A MOOS-Based Online Trajectory Re-planning System For AUVs ⋆

Matko Barisic * **Zoran Vukic** * **Sean P. Kragelund** ** **Nikola Miskovic** *

* *University of Zagreb, Faculty of Electrical Engineering and Computing, Unska 3, HR-10000 Zagreb, Croatia (e-mail: matko.barisic@fer.hr, nikola.miskovic@fer.hr, zoran.vukic@fer.hr).*
** *Naval Postgraduate School, Center for Autonomous Underwater Vehicle Research, 1 University Circle, Monterey, CA 93940, USA (e-mail: spkragel@nps.edu).*

Abstract: This paper describes an open source navigation system architecture for use in autonomous underwater vehicles. It is based on the Mission Oriented Operating System proposed, published and programmed by (Newman (2006), Newman (2008)). It is uniquely applicable for work-in-progress type and development-stage software and capability installation onto an AUV system. This applicability is achieved by its completely modular nature, which is obtained by the operating system kernel running separate processes for each advanced navigation or control feature. Robustness is also achieved in this respect since failures and errors will cause only the individual modules that incurs them to fail. Such critical errors, bugs and failures will thereby be contained and their propagation halted from completely freezing even the low-level control loops and decision-making processes needed to successfully retrieve the malfunctioning AUV.

Keywords: Microcomputer-based control, computer architectures, computer controlled systems, concurrency control.

INTRODUCTION

The Mission Oriented Operating Suite (MOOS) was originally proposed by Newman (2006) and developed for research in behavioral control paradigms in room (closed spaces) robotics. Conceived with that niche in mind, it is a system that leverages advanced capabilities of the embedded controller across the ease of design of complicated and emergent behaviors, intelligent decision-making and state-machine programming. More specifically, it presupposes the existence of operating system kernel routines or dedicated middleware drivers to handle protocols like Ethernet, Bluetooth, IEEE 802.11b/g etc.

This greatly facilitates the use of this framework in one's own prototyping work, which comes down to a natural high level of abstraction when coding. A further strong suit of the MOOS framework is intuitive encapsulation of hierarchical and flow-diagrammatic ideas (familiar to control system engineers) into modules that a software engineer or a software designer is required to code. Although originally the AUV embedded controllers, especially those of experimental craft in the early 80's, would not have come pre-programmed with high levels of abstraction of certain services (Ethernet connectivity, WLAN connectivity etc.) nowadays the situation is markedly different. Use of the Ethernet protocol to connect arrays of sensors, or he use of USB ports with their ability to provide limited power has proliferated. This is allowing the AUV system designers to complement the use of the more classical communication protocols like the serial connection and/or CAN with these new and more easily programmed services.

The cross-AUV architecture proposed by the authors, and tested out on the Hydroid Remus vehicle operated by the Center for Autonomous Underwater Vehicle Research, Naval Postgraduate School, Monterey, USA (Healey et al. (2007)), is further detailed in Section 1: The Navigation System Architecture. The highest level of the navigation system architecture, the online trajectory replanner based on the virtual potentials method (Healey (2006), Barisic et al. (2007), Barisic et al. (2007)) is described in Section 2: The `pVPMPlanner` Module. Results of a *hardware-in-the-loop* simulation and verification run of the proposed system are provided in Section 3. The paper is concluded in Section 4.

1. THE NAVIGATION SYSTEM ARCHITECTURE

As stated in the Introduction, MOOS is a well-balanced framework for rapid prototyping of AUV control solutions. On one side, it decreases the time needed to prototype navigation systems with complex behaviors, intelligent decision-making and optimal use of the AUV resources. On the other hand, it remains true to ANSI C++ coding standard. The importance of the latter is that this makes it a desirable cross-platform solution. Also, this allows the designer to tap into the vast codebase of already existing

⋆ This research was made possible by a financial grant for a study tour in the USA provided by the "Unity Through Knowledge Fund" of the Ministry of Science, Education and Sports of the Republic of Croatia.

