FEFU RoboSub 2014 autonomous underwater vehicle

Roman Babaev, Andrey Gatcenko[1], Vladislav Goy, Mark Guliaev,
Andrey Sakharov, Gleb Shestopalov, Maksim Sporyshev, Anton Tolstonogov

Far Eastern Federal University,
Vladivostok, Russia
[1]gatsenko93@mail.ru

Abstract **– The paper describes the development of the AUV which our team has prepared for RoboSub 2014. The major application of the vehicle besides the competition is its usage for research and development of vision-based navigation and control methods.**

I. Introduction

Our team has participated in RoboSub twice – in 2012 and 2013. Our vehicle is a modification of the previous one.

The vehicle design (fig. 1) is well-proven during the last year, so we decided not to make essential changes in it. The design was slightly refined with a new tool added to perform a manipulation task.

Significant changes were made regarding to the software. These changes give a base for future transition to more convenient tools, save us time during programming and pool tests. Here is a short list of the most significant software updates compared to the last year:

- more reliable approach to messaging;
- refined mission structure;
- task parameters separation;
- Gazebo based simulator;
- segmentation algorithm for vision module;
- postprocessing tools.

II. Hardware

A. Frame and housings

AUV frame is made of polypropylene. Housings were manufactured from aluminum and oxidized. Pneumatic system compensates the water pressure and prevents leaks.

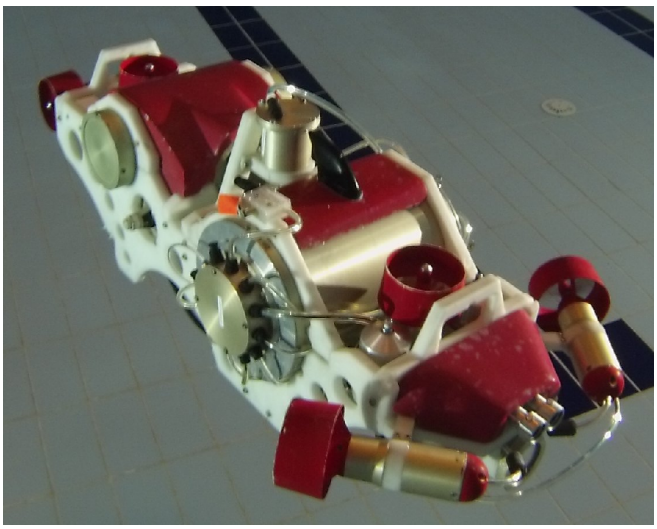

Fig. 1. The AUV appearance.

B. Sensors

Three sensors are used for navigation in space: a depth sensor (DMP331), an inertial module (Xsens Mti IMU) and a Doppler velocity log (Teledyne RD Instruments Explorer DVL). These sensors allow us to control the vehicle in 5 out of 6 degrees of freedom, which is sufficient to accomplish the mission.

There are also two digital USB cameras (EVS VEC-245) on the AUV. One of them is directed forward and the other one is directed down. The cameras work in 800×600 px mode, 15 fps. However only 3-5 fps are used for vision-based control.

There are digital signal processor (DSP, Danville's dsp stack 21369zx) and 3 hydrophones are used in the AUV to perform the pinger task.

Safety sensors are mounted in every waterproof housing. A warning message is sent immediately to a GUI operator program if one of them is activated.

C. Actuators

For successful tasks execution the AUV is equipped with pneumatic system for torpedo shooting which is located directly above the front camera and with system for dropping cargoes based on electromagnets.

A grabber is used for execution of manipulation task. The fig. 2 depicts the disposition of the front camera, torpedoes and the grabber.

There are 5 thrusters on the AUV: 2 vertical ones and 3 horizontal. Thrusters use 24 volts Faulhaber motors and polyurethane propellers.

D. Computers

Two computers are used for calculations. The first one is a single-board computer PC/104-Plus with processor Vortex 86DX (600 MHz, 256 Mb RAM). Most of software modules (including a mission module) and sensor data processing is executed on the computer. The second computer is used for image and hydroacoustics processing. It uses an Intel Atom D510 1.66 GHz processor and 1Gb of RAM.

Fig. 3 shows the structure of the AUV LAN. Two computers are connected through a router. Other components are joined to them via USB and RS-232. Microcontroller for power management and peripherals control is linked in a single CAN bus with thruster microcontrollers. The bus is linked to the computer through a USB-CAN bridge.

