AN ONLINE AUV TRAJECTORY RE-PLANNING SOFTWARE ARCHITECTURE BASED ON THE MOOS ⋆

Matko Barisic * **Sean P. Kragelund** ** **Theodore D. Masek** **
Zoran Vukic *

* *University of Zagreb, Faculty of Electrical Engineering and Computing, Laboratory for Underwater Systems and Technologies, Unska 3, HR-10000 Zagreb, Croatia (e-mail: matko.barisic@fer.hr, zoran.vukic@fer.hr).*
** *Naval Postgraduate School, Mechanical and Astronautical Engineering Department, Center for Autonomous Underwater Vehicle Research, 700 Dyer Rd., Monterey, CA 93940, USA (e-mail: spkragel@nps.edu, tmasek@nps.edu)*

Abstract: This paper discusses an open source navigation system architecture uniquely suited to use in autonomous underwater vehicles. It is based on the Mission Oriented Operating System developed by Newman (2006), Newman (2008). Some of its most pronounced advantages are an orientation towards providing a tool set for rapid prototyping of new control algorithms and support for development-stage software design in AUV systems. Its advantages are a direct result of the MOOS's completely modular nature, and the presented architecture has been built to preserve and, where possible, enhance this modularity. Using the MOOS design templates, control algorithms are encoded as applications "living" in separate processes of the operating system kernel. This methodology provides for a level of robustness instrumental in autonomous vehicle since failures and errors will cause only the individual modules that incur them to fail and corresponding processes to be "killed" by the operating system. Such critical occurrences 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 developed by Newman (2006) within research efforts in behavioral control of closed-space robotics. It represents a collection of code-design templates and classes which allow for the building of a software system consisting of a number of communicating controller strategies, in turn producing emergent behaviors, and allowing for intelligent decision-making. The MOOS is assumed to be built on top of a kernel that "abstracts away" the intricacies of sending and receiving data packets via consumer-grade computer communication protocols like Ethernet, Bluetooth, IEEE 802.11b/g etc.

The encapsulation provided by the MOOS design patterns greatly facilitates rapid prototyping. It should be remembered that the rapid prototyping paradigm heavily relies on the representation of *"machine-near functionalities"* at lower levels of abstraction as "black boxes", offering the capability of easy and fast redesign by reshuffling these functionalities by readable, sparse and near-English code (or visual block-programming). For architectures developed on MOOS design patterns, this translates into the natural expression of hierarchical and flow-diagrammatic ideas (familiar to control system engineers), and easy production of modules without prohibitive amounts of protocol- or platform-specific code. The architecture in stead relies on APIs to various installed sensors and actuators (or actuator control middleware) to provide the platform-specific functionality. This assumption might not have been fulfilled by experimental AUV craft first appearing in the early 80's. Nowadays, the situation, even in the off-the-shelf AUVs' market is markedly different, with almost all sensors being "smart" i.e. coming equipped with dedicated strapped-down single-board computers serving as programmable interfaces. These allow flexible, open source access to sensor's advanced functionalities and rudimentary signal processing. Also, communication pipelines used to transmit data from the smart sensors to the cen-

⋆ 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.

tral processor can be dynamically opened, closed, resized, paused, played or stopped. Use of the Ethernet protocol to connect arrays of sensors mutually, or connect star-shaped configuration of smart sensors to the central processing node is fast becoming an industry standard. Use of USB ports and IEEE1394 (FireWire) is also growing. If the architecture chosen to run high-level mission control functions were not to abstract this layer of system design, it would be expensive, unscalable and too complicated to program network functions and get data flowing through various competing protocols.

All of the above withstanding, the choice of MOOS (current version is described in Newman (2008)) as a navigation system architecture framework does prejudice the choice of components and the embeddd hardware design of the AUV's control system. The embedded control system must be realized as a fully programmable and administrable PC-standard computer. The operating system (although Windows-based builds of MOOS exist) should preferably be *unixoid* (a Linux kernel was used by the authors). Alternatively, if a different OS build of MOOS is used, a minimum level of compliance of the compiler to be used on the OS of choice with some standard GNU C / C++ Linux compiler (`gcc`) features is mandatory. An example of just such a functionality that should be assured (or emulated) is e.g. ability to run POSIX threads.

