.1.1. Main projectfile

The following specifications hold only for the main projectfile (with extension `.swarm`):

- A comment begins with a '#'.

- A line is a comment line (beginning with a '#'), an empty line or a line containing a variable followed by an equal sign followed by a *quoted* value of this variable. Example:

```
VAR_1="value"
VAR_2 = "value"
VAR_3= "value"
VAR_4 ="value"
```

- a variable name has to be of the following form: `[A-Z0-9_]`$^+$

Variables

The main project file contains the variables defined in Tables B.1 and B.2.

Also the following should be considered:

- The order of the variables in the main project file is not important.
- If a variable does not appear in the main projectfile, then its default value will be used if such a default value does exist (otherwise an exception will be thrown while loading the main project-file).

Variable name	Possible Values	Description	Default
PROJECT_NAME	String	Name of the project	–
COMPASS_MODEL	Still needs to be specified by the ASG-Team. For instance NO_COMPASS	Compass model	FULL_COMPASS
ROBOT_FILENAME	For instance robot_file. The extension of the file must not be appended in this variable.	Filename of the robotfile	same as project file
OBSTACLE_FILENAME	For instance obstacle_file. The extension of the file must not be appended in this variable.	Filename of the robotfile	same as project file
STATISTICS_SUBSETS	A concatenation of none or more of the following strings: {ALL}, {ACTALL}, {INACTALL}, {MASTERS}, {ACTMASTERS}, {INACTMASTERS}, {SLAVES}, {ACTSLAVES}, {INACTSLAVES}	Defines the subsets of all robots for which to calculate individual statistical data. E. g. "{ALL} {MASTERS}" will produce statistical information on *all* robots as well as on *masters only*	NONE
STATISTICS_TEMPLATE	One of the following: "ALL", "BASIC" or "NONE"	Identifies the set of informations to calculate for each subset.	ALL
STATISTICS_DATADUMP	Either "FULL" or "NONE"	Whether or not detailled information (E. g. all robots positions at each event) should be streamed to a file during simulation.	NONE
ASG	SYNCHRONOUS, ASYNCHRONOUS or SEMISYNCHRONOUS	Type of ASG	SYNCHRONOUS
ASYNC_ASG_SEED	unsigned int	Seed for asynchronous ASG, only set if ASG=ASYNCHRONOUS	-
ASYNC_ASG_PART_P	double	Participation Probability for asynch ASG, only set if ASG = ASYNCHRONOUS	-
ASYNC_ASG_TIME_P	double	parameter governing the timing of asynch ASG, only set if ASG = ASYNCHRNOUS. The lower this is the more often events happen.	-
ROBOT_CONTROL	see section 2.1	RobotControl to use	-
CAMERA_POSITION	x,y,z, where $x, y, z \in \mathbb{R}$	Initial camera position	0,0,0
CAMERA_DIRECTION	x,y,z, where $x, y, z \in \mathbb{R}$	Initial camera direction	1,0,0