piecewise solutions, coded in C++, a penultimate industry standard. However, it comes at a cost – the reliance on pre-existing class frameworks and vendor-provided APIs. The MOOS classes to be inherited when coding rely heavily on header libraries, pre-compiled dynamically linked libraries, middleware software and drivers for an interface with actuators, sensors and low-level embedded processing (DSP chips, FPGA etc.). This introduces complexity, lengthens the run-time stack and piles context/namespace data on the memory heap. Consequently, the MOOS kernel and the central module, the `MOOSDB`, is only able to support relatively low sample rates. Additionally, the nature of the MOOS kernel (if used as-is) precludes *hardware-level enforcement* of an elective sample time. The messaging system, which in MOOS serves as the one and only synchronization mechanism, is in no way related to the hardware clock.

Most AUVs, however, operate with sample times of ca. 10Hz. Most processors used in embedded controllers, on the other hand, have clock frequencies in the 100 – 1000MHz orders of magnitude. This allows for literally billions of low-level instructions to be carried out within one sample time of the AUV systems (i.e. several hundreds of lines of ANSI C++ code). The ability of the embedded system to process such a large number of commands in a single sampling period, decreases the influence of the aforementioned hardware non-enforceability of the sample time to virtual non-existence.

The choice of MOOS (current version is described in Newman (2008)) as a navigation system architecture framework does pose some requirements on the AUV's embedded control system. It is critical that the embedded control system be a programmable and fully accessible (administrable) PC-standard computer running a *unixoid* operating system kernel (preferably a Linux kernel). Alternatively, it must be able to accept machine code compiled by the GNU C / C++ compiler commonly shipped and bundled with Linux distributions. This means that some minimum level of compatibility between the embedded system and a *unixoid*-kernel-running PC-standard computer must exist, or be emulated in some way (e.g. ability to run POSIX threads).

MOOS functions primarily as a set of the following:

- A ready-made class hierarchy to enable simple application development by inheriting either the `CMOOSApp` class for MOOS *modules*, or the `CMOOSInstrument` class for MOOS *instruments*.
- A ready-made application running the `MOOSDB`, a MOOS database server that collects, collates, indexes and distributes `CMOOSMsg`-class messages from one process to the others.
- A set of coding conventions, naming conventions and best practice examples that allow for human-readable, intuitive and clear code that is easy to distribute within the developers' community.

MOOS supports a fundamentally star-shaped client-server topology, where a set of applications and instruments communicate with a server in order to publish or read messages containing either string or double-precision floating point data.

MOOS applications are stand-alone processes callable individually and built into `.exe` files. When built, they can be shipped out and reused on different systems as-is. The system where a particular application is to be used, naturally, needs to operate a `MOOSDB` server handling the type of messages that the application expects and on the basis of which it performs its functions.

MOOS instruments are a specific type of applications that interface, through a variety of developer-coded methods, functions and calls, with hardware connected to the computer where this "instrument" application is running. The `CMOOSInstrument` class is therefore used to encode an interface between the MOOS star-shaped topology and either the actuators or the sensors of the AUV.

Within the authors' research group, a customized Hydroid Remus vehicle was used as a research platform for the proposed MOOS control system. The vehicle is displayed in figure 1. It is actuated by a classical three-leaved propeller thruster astern and a course rudder in the back, as well as depth control vanes. Its proprioceptive sensors include an inertial measurement unit, a Doppler velocity logger, a short baseline hydroacoustic positioning system, a GPS when in surface cruise and a 3-axial magnetic compass. Its perceptive sensors is a made-to-order BlueView 900kHz multi-beam forward-looking unit with an embedded computer pulling data off of six separate sonar transducers.