The details of a cross-platform architecture suited to the use in different AUVs, proposed by the authors is further detailed in Section 1: The Architecture. Such an architecture, was employed in use in a Hydroid Remus vehicle operated by the Center for Autonomous Underwater Vehicle Research, Naval Postgraduate School, Monterey, USA (Healey et al. (2007)), to program the online trajectory replanner based on the virtual potentials method (Healey (2006), Barisic et al. (2007), Barisic et al. (2007)) with further refinement following the propositions set out in Barisic et al. (2009). This MOOS module is described in Section 2: The `pVPMPlanner`. Experimental results are provided in Section 3. The paper proceeds to the Conclusion and the Acknowledgments, as well as a list of References.

1. THE ARCHITECTURE

Continuing on from the Introduction, MOOS is a balanced, polyvalent framework for rapid prototyping of layered and multi-path AUV control systems. It greatly facilitates the rapid programming of complex software patterns that enable intelligent and robust control by decreasing the time needed to build the class hierarchy to access various AUV resources, such as actuators and sensors, and provide multi-threaded, contemporaneous running environment for complex calculations. It provides all of this while staying true to the ANSI C++ coding standard, thereby ensuring cross-platform compilability. Also, reliance on strict ANSI C++ standards in calling conventions, naming standards and type support, allows for easy, "off the shelf" exploitation of a large pre-existing codebase of problem-focused solutions and algorithms. However, this strategy incites a cost – the clarity, efficiency, conciseness and parsimony of the code, leading to robust and fast run-time behavior are very much influenced by the quality of pre-existing class frameworks for more specific functions of the AUV's subsystems, as well as ease of access to vendor-provided functionality via APIs. Use of advanced hardware or middleware resources relies heavily either on use of third-party software, drivers or interface, or requires a prolonged effort to code such "pipelines" or interface from the ground up, in order to make efficient and robust, memory-sparse use of actuators, sensors and low-level embedded processing (DSP chips, FPGA etc.). However elegantly done, this inevitably introduces complexity, lengthens the run-time stack and piles context/namespace data on the memory heap. In order to make space for supposed heavy-duty processing, adaptation and random memory access, the MOOS kernel and the central module, the `MOOSDB`, allowing for the "backbone" information exchange protocol, only supports relatively low sample rates – up to 50Hz. Additionally, the nature of the MOOS kernel downloadable as-is is not adapted to *hardware-level enforcement* of an elected sample time (so-called "hardware-level hard real time"). The messaging system, which in MOOS serves as the one and only synchronization mechanism, is, by extension from the TCP wherefrom it draws inspiration and on top of which it builds, is in no way pegged against the hardware clock (the oscillator).

Most AUVs, however, belong to the class of low sample rate control systems, operating at sampling frequencies of ca. 10Hz. In contrast, the processors used in embedded controllers of such systems have oscillators running at the frequencies of the 100 – 1000MHz orders of magnitude. This gives the embedded computer the ability to perform billions of low-level instructions within one overarching sample period of the primary control loop (roughly equivalent to several hundreds of lines of ANSI C++ code). The effect this has on the real-time quality of even such an architecture as MOOS, which is not strictly hardware-enforced, is that it can robustly and safely be assumed to always be on time and truly real-time w.r.t. to all necessary control functions.

Regardless, the choice of MOOS (current version is described in Newman (2008)) as the architecture framework does condition the engineering considerations when planning, commissioning and running the embedded control system of the AUV. It is implicit in setting up a MOOS environment that it runs on a programmable and fully accessible (administrable) PC-standard computer running a *unixoid* operating system kernel (preferably a Linux kernel). At the very least, the operating system must be able to accept and run machine code compiled by the GNU C / C++ compiler commonly shipped and bundled with Linux distributions, thereby providing a minimum level of *unixoid* compatibility (e.g. ability to run POSIX threads).

MOOS in effect is a collection of the following, offering the complete spectrum of implementation tools for a dedicated, individualized control system of an AUV:

- A class hierarchy enabling simple inheritance-driven application development by inheriting either from the `CMOOSApp` class for MOOS *modules*, or the `CMOOSInstrument` class for MOOS *instruments*.
- An open-source complete application running the `MOOSDB`, a MOOS database server that collects, collates, indexes and distributes `CMOOSMsg`-class messages from one process to the others.

- A portfolio of conventions on coding best practices, including naming variables, methods, use of full virtual classes, prototypes and event-driven protected methods allowing for human-readable, intuitive and clearly laid out code that is easy to distribute within the developers' community.