Table B.1.: Variables in the main project file

Variable name	Possible Values	Description	Default
MARKER_REQUEST_HANDLER_TYPE	element from {STANDARD,NONE}	Type of Marker Request Handler to use	{NONE}
TYPE_CHANGE_REQUEST_HANDLER_TYPE	element from {STANDARD,NONE}	Type of Type Change Request Handler to use.	{NONE}
POSITION_REQUEST_HANDLER_TYPE	element from {VECTOR,COLLISION,NONE}	Type of Position Request Handler to use	{NONE}
VELOCITY_REQUEST_HANDLER_TYPE	element from {VECTOR,NONE}	Type of Velocity Request Handler to use	{NONE}
ACCELERATION_REQUEST_HANDLER_TYPE	element from {VECTOR,NONE}	Type of Acceleration Request Handler to use	v
STANDARD_MARKER_REQUEST_HANDLER_SEED	integer	Seed for Marker Request Handler to use	{NONE}
STANDARD_TYPE_CHANGE_REQUEST_HANDLER_SEED	integer	Seed for Type Change Request Handler to use.	-
POSITION_REQUEST_HANDLER_SEED	integer	Seed for Position Request Handler to use	-
COLLISION_POSITION_REQUEST_HANDLER_SEED	integer	Seed for Position Request Handler (of type COLLISION) to use	-
VELOCITY_REQUEST_HANDLER_SEED	integer	Seed for Velocity Request Handler to use	-
ACCELERATION_REQUEST_HANDLER_SEED	integer	Seed for Acceleration Request Handler to use	-
STANDARD_MARKER_REQUEST_HANDLER_DISCARD_PROB	element from interval [0, 1]	Discard probability for Marker Request Handler to use	-
STANDARD_TYPE_CHANGE_REQUEST_HANDLER_DISCARD_PROB	element from interval [0, 1]	Discard probability for Type Change Request Handler to use.	-
POSITION_REQUEST_HANDLER_DISCARD_PROB	element from interval [0, 1]	Discard probability for Position Request Handler to use	-
COLLISION_POSITION_REQUEST_HANDLER_DISCARD_PROB	element from interval [0, 1]	Discard probability for Position Request Handler (of type COLLISION) to use	-
VELOCITY_REQUEST_HANDLER_DISCARD_PROB	element from interval [0, 1]	Discard probability for Velocity Request Handler to use	-
ACCELERATION_REQUEST_HANDLER_DISCARD_PROB	element from interval [0, 1]	Discard probability for Acceleration Request Handler to use	-
POSITION_REQUEST_HANDLER_MODIFIER	list of vector modifiers (see 2.1)	List of vector modifiers for Position Request Handler to use	-
COLLISION_POSITION_REQUEST_HANDLER_MODIFIER	list of vector modifiers (see 2.1)	List of vector modifiers for Position Request Handler (of type COLLISION) to use	-
VELOCITY_REQUEST_HANDLER_MODIFIER	list of vector modifiers (see 2.1)	List of vector modifiers for Velocity Request Handler to use	-
ACCELERATION_REQUEST_HANDLER_MODIFIER	list of vector modifiers (see 2.1)	List of vector modifiers for Acceleration Request Handler to use	-
COLLISION_POSITION_REQUEST_HANDLER_STRATEGY	element from {STOP,TOUCH}	Type of strategy to use for collision handling (see 2.1)	-
COLLISION_POSITION_REQUEST_HANDLER_CLEARANCE	positive floating point value	Two objects with distance less than this value will be considered colliding	-

Table B.2.: Variables in the main project file

Example of a main project file

A main project file may look like:

```
#
# Description about configuration.
#

  PROJECT_NAME="My Exciting Project"
  COMPASS_MODEL="NO_COMPASS"
  ROBOT_FILENAME="myrobots"
  OBSTACLE_FILENAME="myobstacle"
  STATISTICS_MODULE="0"
  ASG="ASYNCHRONOUS"
  ROBOT_CONTROL="ROBOT_TYPE_ROBOT_CONTROL"
  MASTER_VIEW="GLOBAL_VIEW"
  SLAVE_VIEW="ONE_POINT_FORMATION_VIEW"
  SLAVE_ONE_POINT_FORMATION_VIEW_RADIUS="5.0"

  CAMERA_POSITION="0,0,0"
  CAMERA_DIRECTION="1.5,0,0.5"

  MARKER_REQUEST_HANDLER_TYPE="STANDARD"
  STANDARD_MARKER_REQUEST_HANDLER_DISCARD_PROB="0.5"
  STANDARD_MARKER_REQUEST_HANDLER_SEED="1"

  TYPE_CHANGE_REQUEST_HANDLER_TYPE="NONE"
  # no additional variables needed

  POSITION_REQUEST_HANDLER_TYPE="VECTOR"
  VECTOR_POSITION_REQUEST_HANDLER_DISCARD_PROB="0.1"
  VECTOR_POSITION_REQUEST_HANDLER_SEED="3"
  VECTOR_POSITION_REQUEST_HANDLER_MODIFIER="(VECTOR_TRIMMER,1.5);(
      VECTOR_RANDOMIZER,5,2.5)"

  VELOCITY_REQUEST_HANDLER_TYPE="VECTOR"
  VECTOR_VELOCITY_REQUEST_HANDLER_DISCARD_PROB="0.1"
  VECTOR_VELOCITY_REQUEST_HANDLER_SEED="3"
  VECTOR_VELOCITY_REQUEST_HANDLER_MODIFIER="(VECTOR_TRIMMER,1.5);(
      VECTOR_RANDOMIZER,5,2.5)"
```