E. General construction

Approximate dimensions of the vehicle: 1.1×0.5×0.4 meters, weight – 40 kilos.

The design of the AUV is demonstrated on fig. 4. There are housing with electronics in front of the AUV and housings with the DVL and the battery in the back.

The inertial unit containing a compass is placed in a separate housing located in the upper section of the vehicle as far as possible from the thrusters. Hydrophones are spaced at three corners and located at the bottom in the inner side.

III. Software

A. Operator software

Our operator software allows us to control the vehicle and to monitor the vehicle status.

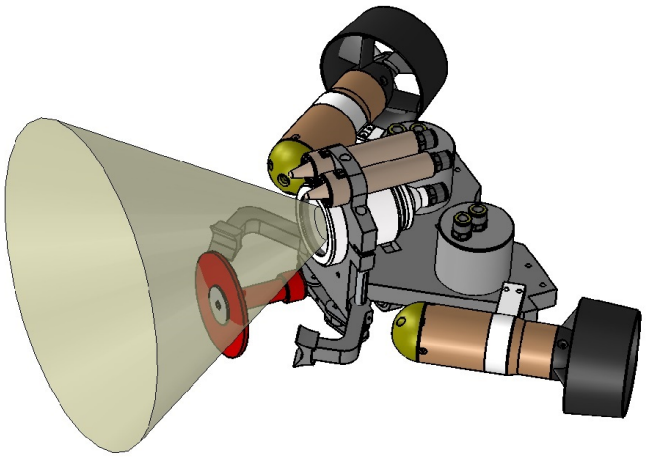

Fig. 2. CAD drawing of the front part of the vehicle. Torpedoes, the front camera and the grabber can be seen.

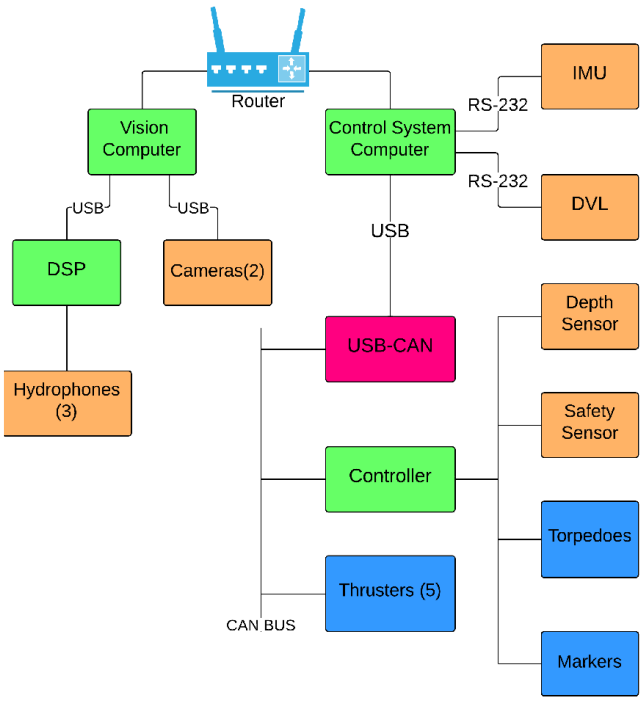

Fig. 3. AUV LAN.

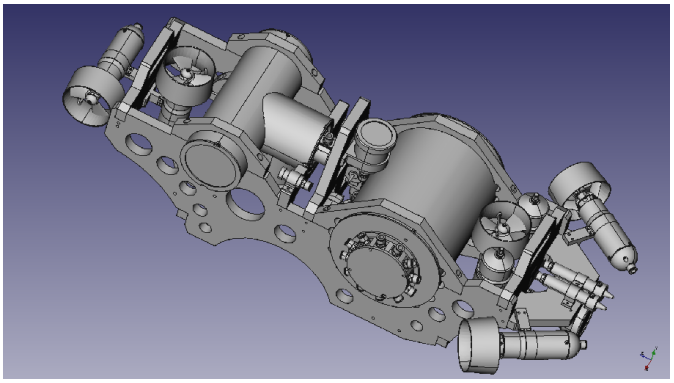

Fig. 4. The AUV CAD model (exported to FreeCAD).