MOOS supports a multiple-cell topology where each cell is star-shaped with the `MOOSDB` server at the hub of the star and MOOS *modules* or *instruments* at the spokes. Thereby, modules or instruments exchange information exclusively via the central server, maintaining information flow of character string or double-precision floating point data.

Each MOOS application, which is either a *module* or an *insturment* is a separate, stand-alone operating system process which maintains its own dedicated heap in the RAM memory allocated by the operating system. This allows the application, once built into `exe` files, to be "shipped out" i.e. made available for reuse, which is of the simplest possible form – running the application as-is, provides its functionality, with the application itself taking over the responsibility to locate the `MOOSDB` server, connect to it, subscribe to messages and post and receive messages of certain types.

MOOS instruments, in contrast to the more generic *modules*, coded by inheriting from the `CMOOSApp` core class, are inherited from the `CMOOSInstrument` class, itself a direct inheritor of `CMOOSApp`, are a specific type of applications that possess "black box" developer-provided ability to exchange data through an RS-232 serial communication channel. This is simply accessed in coding the AUV control system by virtue of the `m_Port` object included in any `CMOOSInstrument` or inherited class, encapsulating behavior of a COM port. Objects of classes inheriting from `CMOOSInstrument` are therefore, as a rule, used within the MOOS architecture as "bridges" between the central, star-shaped piece of architecture, and vendor-provided and -specified pipelines for communicating with payload sensors and actuators of the AUV.

The research group within which this work was performed utilizes a customized Hydroid Remus vehicle as a research platform for the proposed MOOS control system, shown in figure 1. Itis a classic cruise-class torpedo-hulled AUV propelled by a three-leaved propeller thruster astern. Steering is by a collection of four control surfaces controllable in pairs – one pair constituting a course rudder and another serving as depth control vanes. Its navigation sensors pack consists of a fused sensing bank into which feed 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. These navigation sensors are complemented by a forward-look BlueView 900kHz multi-beam sonar with a dedicated embedded computer producing imagery by collecting data off of six separate transducers.

The navigation data mentioned above are all fused together using a Kalman filter running on a "black-box" vendor-programmed embedded computer system. This system is inaccessible to developers of custom-made AUV control solutions, containing primary functionalities that ensure correct and fault-free operation of the Remus. How-

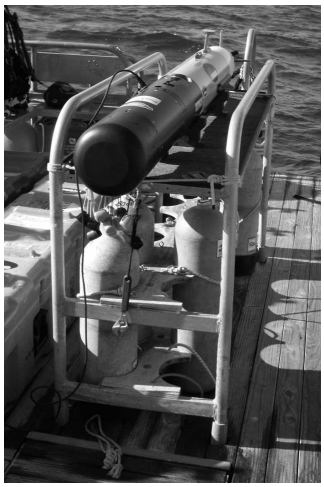

Fig. 1. The Hydroid Remus vehicle.

ever, Hydroid, the vendor of Remus vehicles provides a *secondary PC-104 computer stack* that has an open architecture and is fully administrable by prospective developers, including choice of operating system to install and all ancillary software. In the instance of the authors' Remus, the secondary computer runs a *Kubuntu Linux* operating system, on top of which the `MOOSDB` and all developed applications and instruments have been compiled and run as separate OS processes. The interconnection between the two embedded computers, wherein the primary one is basically an abstraction of the functions of the Remus vehicle itself is via a compact set of string-formatted instructions piped through an RS-232 serial communications port. This allows the "bridging", `iRemus` software module running on the secondary computer within the MOOS framework to be implemented as an object of a class inheriting `CMOOSInstrument`. The simpe task of the `iRemus` is to correctly set up the RS-232 port towards the primary computer, and to serve as a translator or a re-parser of Hydroid (vendor) -defined string messages provided by the vehicle's primary computer. This is the only data-stream coming out of an otherwise "black box" setup of the primary computer integrated with the sensors and the actuators. The `iRemus` translates the incoming datastream into `CMOOSMsg`-class messages to be posted to the `MOOSDB`. The `iRemus` also parses in the opposite direction, fetching `CMOOSMsg`-es intended for it (addressed to it) and rebuilding the commands and set-points for the vehicle in observance of the Hydroid-defined string message format that the primary computer is asble to interpret. In this way, the Remus can be commanded to navigate in a *line-of-sight*-following manner towards a certain waypoint (defined in any one of the GPS coordinate frames: decimal latitude and longitude, the local UTM sheet, or the completely localized local tangent plane). Alternatively, Remus can operate in servoing mode to follow a preset heading or achieve the commanded rate of yaw (heading rate of change). The incoming datastream from the primary computer to the secondary, is the output of a proprietary (Hydroid-developed) Kalman filter, containing the best-estimate navigation fix. The navigation fix consists of the best estimates (presupposing Gaussian noise) of $[x, y, z, a, \rho, \phi, \theta, p, q, r]$, where:
- x, y are global spherical coordinates following any one