B.1.2. Robot file

The robotfile uses a csv-compatible format. Therefore the information for one robot has to be saved in exactly one line of the file. Each line contains the following data. The order of this data is important!

You can declare specific lines as comments by setting # as the first sign of the corresponding line.

- ID-number
- initial position (x, y, z)
- initial type (for instance master, slave,...)
- initial velocity (x, y, z)

- initial acceleration (x, y, z)
- initial status (maybe sleeping or ready; still has to be specified more precisely)
- initial marker information (still has to be specified)
- algorithm to use (shortcut for an algorithm; still needs to be specified)
- color (using this color a robot is marked for instance for a special treatment during the visualization; this color isn't used anywhere else); Integers correspond to the following colors: 0 green, 1 blue, 2 cyan, 3 red, 4 magenta, 5 yellow, 6 white, 7 black, 8 orange, 9 purple
- coordinate system axes (triple $x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3$; this field will be left empty, if axes are supposed to be generated uniformly at random)

The first line always is (column headers):

```
1  "ID","x-position","y-position","z-position","type","x-velocity","y-velocity","
       z-velocity","x-acceleration","y-acceleration","z-acceleration","status","
       marker-info","algorithm","color","x-axis-1","x-axis-2","x-axis-3","y-axis-1",
       "y-axis-2","y-axis-3","z-axis-1","z-axis-2","z-axis-3"
```

Each non-number is quoted and each number may be quoted.

Example of a robot file

```
1  "ID","x-position","y-position","z-position","type","x-velocity","y-velocity","
       z-velocity","x-acceleration","y-acceleration","z-acceleration","status","
       marker-info","algorithm","color","x-axis-1","x-axis-2","x-axis-3","y-axis-1","
       y-axis-2","y-axis-3","z-axis-1","z-axis-2","z-axis-3"
2  0,5.3,9.2,6.4,"master",1.5,2.5,3.5,1.5,2.5,3.5,"sleeping",0,0,0,1,0,0,0,1,0,0,0,1
3  1,"2.5","4.2","8.8","slave",1.5,2.5,3.5,1.5,2.5,3.5,"ready",0,1,0,1,0,0,0,1,0,0,0,1
```

B.1.3. Obstacle file

Like the robot file the obstacle file uses a csv-compatible format. Therefore the information for one robot has to be saved in exactly one line of the file. Each line contains the following data. The order of this data is important!

You can declare specific lines as comments by setting # as the first sign of the corresponding line.

- type (marker, sphere or box)
- position (x, y, z)
- marker information (still needs to be specified)
- $x/y/z$-lengths or radius (depending on type)

The first line always is (column headers):

```
"type","x-position","y-position","z-position","marker-info","size-info","",""
```

Each non-number is quoted.

Example of an obstacle file

```
"type","x-position","y-position","z-position","marker-info","size-info","",""
"box",2.0,3.0,4.0,0,1.0,2.0,3.0,
"sphere",3.4,5.2,5.1,0,5.0,"",""
"marker",3.5,1.4,5.1,0,"","",""
```

As you can already see in the example, if the type of an obstacle is sphere, then the last two values must be empty, i. e. ",". Analoguos, if the type is marker, the last three values must be empty, i. e. ",",".