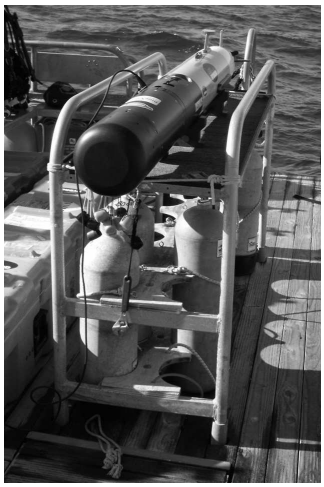

Fig. 1. The Hydroid Remus vehicle.

All of the proprioceptive data are collated and filtered with a Kalman filter aboard a black-box, non-accessible embedded computer system that encapsulates the basic functionalities of the vehicle. This primary computer, which cannot be tampered with, is supplemented with a *secondary PC-104 computer stack* that is developer-administrable, and is running a *Kubuntu Linux* operating system, the `MOOSDB` and all developed applications and instruments. The primary computer communicates with this secondary computer through a simple set of string messages piped through an RS-232 serial communications port. The incoming serial communication is accepted, broken down and re-parsed as `CMOOSMsg`-class messages by the `iRemus` instrument application. In this way, the navigation data of the Remus are made available to higher-level func-

tionalities coded in the developed CMOOSApp applications (modules). Alternatively, the iRemus instrument logs all incoming messages published via the MOOSDB by modules intending to guide the vehicle, extracts the necessary numerical information (the channel feeds) and pipes it in required format through the serial port to the primary.

The basic architecture of the proposed navigation system is displayed in figure 2.

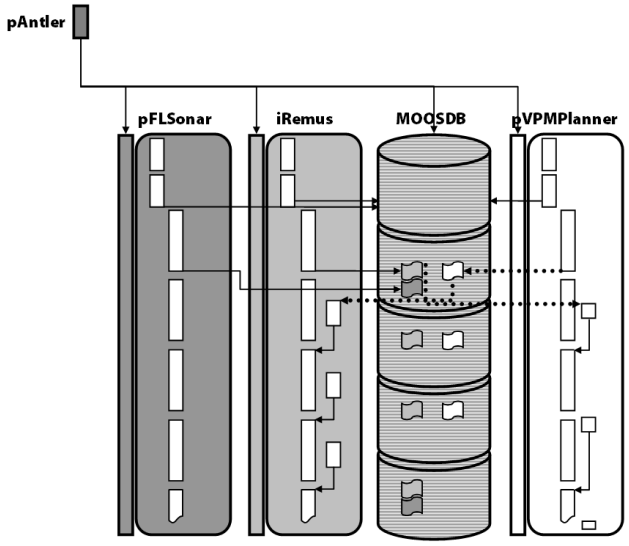

Fig. 2. The MOOS-based navigation system architecture.

The pAntler module is the one that is run initially and that is a short "starter" application which assures that the MOOSDB server and all applications start as close to contemporaneously as possible (rather than applications being started by a human operator). Also, more importantly, pAntler reads from a UTF-8 textual script which an experimenter uses to specify the parameters and starting option of *every* module that is being started. The example of a starting script is given in table 1.

After pAntler has sequentially run all the applications, each proceeds with its own *OnStartUp* and *OnConnectToServer* methods. The first process rectangle within each application in figure 2 is the *OnStartUp* which is used in a unique way by different applications. Usually, this method is responsible for reading scripts containing default settings, loading pre-existing data from certain files and ascertaining all dynamically linked libraries, ports, threads, pipelines and other functionalities are present and running as is necessary before each application is able to proceed. In *OnConnectToServer* (the second process rectangle within each application in figure 2), each module uses a sequence of *Register* methods to register which messages it will trigger to. The triggering is then taken care of by CMOOSApp-encapsulated code of the *OnNewMail* function (represented by the right-alligned, short process rectangles in each application in figure 2).

From the developer's point of view, the *OnNewMail* method of each application is started asynchronously in an encapsulated, black-box manner. It is entered into asynchronously, depending on each particular module's message subscription. The instant at which the method is entered into is determined by the "tick" of the MOOSDB server's synchronization mechanism, if new messages of the type to which an application has subscribed are present since the last synchronization.