of the: decimal longitude and latitude, appropriate UTM sheet or local tangent plane conventions,
- z is calibrated depth in meters,
- a is the altitude from bottom obtained by echo-sounding,
- (ρ, ϕ, θ) are the attitude components (yaw, pitch and roll), obtained by gyro-compasses in the Remus's inertial AHRS [1],
- (p, q, r) is the body-fixed reference coordinate frame decomposition of the velocity (surge, sway and heave) obtained by a coordinate transformation of the signals from the DVL [2].

All of these data are thereby made available to any other developer-coded `CMOOSApp` applications connected to the same instance of the `MOOSDB` server.

The latter architecture arising from this type of setup, is displayed in figure 2.

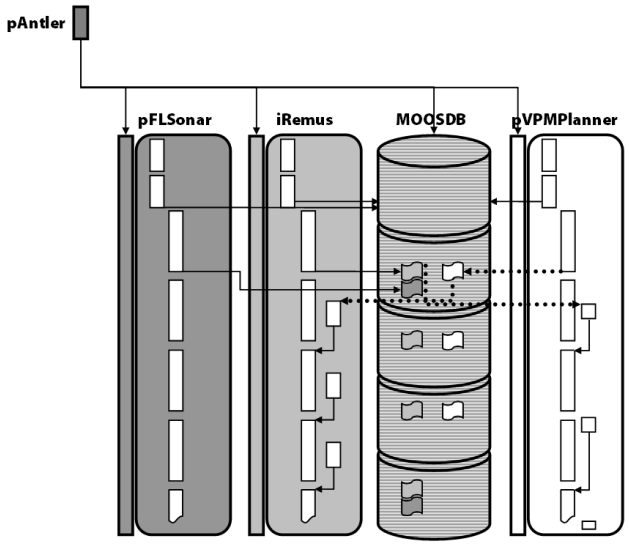

Fig. 2. The MOOS-based navigation system architecture.

The `pAntler` is the MOOS-community-provided open source application that is used as an automatic "starter" of a customizable number of other `CMOOSApp` applications, as listed in the relevant `.moos` script, which runs the `MOOSDB` server and proceeds through the scripted list of applications, forwarding if so scripted any possible command-line parameters to every application as it is started, in quick succession. The example of a `.moos` script, used to script the applications and their command-line parameters for the benefit of the `pAntler` application, but also containing separate sections that can be used to parameterize in more detail every application started with the script is displayed in Table 1.

After all the necessary applications are started, each of them enters its *OnStartUp* and *OnConnectToServer* methods. The former, *OnStartUp*, contains application-specific initialization data that either uses preset hard-coded values entered manually at compile-time, or makes use of the helper object *m_MissionFileReader*. The *m_MissionFileReader* encapsulates the parser for the `.moos` file and, when called from within a developer-coded `CMOOSApp` application it will start parsing from the "`ProcessConfig = name of application`" line. The latter of the two methods mentioned, *OnConnectToServer*, is used by each application to set up the communication with the `MOOSDB`. It will run as necessary, either at the beginning of application's life or after communication with the server is reestablished following a temporary breakdown. Within it, the application subscribes to certain `CMOOSMsg`-es by calling a block of *Register* methods. The messages themselves are fetched within the *OnNewMail* method (represented by the short process rectangles in the right column of each application track in figure 2). This method behaves asynchronously w.r.t. the main (left column) course of application's life (marked out by the left column of each application's process rectangles in figure 2). This is achieved by encapsulated code in the `MOOSDB` server and the `CMOOSApp` class's default instantiation of the *OnNewMail* method. The actual times at which *OnNewMail* runs through are dependent on the `CommsTick` (cf. table 1) to which communication between `MOOSDB` and all other `CMOOSApp` applications called by *pAntler* is synchronized. At each loop-around of the server code (determined by the value of the `AppTick` parameter of the `.moos` file), the server checks its hold of `CMOOSMsg`-es against the table of registered applications. It short-lists those applications that it needs to notify that new messages of the variety they have subscribed to have been deposited in the hold. Come the next `CommsTick`, it will notify these applications, causing *OnNewMail* of each of them to be entered into.