C. Scaling Tests

C.1. Scaling Tests

We present preliminary performance results of RobotSwarmSimulator. The goal of the performance tests was to find out how the software scales with the number of simulated robots and which parts of the simulation take the most time to compute. All measurements have been taken on a Windows Vista PC with the following hardware: Intel Core 2 Duo at 2,53 Ghz and 4 GB DDR3 Ram at 1066 Mhz. The code has been compiled using MinGW GCC 3.4.5.

At all times, the visualization component of RobotSwarmSimulator has been switched on. It runs in its own thread and consumes world information objects generated by the actual simulation kernel. To get an idea of the impact of visualization on the performance as a whole, we independently measured how long setting up each rendering frame takes. In between each frame, a variable amount of simulation time passes by, this amount is referred to as processing time. Figure C.1 shows that rendering one frame increases linearly with the amount of robots. At the target of 1000 robots it takes about 0.01 seconds.

In the synchronous time model, each simulation step consists of updating all robots views (look event), executing each robots algorithm on that view (compute event) and then handling each robot's request to the world state (handle request event). We measured the computation time for each of these three events. The figures given represent the average time for one event after 33 steps have been simulated.

We first look at the simplest case when no algorithm at all is loaded onto the robots and they receive the global view. All robots are distributed randomly in space and receive random velocity and acceleration. Figure C.2 tells us that Look and Request events have similar computation times and stay constant regarding the total number of robots. The compute events take almost no time and can be disregarded here. Specifically, Look and Request executed in respectively 0.15 and 0.34 seconds which results in around 0.5 seconds for a simulation step. Note that the robots actually do not return any requests so that this time can probably be regarded as overhead for copying data around.

Now we consider the Circle algorithm, specified in `circle.lua`. Here, every robot calculates the center point of all visible robots and generates a velocity request so that it rotates around the center. The robots form a rotating ring. It has been tested with global (one huge ring) and local view (several smaller rings).