Table 1. Example of a starting script used by pAntler.

```
ServerHost = localhost
ServerPort = 9000
Simulator  = true

Community  = remus

LatOrigin  = 36.6284333
LongOrigin = -121.9119833

//------------------------------------------
// Antler configuration block
ProcessConfig = ANTLER
{
    MSBetweenLaunches = 200

    Run = ./MOOSDB          @ NewConsole = true
    Run = ./iREMUS          @ NewConsole = true
    Run = ./pEchoVar        @ NewConsole = true
    Run = ./uXMS            @ NewConsole = true
    Run = ./iRemote         @ NewConsole = true
    Run = ./pVPMPlanner     @ NewConsole = true
    Run = ./pFLSim          @ NewConsole = true
}

//------------------------------------------
//iREMUS config block
ProcessConfig = iREMUS
{
    AppTick                 = 10
    CommsTick               = 10
    Port                    = /dev/ttyS1
    BaudRate                = 57600
    Streaming               = true
    Type                    = AUV
    Resolution              = .1
}
```

From then on, (notwithstanding explicit code inserted by the developer in his own application class inheriting from CMOOSApp) each application enters an infinite loop of calls to the *Iterate* method. This is represented by a uniform progression of process rectangles in each application track in figure 2. The loop of *Iterate*-s is escapable by a interrupt-like mechanism which a developer of every sane application within the MOOS navigation architecture is responsible for.

Typically, each application uses its *OnNewMail* to simply make an application-local copy of pertinent data encoded in the CMOOSMsg-class message, which can then be accessed in the *Iterate* method.

2. THE PVPMPLANNER MODULE

The pVPMPlanner module is an implementation of the trajectory-planning framework proposed in Barisic et al. (2007) and Barisic et al. (2008). The brief layout of the mathematics of the virtual potential (*decentralized control function*-based) method of planning the trajectory of an underwater vehicle follows:

$$
\begin{aligned}
E(\boldsymbol{p}) &= E_s(\boldsymbol{p}) + E_r(\boldsymbol{p}) \\
&= \sum_{N}^{i=1} f_s^{(i)}(\boldsymbol{p}) + \sum_{N}^{i=1} f_r^{(i)}(\boldsymbol{p}, \boldsymbol{p}_{AUV}) \\
&= \sum_{N}^{i=1} \left[f_s^{(i)}(\boldsymbol{p}) + f_r^{(i)}(\boldsymbol{p}, \boldsymbol{p}_{AUV}) \right] \\
\therefore f_{obj} &\doteq f_s + f_r
\end{aligned} \tag{1}
$$

Where:
- $E(\boldsymbol{p})$ is the artificial potential field arising due to the interaction of obstacles and the current waypoint,
- $E_s(\boldsymbol{p})$ is the *static* part of the potential contribution of each feature (obstacle or waypoint, defined by (2),
- $E_r(\boldsymbol{p})$ is the *rotor* part of the potential contribution of each feature, calculated as a "slanted potential tableau" of a gradient dependent on the proximity of the feature, and the direction as laid out in figure 3 (details in Barisic et al. (2008)),
- f_{obj} is the function that encodes both the static and rotor part of the potential distribution of each object,
- f_s is that part of the function describing the spatial distribution of the static part of the potential,
- f_r is that part of the function describing the spatial distribution of the rotor part of the potential.

The static part of the obstacles' potential distribution function is given by:

$$
f_s^{(obj)} = \exp\left(A^+ / d(\boldsymbol{p}, \boldsymbol{p}_{obs})\right) - 1 \tag{2}
$$

Where:
- A^+ is a method-independent parameter describing the severity of the obstacle's repulsion,
- $d(\boldsymbol{p}(k), \boldsymbol{p}_{obs}(k))$ is the Euclidan geometric distance at time instant k, between the AUV ($\boldsymbol{p}(k)$), and the representative point of the obstacle, dependent on its class (a vertex, barycenter, of an obstacle of one of the classes: rectangle, circle, ellipse, triangle etc.); This distance is encoded as a protected method overriding the abstract method of the superclass `Feature` for each particular subclass.