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
}
```

[1] attitude and heading reference system
[2] Doppler velocity logger

After *OnStartup* and *OnConnectToServer* of an application have finished executing, each application enters an infinite loop of calls to the *Iterate* method timed to the beat of the AppTick parameter in the .moos file. Naturally, *OnNewMail* gets called simultaneously, as needed and as driven by the communication functionality encapsulated in MOOSDB and the method itself. This main course of each application's life is represented by a uniform progression of process rectangles in the right column of each of the application tracks in figure 2. The loop of *Iterate*-s is escapable by a interrupt-like mechanism which must be explicitly coded in to ensure sanity and finiteness of time of execution of every application developed in the MOOS framework.

Typically, each application's *OnNewMail* is used to ensure non-volatility of the data posted to the MOOSDB and used by the application in the immediately following *Iterate* loop. time-stamped data of interest, contained in the CMOOSMsg-es that the application has subscribed to, is typically copied, in *OnNewMail* to application-local variables (variables within the scope of the application's main object, of the class inheriting from CMOOSApp). These variables, as opposed to the CMOOSMsg-es maintained by the MOOSDB, cannot from then on up until the execution of the application's own *Iterate* loop, when they are operated on, be changed. The *Iterate* can call *Notify* in order to publish CMOOSMsg-es containing results of "payload" data manipulations of a certain application.

2. THE PVPMPLANNER

The pVPMPlanner application implements virtual potential method (hence – VPM) trajectory planning theoretically developed in Barisic et al. (2007), Barisic et al. (2008) and further refined in Barisic et al. (2009). The brief layout of the mathematics, in accordance with the aforementioned references 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 *stator* 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 stator, $E(\boldsymbol{p})$ part of the obstacles' potential distribution function is defined as:

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

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 Euclidian distance (at time k), between the AUV (given by $\boldsymbol{p}(k)$), and the obstacle. The distance to the obstacle, $d(\cdot)$, is a Euclidian distance to the obstacle's representative point (geometric or barycenter or vertex, depending on the class of obstacle). The determination of the representative point (e.g. the vertex or the side of a polygon closest to position $\boldsymbol{p}$) is encoded in the function $d(\cdot)$.

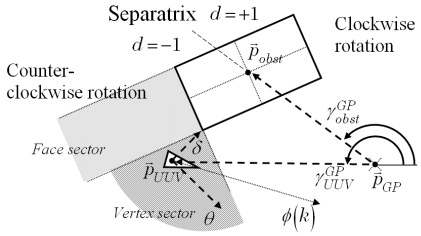

Fig. 3. The calculations determining the direction of the "spinning top" tableau of rotor potential around a rectangular obstacle.

Thus defined potential distribution functions are used to calculate the total value of the virtual potential field permeating the space around the AUV, and its numerically approximated gradient is used as the *controlling force*. This is the quantity that, modelling the AUV as an ideal double-integrator, will reproduce a trajectory optimal w.r.t. to the criterion implicitly selected by the mathematical form of the potential distribution functions. In reality, complex thruster allocation coupled with lower-level control loops of each thruster are designed for the thrusters to produce the net total force on the AUV as close as possible to the *controlling force*. the *controlling force* itself is defined 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:
- $\text{bound}\left\{\mathbb{R}^n, \mathbb{R}_0^+\right\}$ denotes a function of bounding a vector's norm, $\text{bound}\{\boldsymbol{a}, a_{max}\} = \boldsymbol{a}/|\boldsymbol{a}| \cdot \inf(a_{max}, |a|)$.

For purposes of this research, rather than employ specific thruster allocation taking into account the geometry and the rigid-body dynamics, as well as possible hydrodynamic influences on the AUV's dynamic, this *controlling force* is decomposed into control feeds to be used by a servoing type parallel (non-interactive) PI-D topology linear controller for the following state-space variables of an AUV's dynamic model:

- Heading rate of change – needing the separate control feeds for $\{\phi_c, \omega_c, \alpha_c\}$, where:
  - ϕ_c is the integral channel reference feed or *course*,
  - ω_c is the proportional channel reference feed or *yaw angular speed*,
  - α_c is the derivative channel reference feed or *yaw angular acceleration*.

- Forward speed – needing the separate control feeds for $\{I_{surge}, v_c, a_c\}$, where:
  - I_{surge} is the Euclidian distance between the actual location of the AUV and the predicted location of the ideal double-integrator model representing the integral channel reference feed,
  - v_c is the commanded *surge speed*, representing the proportional channel reference feed,
  - a_c is the commanded *surge acceleration*, representing the derivative channel reference feed.

The feeds are parsed into `CMOOSMsg`-es and posted on the `MOOSDB` server. They are fetched from the server by the `iRemus` application and parsed into messages streaming out into the primary embedded computer as explained in Section 1.

The feeds are defined as follows:

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

Where:
- $\boldsymbol{v}(k)$ is the ideal speed *vector* of the double-integrator model,
- $(\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2)$ are an orthonormal basis of the AUV's body-fixed coordinate system w.r.t. 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 surge speed – the proportional channel reference feed at time k for the AUV.

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

Where:
- $a_c(k)$ is the derivative channel reference feed at time k.

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

Where:
- $\phi(k)$ is the course command at time k – the integral channel reference feed for the course non-interactive PI-D controller,
- 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 ideal position of the double-integrator model.

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

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 angular speed command at time k.

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

Where:
- $\omega(k)$ is the actual measured yaw angular speed of the AUV at time k,
- $v(k)$ is the actual measured surge 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 feed for the course non-interactive PI-D controller, at time k.

The obstacles whose potential distribution functions are used in equation (1), are maintained in a container object within the the `pVPMPlanner`. They are fetched from the `MOOSDB` on a message-by-message basis every time `qemphOnNewMail` is called, where they are posted by the `pFLSonar` module (cf. figure 2). Each of the messages contains a mathematical description of the bounding boxes of obstacles detecetd by the use of the BlueView forward-look sonar, encoding the box's four degrees of freedom:

- $\{x, y, z\}$ – the vector of the box's geometric center.
- ϕ – the amount of the box's rotation about the Z axis in the X-Y plane, according to figure 4.

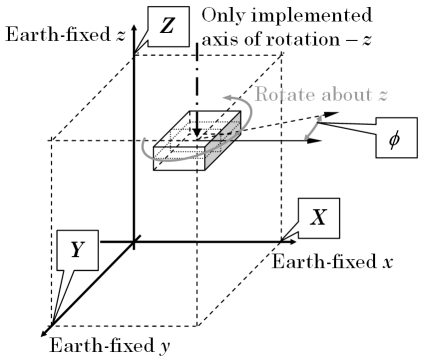

Fig. 4. The description of a box containing a generalized obstacle with four degrees of freedom in the 3D mission space.

The presented version of the `pFLSonar` bounds agglomerations of pixels by this type of 4-DOF boxes as an intermediary step in the full development of the `pFLSonar` development. This is compatible with the `pVPMPlanner`'s current level of development, allowing it to plan AUV trajectories in the 2D X-Y plane of navigation. Each orthogonal obstacle is characterized by a potential distribution function of the form presented in figure 5.

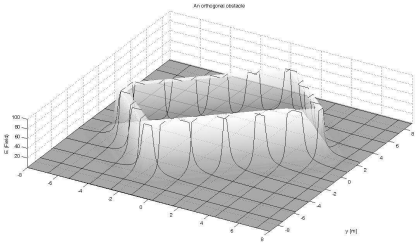

Fig. 5. The rectangle potential distribution function of the form $f_{rect} = \exp\left(A^+/d(\boldsymbol{p}, \boldsymbol{p}_{bc})\right) - 1$.

3. HARDWARE-IN-THE-LOOP SIMULATION