It connects to the control system via local network (connection is established between the router on the vehicle board and operator's laptop via Ethernet or Wi-Fi). During development of operator software we paid special attention to the information from safety sensors such as water alarm sensors and battery voltage.

Besides, the GUI provides the capabilities for vehicle remote control, for video module control and for receiving images from cameras (images are compressed in JPEG format for transmitting over the network).

The operator GUI interface is given in fig. 6.

B. General structure

We have used hierarchic structure of control system for our vehicle. Each node of this structure is implemented as a separate executable module. Information transmission between modules organized by messaging.

Our control system structure presented at fig. 5.

System consists of 10 modules. Mission module which defines vehicle actions is placed on top of the hierarchy. Regulators module is responsible for motion control. Navigation module processes the information from navigation sensors to provide it to regulators and mission module. Other modules exchange data with hardware devices. They send commands to actuators and process sensor data.

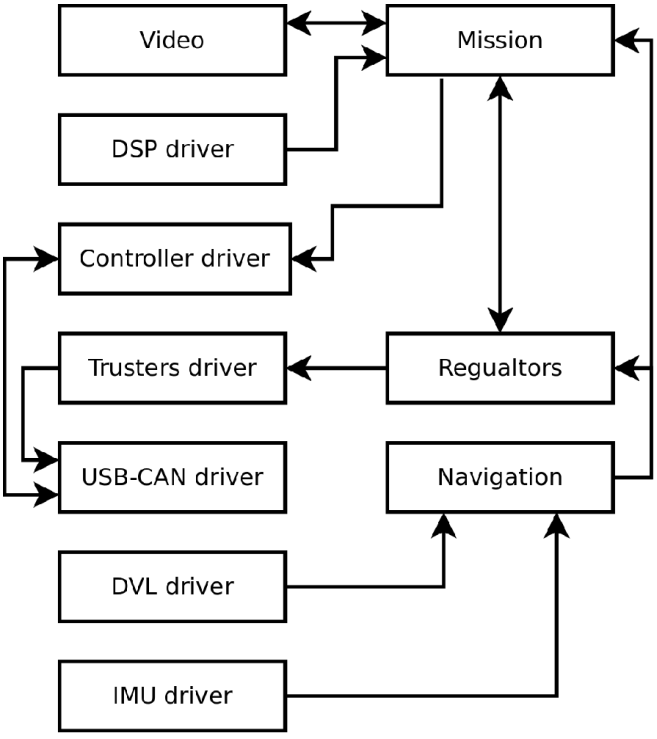

Fig. 5. Control system structure.

C. Module communication framework

We use an IPC library [1] for communication between modules in our control system. There were several problems with IPC related to necessity of accurate description of each message, neat memory management, the usage of C interface for message processing, which inevitably led to errors.

Data transfer between modules of control system is a common topic and there are some means in robotics software frameworks to solve this problem. It could be solved via both shared memory (e.g. MOOS control system [2], Cornell University Team control system [3]), and via messaging. We decided to use the second option since our current software is based on messaging. Among robotics platforms which use messaging we have chosen ROS [4] as our future platform – its usage is growing intensively, there are examples of its use in an AUV specially designed for RoboSub (e.g. Bumblbee team [5]).

Full transition to ROS is a labour-intensive process. So, we decided to implement a step-by-step transition, starting with the declarative description of messages in ROS format. We developed a program for this, which allows to generate a IPC formats end an according C++ code for messages described in ROS format.

We have also developed a library, which encapsulates all IPC facilities and simplifies programmer's work in the process of writing or alternating modules.

D. Mission module

The mission module is located on the top of control system hierarchy. It concentrates the motion and actions logic of the vehicle.

The mission consists of separate tasks, which are executed in a linear sequence. The order of execution is described in the set of configuration files. There is a global map that stores information about recently found orange stripes. This information is used for vehicle positioning before starting the next scheduled task.

Our previous task execution scheme was pretty similar to the state machine. This year the set of states for each task is explicitly formulated for a state machine. We have strictly structured our code in accordance with that and developed a corresponding object oriented structure. A typical action in one of the states is stabilization of the vehicle in front of an object in the video frame.