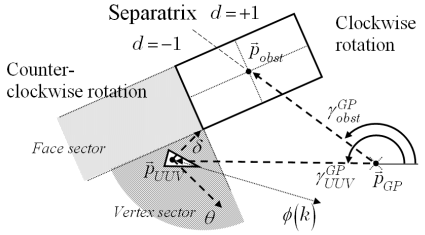

Fig. 3. The calculations determining the direction of the "spinning top" tableau of rotor potential around a rectangular obstacle.

Taking all of the above into account, the *controlling force*, a quantity that, with idealized double-integrator model of the AUV's dynamics, its thrusters would ideally need to reproduce to move the AUV along this optimal trajectory, is calculated as follows:

$$
\boldsymbol{F} = \text{bound}\left\{ \max_i \left[E(\boldsymbol{p}) - E\left(\boldsymbol{p}_\epsilon^{(i)}\right) \right] - \mu \cdot \boldsymbol{v}, F_{max} \right\} \tag{3}
$$

Where:
- bound $\left\{\mathbb{R}^n, \mathbb{R}_0^+\right\}$ denotes a function of bounding a vector's norm, bound $\{\boldsymbol{a}, a_{max}\} = \boldsymbol{a} / |\boldsymbol{a}| \cdot a_{max}$.

This *controlling force* is decomposed into the control feeds. The feeds are forwarded through `CMOOSMsg`-class message objects posted on the `MOOSDB` server by the `pVPMPlanner` application and picked up by the `iRemus` application described in the previous section. They consist of a derivative, proportional and integral feeds for both of the speed and course commands, $\{(a_c, v_c, I_{surge}), (\alpha_c, \omega_c, \phi_c)\}$:

$$
\begin{aligned}
\boldsymbol{v}(k) &= \text{bound}\left[\frac{T \cdot (\boldsymbol{F}(k-1) + \boldsymbol{F}(k))}{2} + \boldsymbol{v}(k-1), v_{max} \right] \\
\boldsymbol{v}_c &= \langle \boldsymbol{v}_c, \hat{\mathbf{e}}_1 \rangle \cdot \hat{\mathbf{e}}_1 + \langle \boldsymbol{v}_c, \hat{\mathbf{e}}_2 \rangle \cdot \hat{\mathbf{e}}_2 \\
v_c(k) &= \langle \boldsymbol{v}(k), \hat{\mathbf{e}}_1 \rangle
\end{aligned} \tag{4}
$$

Where:
- $\boldsymbol{v}(k)$ is the idealized speed *vector*,
- $(\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2)$ are an orthonormal basis defining the orientation of the AUV's coordinate system in the Earth-fixed coordinate system,
- $\langle \cdot, \cdot \rangle$ is the scalar product of two vectors, $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = |\boldsymbol{a}| \cdot |\boldsymbol{b}| \cdot \cos \angle (\boldsymbol{a}, \boldsymbol{b})$,
- $v_c(k)$ is the helm speed command at time k for the AUV.

$$
a_c = |\boldsymbol{F}(k)| \tag{5}
$$

Where:
- $a_c(k)$ is the derivative channel (acceleration) of the helm speed control subsystem at time k.

$$
\phi(k) = \text{atan2}(\boldsymbol{v}(k)) \tag{6}
$$

Where:
- $\phi(k)$ is the course command at time k,
- atan2$(\cdot)$ is the four-quadrant arc-tangent in an Earth-fixed coordinate system.