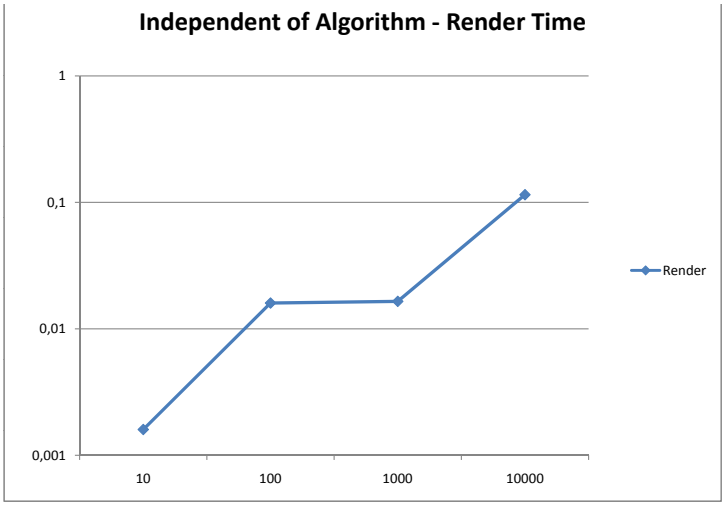

Figure C.1.: Scaling test of rendering

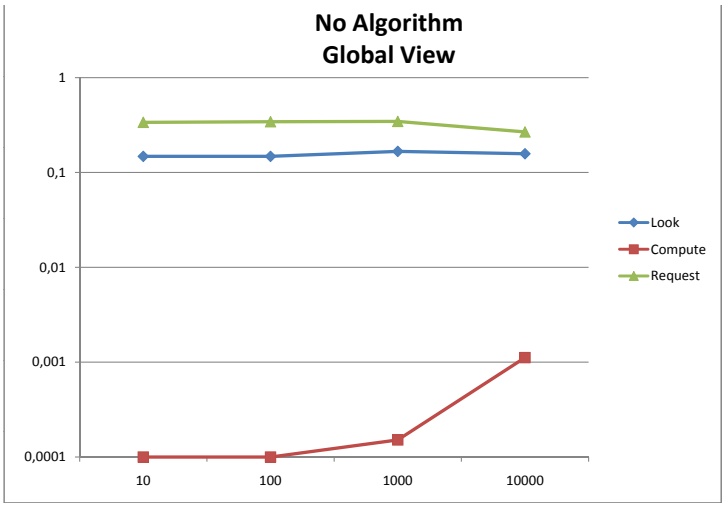

Figure C.2.: Scaling test – no algorithm and global view

Figure C.4 shows how performance scales in this scenario. The three different event types behave very differently. The compute event has a computation time $\mathbb{O}(n^2)$ where n is the number of robots. This should be expected, as each robot regards each other in the computation of the center point. Surprisingly, the times for look and request actually decrease. When simulating 1000 robots, compute is the dominant term as each event takes 13 seconds to execute, compared to 0.010 and 0.014 for look and request respectively.

Consider the same algorithm with only a local view. We use the spheric view with a radius of 3. As it can be seen in Figure C.3, the different types of events scale similarly, however in a different relation to each other. Look and request events take the same time as when using global view, which means that the octree implementation is so efficient that it is not slower than simply copying all the robots. Compute events however only take 0.94 seconds. This is due to the fact that now every robots consider only neighboring robots for calculating the center point.

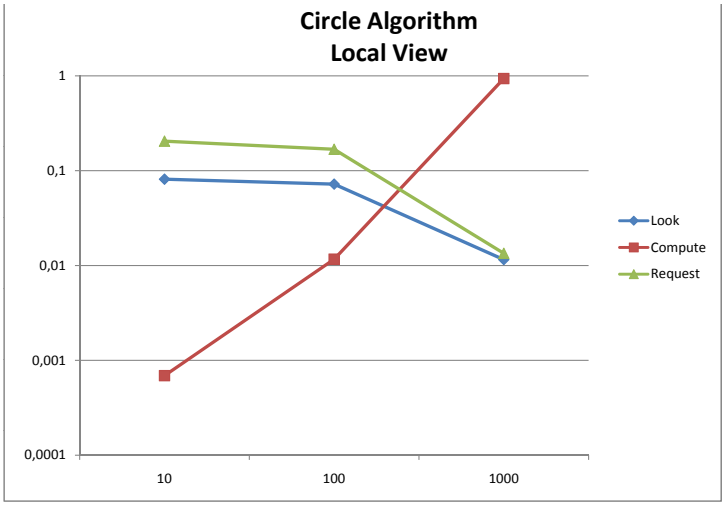

Figure C.3.: Scaling test – circle local

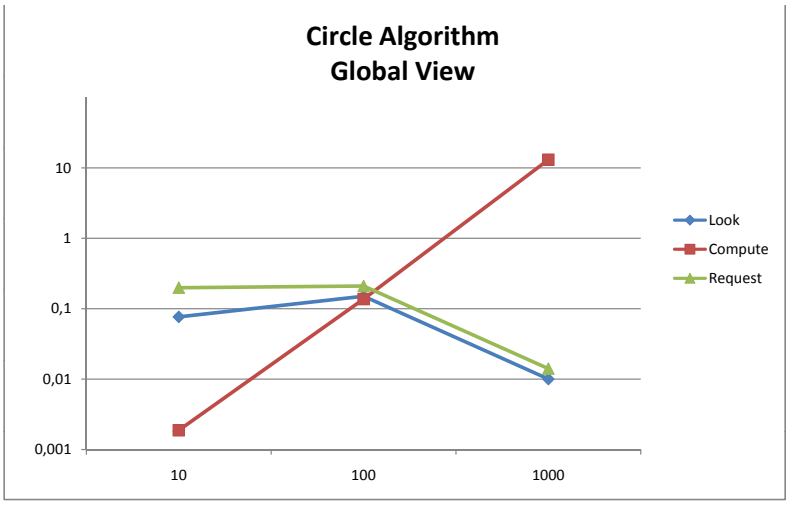

Figure C.4.: Scaling test – circle global