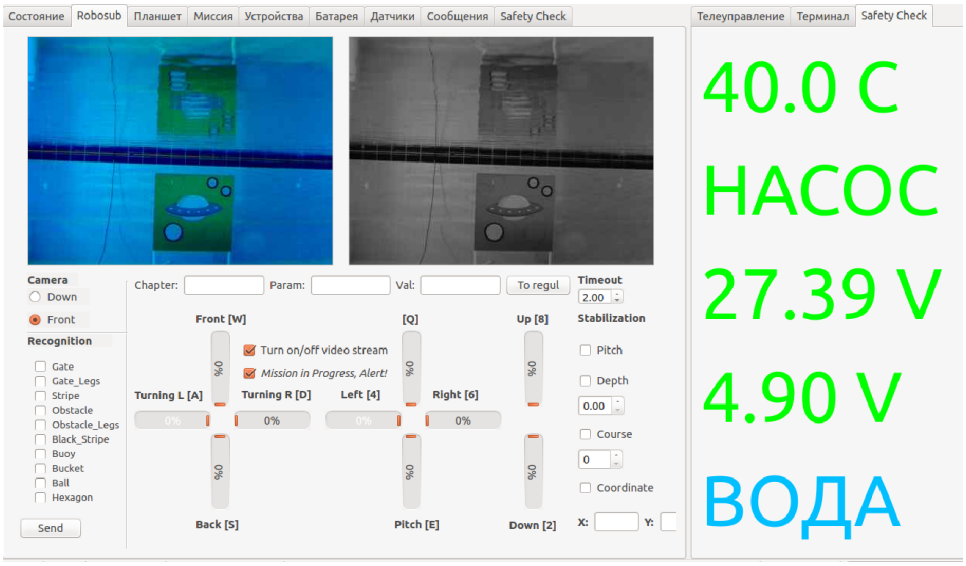

Fig. 6. Interface of one of operator GUI sheets.

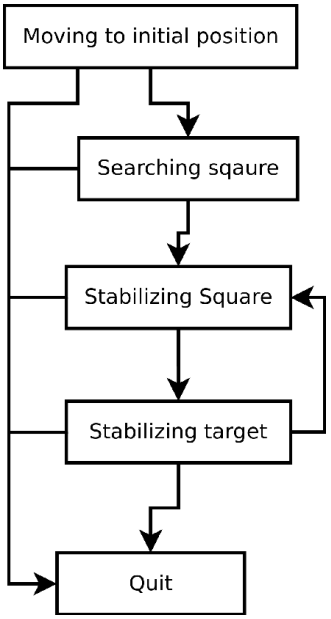

Fig. 7. Finite state machine for the shooting task.

An example of the finite state machine for the shooting task is shown in the fig. 7.

The mission module is implemented in several C++ classes, their interaction is shown in fig. 8.

The Environment class receives and processes navigation messages, and also stores the physical vehicle parameters (e.g. it stores camera models) and information about environment objects used by tasks. Tasks can communicate via this class. The Environment is the interlayer for interaction between mission and lower layers.

The mission class is designed to create, configure and execute each of the tasks. The configuration of each task is formed here. Each task has a corresponding C++ class. Each of those classes uses the state machine implementation for process its states. It analyses data from the Environment.

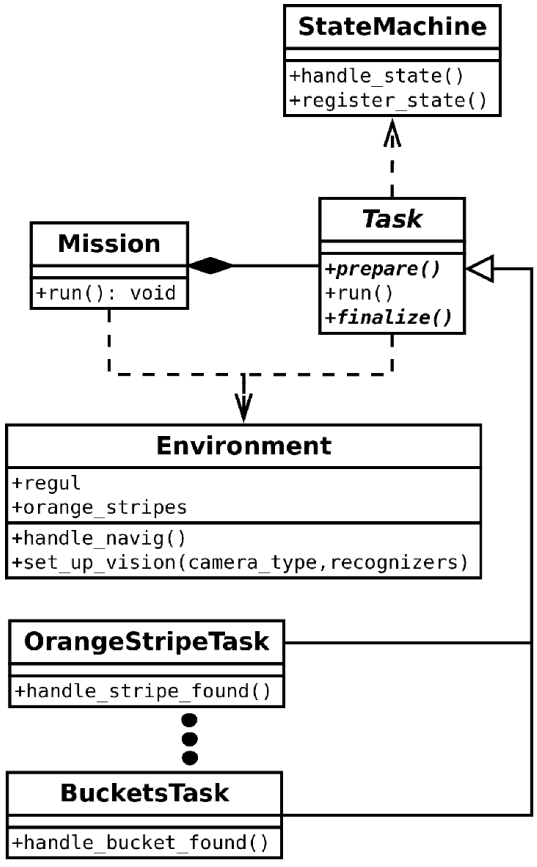

Fig. 8. UML class diagram used in the mission.