$$
I_{surge} = |\boldsymbol{p}(k) - \boldsymbol{p}_c(k)| \tag{7}
$$

Where:
- $\boldsymbol{p}(k)$ is the actual position vector of the AUV,
- $\boldsymbol{p}_c(k)$ is the commanded or idealized expected position of a *double-integrator* dynamic model of the AUV.

$$
\omega_c(k) = \frac{[\phi_c(k) - \phi(k-1)]}{T} \tag{8}
$$

Where:
- $\phi_c(k)$ is the course command at time k,
- $\phi(k-1)$ is the actual course of the AUV at time $k-1$,
- $\omega_c(k)$ is the yaw-rate command at time k.

$$
r(k) = \frac{\omega(k)}{v(k)} \tag{9}
$$

Where:
- $\omega(k)$ is the actual measured yaw-rate of the AUV at time k,
- $v(k)$ is the actual measured speed of the AUV at time k,
- $r(k)$ is the turning radius of the AUV at time k.

$$
\alpha_c(k) = \frac{r(k) \cdot a_c(k) - v_c(k) \cdot [r(k) - r(k-1)] / T}{r^2(k)} \tag{10}
$$

Where:
- $\alpha_c(k)$ is the derivative channel of the yaw-rate control subsystem at time k.

In order for potential functions to be calculated, as required by equation (1), the `pVPMPlanner` module contains the representation of the collection of obstacles. These are read from the `MOOSDB` server, where they are asynchronously posted in real time by the `pFLSonar` module. The messages contain a mathematical description of bounding boxes of perceived obstacles with four degrees of freedom, the $\{x, y, z\}$ vector of the barycenter of the cuboid and the rotation about the Z axis of the external, Earth-fixed reference frame, ϕ, according to figure 4.

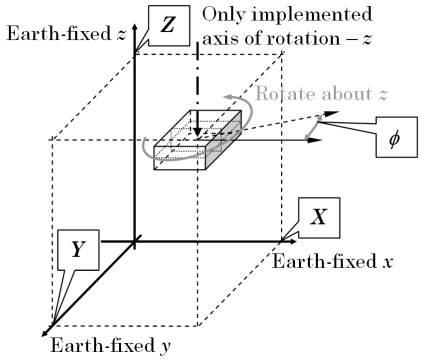

Fig. 4. The description of a box containing a generalized obstacle with four degrees of freedom in the 3D mission space.

The boxes are "flattened" in 2D since this version of the `pVPMPlanner` plans trajectories at constant depth, in order to produce a virtual potential object with the *potential distribution function* represented in figure 5.

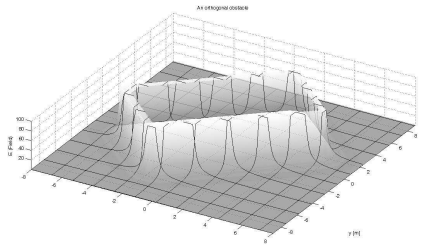

Fig. 5. The rectangle potential distribution function of the form $f_{rect} = \exp\left(A^{+}/d(\boldsymbol{p}, \boldsymbol{p}_{bc})\right) - 1$.

All of the above functionality is implemented in a class called `VPMPlanner` of which the class inheriting `CMOOSApp`, `pVPMPlanner`, is just a thin wrapper to allow the planner to function within the MOOS paradigm. In the following paragraphs, teletype text will be used for class names, emphasized teletype for method names and emphasized roman text for variable (object) names.

The collection of all perceived rectangles of this form is encoded by an object *TheWorld* of the class `World`. This class inherits from `Feature`, adding to the realization of `Feature` a protected variable *Registry*, which is a vector of records `vector<sRecord>`, with `sRecord` being a structure with the fields {*sFeatureName, pFeaturePrototype*}. *sFeatureName* is an identifier string (e.g. "Circle", "Rectangle", "Triangle"), and *pFeaturePrototype* is a pointer to a prototypical feature. By calling any one of the related family of overloaded methods ***`PutFeature`***, a feature of a certain type is instantiated. This is achieved by ***`clone`***-ing the prototype pointed to by *pFeaturePrototype* and the new instance pushed on top of the collection ***`composition`*** defined in the superclass `Feature`. Every `World`-class object is considered to be uninitialized before at least one type of feature has been registered. This needs to be performed by whichever class instantiates a member of the `World` class before this user class can correctly used the World thus instantiated.