The presented architecture was first tested for sanity and functional operability by a hardware-in-the-loop test. Hydroid, the vendor of the Remus, provides off-the-shelf software separate from the MOOS architecture that runs on a dedicated laptop computer. The program interfaces directly with the otherwise "black box" primary embedded computer (cf. Section 1). Among its functions is the simulation mode that allows the primary embedded controller to *simulate* cruising conditions, while maintaining the *illusion – simulation* on the end of the command pipeline connecting it to the secondary computer running the herein described architecture.

Due to the fact that the BlueView forward-look sonar head, supplying raw data to the `pFLSonar`, does not function out of water, the `pFLSonar` application is for this purpose replaced in the architecture presented in figure 2, by a similar application, `pFLSim`, that presents exactly the same interface to the `MOOSDB` (posts and fetches the same type of `CMOOSMsg`-es). This altered program runs through a script-based file listing simulated obstacle positions and types and accordingly posts appropriate messages to `MOOSDB`.

With a hardware-in-the-loop setup detailed in the last two paragraphs, a hardware-in-the-loop mission was simulated for a mission space displayed in figure 6. In the figure, the third axis is used to visualize the artificial potential, since trajectory planning is constrained to 2D in the X-Y plane.

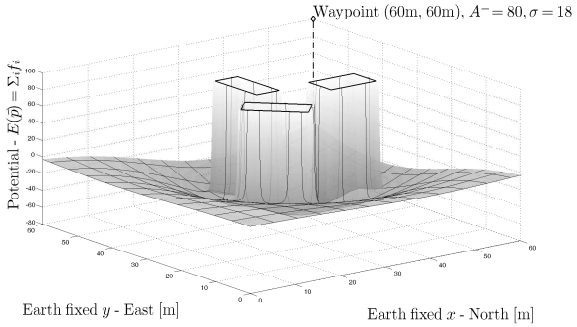

Fig. 6. The preprogrammed mission space with obstacles.

The ensuing cruise allong a trajectory optimal w.r.t. the potential map in figure 6, within the HILS [3] environment set up in the aforementioned fashion is presented in figure 7.

4. CONCLUSION

A navigation system architecture built on top of the Mission Oriented Operating Suite (Newman (2006), Newman (2008)) was proposed, implemented and tested for sanity and functional operability in HILS 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 navigation data (Earth-fixed coordinate fixes, as well as AUV body-fixed coordinate frame speeds and accelerations), and the

[3] Hardware-in-the-loop simulation.

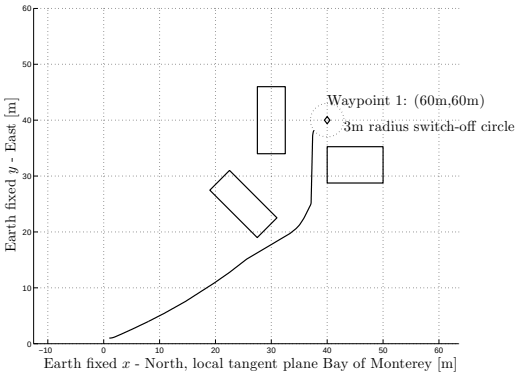

Fig. 7. The resulting trajectory.

mapping data (sonar readings of box-enclosed obstacles throughout the navigated part of the mission space) is easily achieved by using the MOOS community-provided `pLogger` open source application bundled with the MOOS. Relying on these logs, a renavigation of such trajectory is possible event without the `pVPMPlanner` (although the feedback action is lost and replaced by pure feedforward behavior), by using the `pLogger` to upload, rather than download messages used in the HILS setup, in real-time during the actual cruise. The computational load on the secondary computer running the entire required MOOS community of applications and the `MOOSDB` server aboard the Remus, is light enough that both can perform in real time in actual water experiments.

Further work will consist of:

- Experimenting underwater,
- Readjusting the `m_MissionFileReader` to parse an XML schema scripting language,
- Implementing a set of applications and tools following MOOS architecture that will allow for remote interfacing, display, renavigation and post-experimental measurement and visualization from locations connected through the TCP.

5. ACKNOWLEDGMENTS

This research was made possible through the following contributions:

(1) *Professional and technological equipment* — An FP7 Capacities project within the REGPOT-2008-1 call, ***"Developing 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 [4], Croatia.
(3) *Technical expertise* provided by the Center for Underwater Systems and Technologies, Zagreb [5].

[4] The Ministry of Science, Education and Sports of the Republic of Croatia.
[5] a non-for-profit non-government civil organization of underwater system engineers and scientists registered in Zagreb, Croatia.

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.

Barisic, M., Vukic, Z., and Miskovic, N. The Virtual Potential Field Method As a Tool For Formation Guidance of AUV-s In *Proceedings of the 11th IASTED International Conference on Control and Applications*, on CD, IASTED, Calgary, Canada, 2009.

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, Cambridge, MA, USA, 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*.