A YAML is used for mission configuration files description. A tasks reads parameters from three sources in the following order:

1. `task.yml`
2. `<task_name>.yml`
3. `mission.yml`

The `task.yml` file describes default parameters which are common for all tasks. A

separate file for each kind of task gives parameters common to all instances of tasks of this kind. The `mission.yml` file contains mission run parameters. A task could have about 50 parameters, this approach allows to decrease the size of the main configuration file and to make a run setup easier.

The mission execution time can be essential this year. So it's important to stabilize fast at objects of interest. We have added a differential component to a vision regulator for that. We have adjusted coefficients using our postprocessing tools. The fig. 9 depicts a process of stabilization at the orange stripe using a new regulator.

E. Low-level controls

Regulators is a module which mission module communicates with for controlling the AUV motion. Regulator interface provides separate control for coordinates, roll, depth and heading. We use PID-controllers for stabilizations. The result of heading control during one of runs is shown in fig. 10.

F. Video module

The main goal of video module is analyzing images taken from cameras and passing information about detected objects to mission module. Mission sends to video module a list of required objects and the number of camera to take images from.

OpenCV library is used for image processing. The version 2.4.8 is installed on the vehicle.

Each detection algorithm usually consists of the following parts:

1. Preprocessing. It's done by applying filters or clustering algorithms to correct underwater colors, remove noise and small useless details. Color correction is done by increasing red channel in RGB image. Then three ways of image preprocessing are used: median blur, gaussian blur, SEEDS superpixel segmentation. Median and gaussian filters are implemented in OpenCV. We have developed a custom SEEDS implementation following [6].

2. Color binarization. The source image is translated to HSV color space and the binarization is executed by hue and saturation threshold.

3. Contour analysis. It is based on OpenCV implementations of Suzuki-Abe algorithm and polygon approximation, a custom Hough implementation is also used.

We've calibrated our cameras using OpenCV calibration tools. A chessboard of 1×1 meter size was made. The chessboard is positioned so that it lies entirely in the field of vehicle camera view, the camera takes images. Then the images are given to the calibration program which detects angles on the chessboard and calculates camera parameters, such as focal distance and radial, tangential distortion factors. All this parameters are considered in our camera model which helps us to calculate the heading to the detected object and determine distance to this object more accurately.

G. Implementation

C++ programming language is used in our vehicle control system software implementation. Some modules significantly depend on C++11 standard features so we were to update GNU compiler to 4.8 version.

Boost libraries provides us with some useful extensions that we apply to, for example, our thread pool implementation. We also need it in command line interface implementation for our tools.

Cmake intelligent build system is used to build our system. Bash scripts are used to launch it.

We have 6 software engineers in our team. Git version control system and central private repository on bitbucket.org are chosen to organize their team development process.

IV. TESTS AND TRIALS

A. Simulator

Our simulation system was implemented using Gazebo simulation tools and API.

Gazebo uses Protobufs for interprocess communication, so we use adapter tool which converts messages from our internal format to protobufs. This approach allows us to use the same software with both simulator and real devices.

Simulation system GUI shown in fig. 11.