In addition to the *TheWorld* object, the `VPMPlanner` includes an instance of a further sub-class of the `World` class, `Itinerary` – *TheItinerary*. This overrides the ***`potential`*** method inherited all the way down from the grandparent-class `Feature` so that only the potential contribution of the current waypoint is returned, rather than the sum over the collection *composition*, which is the default realization of `Feature:`***`potential`***. However, before returning a value, the method *potential* uses the public method *distance* to check if the current position of the AUV is such that in stead of the current waypoint(accessible through the protected iterator *itrGP*), the next waypoint in the Itinerary should be used (by incrementing *itrGP* if necessary, prior to returning **itrGP→`potential`*). The change in the *itrGP* is invoked by ***`distance`*** when the AUV enters within d meters of the current waypoint. The list of these "triggering" distances is kept in a protected collection of the type `vector<double>`, *trigDs*. This vector is ***`insert`***-ed into by calling `Itinerary:`***`PutFeature`***.

3. HARDWARE-IN-THE-LOOP SIMULATION

The system was first tested for sanity and functional operability by a hardware-in-the-loop test. Hydroid's off-the-shelf interfacing software that reads data directly from the primary, programmatically non-accessible computer of can be used to order the Remus to simulate cruise and operation while on dry land.

Due to the fact that the BlueView sonar head does not function out of water, the `pFLSonar` module displayed in figure 2 was replaced by a similar module `pFLSim`. This altered module interpolates the craft's position within a script-defined virtual mission space. The module then delivers simulated data on obstacles in real-time, based on a textual script. Using such a setup, the mission space displayed in figure 6 (in stead of depth, potential is the third axis, since the problem was constrained to 2D) was preprogrammed.

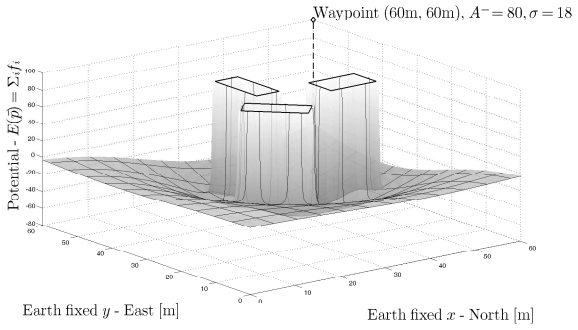

Fig. 6. The preprogrammed mission space with obstacles.

The results of simulated cruise (from the standpoint of the MOOS part of the system and specifically the `pVPMPlanner`, the situation is completely the same as if the AUV was actually cruising in open water) are displayed in figure 7.

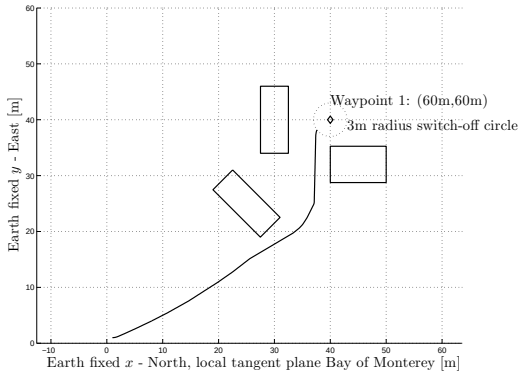

Fig. 7. The resulting trajectory.