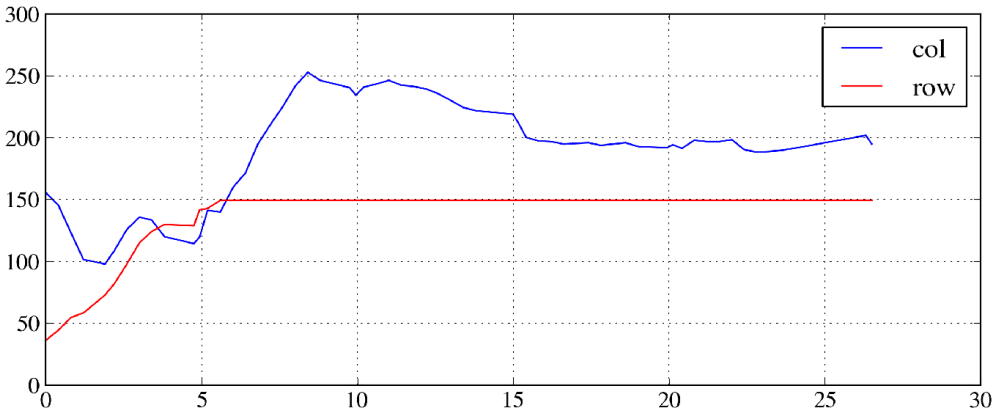

Fig. 9. Position of the orange path center in the frame. The distance is shown in pixels. X axis shows time (in seconds). The center of the frame is (200, 150). Straight line on “row” is due to the fact that the path does not fit into the frame completely.

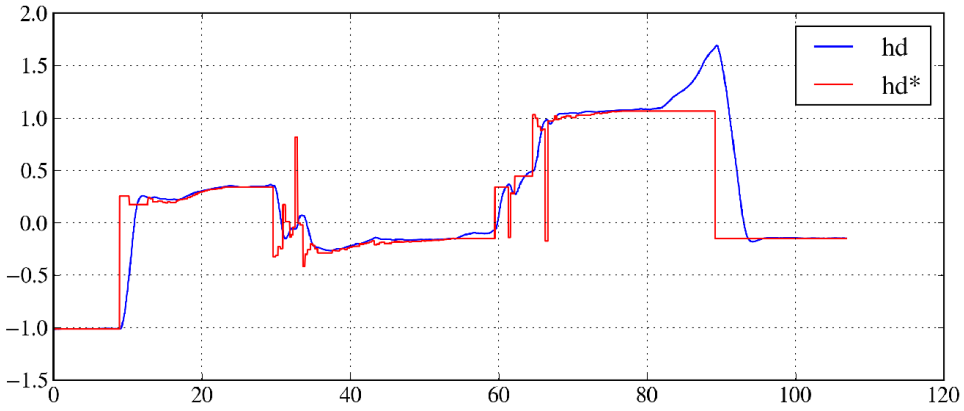

Fig. 10. Change of the programmed and the actual heading during one of runs. The heading is measured in radians. X axis shows time (in seconds).