4. CONCLUSION

A navigation system architecture relying on the Mission Oriented Operating Suite (Newman (2006), Newman (2008)) was proposed, implemented and tested for sanity and functional operability in hardware-in-the-loop simulation mode. The results are satisfactory, with the `MOOSDB` server performing well and assuring synchronization and controllability of the Remus craft. Logging and storing of both the proprioceptive data (navigation fixes, speeds and accelerations in AUV-fixed coordinate system), and the perceptive data (sonar readings of box-enclosed obstacles throughout the navigated part of the mission space) is easily achieved by the `pLogger` ready-made module shipped with the MOOS infrastructure. Renavigation in water, or in different scenarios, for purposes of analysis is possible even without the `pVPMPlanner`. This is achieved by "replaying" data obtained by the `pLogger` and stored in a UTF-8 textual file – essentially parsing it back into messages assuring the exact same time sequence. Computational load on the secondary computer installed aboard the Remus, incurred by running the `pVPMPlanner` and `pFLSonar` is light enough that both can perform in real time in actual water experiments.

Further work will consist of:

- Running a water experiment,
- Proposing and implementing a scripting language for defaults and presets for all modules involved modeled after an XML schema,
- Implementing a network-oriented application for remote interfacing, display, renavigation and post-experimental measurement and visualization from distant locations.

  This will rely on the use of a "bridge" between the Remus' off-the-shelf Ethernet switch used for accessing the secondary and primary computers, and a broader network (institution's LAN or even Internet). In preliminary scenarios, this is simply a laptop computer connected physically to the Remus' switch with one Ethernet port and running a 3G cellular model connecting to a commercial service at the same time. This commercial service can be used together with VPN software to enter the institution's domain through the Internet as a peer on a local network.

5. ACKNOWLEDGMENTS

This research was made possible through the following contributions:

(1) *Professional and technological equipment* — An FP7 Capacities Project in Response to the REGPOT-2008-1 Call, ***"Strengthening the Croatian Underwater Research Potential – CURE"***.
(2) *Scientific and professional secondment / internship grant for stay at the NPS, Monterey, California, USA* — "Unity Through Knowledge Fund", MOSES[1], Croatia.
(3) *Technical expertise* provided by the Center for Underwater Systems and Technologies, Zagreb[2].

REFERENCES

Barisic, M., Vukic, Z., and Miskovic, N. A Kinematic Virtual Potentials Trajectory Planner For AUV-s In Devy, M., editor, *Proceedings of the 6th IFAC Symposium on Intelligent Autonomous Vehicles*, on CD, National Center for Scientific Research, Laboratoire d'Analyse et d'Architecture des Systemes, Tolouse, France, 2007.

Barisic, M., Vukic, Z., and Omerdic, E. Introduction of Rotors to a Virtual Potentials UUV Trajectory In Toal, D. & Geoff, R., editors, *Proceedings of the 2nd Workshop on Navigation, Guidance and Control of Underwater Vehicles*, on CD, University of Limerick, Limerick, Ireland, 2008.

Healey, A. J. Artificial Potential Functions Chapter 3 in Roberts & Sutton, editors, *Guidance Laws, Obstacle Avoidance, Artificial Potential Functions*, part of IEEE Control Series 69, March 2006.

Healey, A.J. et al. Collaborative Unmanned Systems for Maritime and Port Security Operations In Vukic Z. & Longhi, S., editors, *Proceedings of the 7th IFAC Conference on Control Applications in Marine Systems*, on CD, Center for Underwater Systems and Technologies, Zagreb, Croatia, 2007.

Newman, P. (2006) *MOOS - Mission Orientated Operating Suite* Massachusetts Institute of Technology, Dept. of Ocean Engineering, 2006.

Newman, P. (2008). Under the Hood of the MOOS Communications API In Newman, P., editor, *http://www.robots.ox.ac.uk/∼pnewman/MOOSDocumentation/CommsArchitecture/latex//CommsArchitecture.pdf*.

[1] The Ministry of Science, Education and Sports of the Republic of Croatia.

[2] a non-for-profit non-government civil organization of underwater system engineers and scientists registered in Zagreb, Croatia.