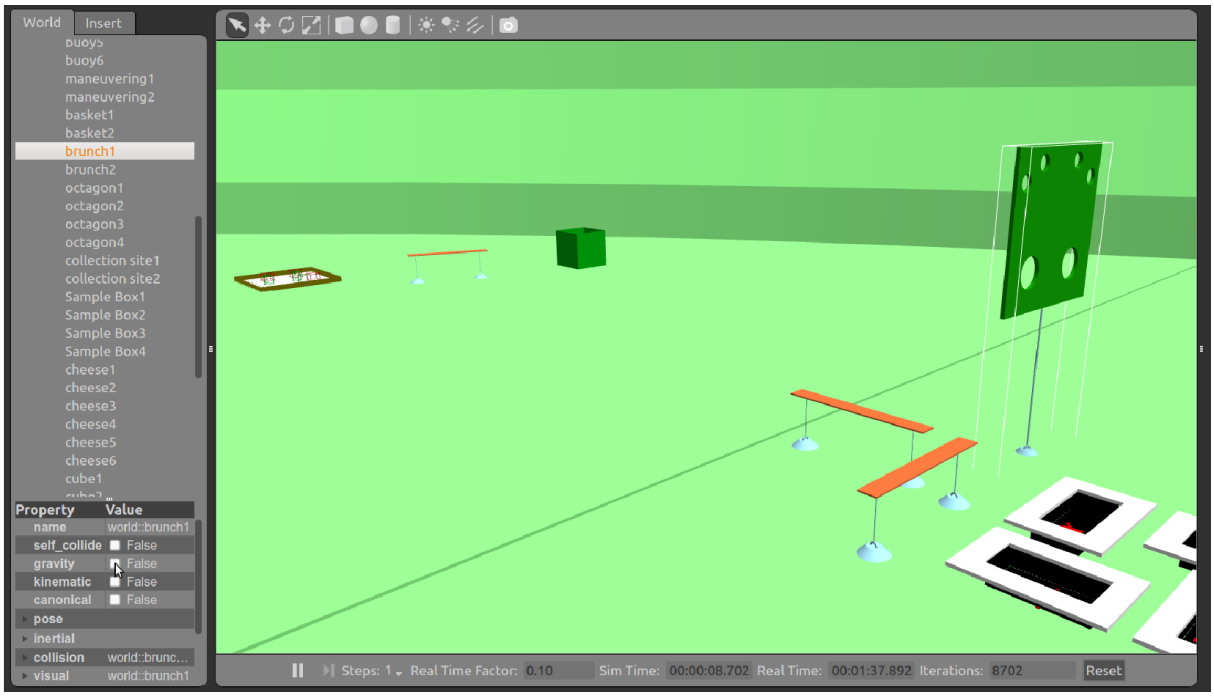

Fig. 11. Gazebo GUI.

B. Pool tests

One of the most essential parts in vehicle preparation is testing in pool.

This process is required to detect hardware failures, produce some components setup and find out unexpected bugs, that haven't been appeared earlier.

We started to test our vehicle in pool at the beginning of June in our campus swimming pool (see fig. 12).

We use text logs to debug our software. It Is human-readable could be fluently analyzed without special software. We have an agreement of a “basic form” of the log file. It is a CSV-like format with series of numbers. We have developed log_lot – a Python utility using Matplotlib to show charts for series in a basic form log. Log_plot has more then 10 parameters allowing to point out a given period of time, axis names etc. It receives the data from the standard input.

We also have a log_merge utility for merging to logs form two separate modules.

The mission log is not in the basic form since in contains text messages for human analysis. A simple run on the file with regular expressions can extract the specified info and represent in is the basic form.

Since all of our utilities use standard input and output, Unix pipes can be used for rapid log analysis. For example:

```
ssh user@auv cat /home/run/mission.log |
log_parser.py | log_plot.py row col
```

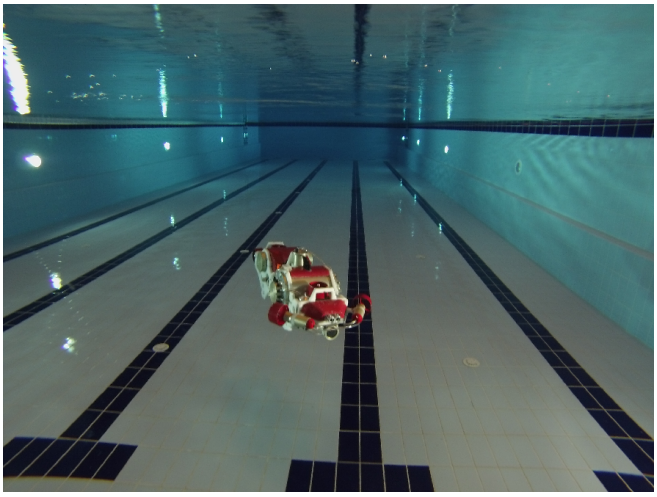

Fig. 12. FEFU pool with the AUV during a pool test.

ACKNOWLEDGMENT

The development of FEFU AUV is supported by Far Eastern Federal University and by Institute for Marine Technology Problems of Far Eastern Branch of Russian Academy of Sciences.

We would like to thank our mentor Dr. Alexander Scherbatyuk. We also would like to thank those from FEFU and IMTP FEB RAS who have supported us in this work.

REFERENCES

1. R. Simmons, *Inter-Process Communication: a reference manual*, http://www.cs.cmu.edu.
2. P. Newman, *MOOS – mission orientated operating suite*, Massachusetts Institute of Technology, Tech. Rep. 2299/08, 2008.
3. M. Burkardt, L. Barron, T. Brook et al, *Cornell University Autonomous Underwater Vehicle: Design and Implementation of the Ragnarök AUV*, http://cuauv.ece.cornell.edu.
4. M. Quigley, B. Gerkey, K. Conley et al, *ROS: an open-source robot operating system*, Open-source software workshop of the Int. Conf. on Robotics and Automation, Kobe, Japan, 2009.
5. O.T. Chang, G.E. Wei, J. Ong et al, *BBAUV: Autonomous Underwater Vehicle, software overview*, www.bbauv.com.
6. M.V. den Bergh, X. Boix G. Roig et al, *SEEDS: Superpixels Extracted via Energy-Driven Sampling*, ECCV 2012.