Electrical and Computers Engineering

Automation, Production and Industrial Electronics specialization

Projecto, Seminário, Trabalho de Final de Curso (PSTFC)

Multi-UAV command and control system for mixed-initiative coordinated missions

This project was developed under the AsasF project and was supervised by

Gil Manuel Gonçalves and João Borges Sousa

from the Electrical and Computers Engineering Department

from the Faculty of Engineering of University of Porto
# Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>2</td>
</tr>
<tr>
<td>Index of figures</td>
<td>4</td>
</tr>
<tr>
<td>Overview of the final project</td>
<td>6</td>
</tr>
<tr>
<td>List of acronyms</td>
<td>7</td>
</tr>
<tr>
<td>AsasF Summary/Goals</td>
<td>8</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>9</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>9</td>
</tr>
<tr>
<td>1.2 Related work</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Outline of the report</td>
<td>12</td>
</tr>
<tr>
<td>2. The AsasF project</td>
<td>13</td>
</tr>
<tr>
<td>2.1 System requirements</td>
<td>13</td>
</tr>
<tr>
<td>2.2 System Breakdown Structure</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Crucial points</td>
<td>17</td>
</tr>
<tr>
<td>3. Tools</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Piccolo</td>
<td>19</td>
</tr>
<tr>
<td>3.1.1 Operator Interface</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2 HWIL and SWIL</td>
<td>20</td>
</tr>
<tr>
<td>3.1.3 Calibration</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Seaware (Middleware)</td>
<td>26</td>
</tr>
<tr>
<td>3.3 Neptus</td>
<td>27</td>
</tr>
<tr>
<td>3.4 Hybrid Systems</td>
<td>30</td>
</tr>
<tr>
<td>3.5 Reach sets</td>
<td>34</td>
</tr>
<tr>
<td>3.5.1 Definitions</td>
<td>34</td>
</tr>
<tr>
<td>3.5.2 Why are we interested in reach sets?</td>
<td>37</td>
</tr>
<tr>
<td>3.5.3 A construction for pursuit evasion games</td>
<td>39</td>
</tr>
<tr>
<td>4. Apollo’s proposed Architecture</td>
<td>44</td>
</tr>
<tr>
<td>4.1 Hierarchical control architecture</td>
<td>44</td>
</tr>
<tr>
<td>4.2 Expandability and Modularity</td>
<td>47</td>
</tr>
<tr>
<td>4.3 Apollo</td>
<td>48</td>
</tr>
<tr>
<td>5. Apollo’s development</td>
<td>50</td>
</tr>
<tr>
<td>5.1 Hybrid System</td>
<td>50</td>
</tr>
<tr>
<td>5.2 Hybrid Controllers</td>
<td>52</td>
</tr>
<tr>
<td>5.2.1 Mission Supervisor</td>
<td>52</td>
</tr>
<tr>
<td>5.2.2 Vehicle Supervisor</td>
<td>53</td>
</tr>
<tr>
<td>5.2.3 Manoeuvre Controllers</td>
<td>54</td>
</tr>
<tr>
<td>5.3 Object Oriented implementation</td>
<td>56</td>
</tr>
<tr>
<td>5.4 Error detection</td>
<td>58</td>
</tr>
<tr>
<td>5.5 Guaranteed manoeuvres</td>
<td>59</td>
</tr>
<tr>
<td>6. Apollo tests and results</td>
<td>65</td>
</tr>
</tbody>
</table>
Index of figures

Figure 1: a) Lusitânia  b) Brutus v1  c) Brutus v2 .................................................. 14
Figure 2: Piccolo avionics ................................................................................. 14
Figure 3: Ground Station .................................................................................. 15
Figure 4: Camera .............................................................................................. 15
Figure 5: Coordinated control architecture ..................................................... 16
Figure 6: Piccolo Operator Interface ................................................................. 20
Figure 7: Exchanged information in HWIL ....................................................... 20
Figure 8: Schematic for HWIL .......................................................................... 21
Figure 9: CAN interface (left) and Piccolo (right) ............................................. 21
Figure 10: Zero Euler angle orientation between Piccolo and aircraft .......... 22
Figure 11: Picture Piccolo mounting on aircraft and servo connections ........ 23
Figure 12: Surface calibration ........................................................................... 24
Figure 13: Gains tuning ..................................................................................... 24
Figure 14: Piccolo External Interface and harness .......................................... 25
Figure 15: Publish/Subscribe Middleware ......................................................... 26
Figure 16: Communication model ..................................................................... 27
Figure 17: Neptus (Underwater vehicles) .......................................................... 29
Figure 18: Mission planning ............................................................................. 29
Figure 19: Generic Hybrid System ................................................................... 31
Figure 20: Coordinate system .......................................................................... 32
Figure 21: Hybrid automaton .......................................................................... 32
Figure 22: the set M to reach ........................................................................... 42
Figure 23: Backward reach set .......................................................................... 42
Figure 24: Hierarchical control architecture for UAVs .................................... 45
Figure 25: Real/Simulated vehicles ................................................................... 46
Figure 26: Hardware structure with extra onboard processing ....................... 46
Figure 27: Modular architecture ....................................................................... 47
Figure 28: *Piccolo – Seaware - Neptus* schematic ........................................ 48
Figure 29: Schematic for HWIL and Neptus ....................................................... 49
Figure 30: Hybrid system developed ................................................................. 51
Figure 31: Mission Supervisor .......................................................................... 52
Figure 32: Vehicle Supervisor .......................................................................... 53
Figure 33: Manoeuvre controller ................................................................. 54
Figure 34: Manoeuvre Controller ............................................................... 55
Figure 35: System implementation ............................................................... 56
Figure 36: manoeuvre classes declaration .................................................. 57
Figure 37: Manoeuvre scenario ................................................................. 59
Figure 38: The safe set ................................................................. 62
Figure 39: UAV to reach WP with wind ..................................................... 64
Figure 40: UAV to reach moving WP with no wind ..................................... 64
Figure 41: HWIL setup ............................................................................. 65
Figure 42: UAV Console, Piccolo OI ......................................................... 66
Figure 43: BAT-4 aircraft .......................................................................... 76
Figure 44: Experimental setup ................................................................. 76
Figure 45: BAT-4 after the crash ................................................................. 77
Figure 46: Operator Interface starting ....................................................... 80
Figure 47: Quick plan ................................................................................ 81
Figure 48: Simulator window .................................................................... 82
Overview of the final project

Project objectives

The project aims at developing a mixed-initiative command and control system (human operator in the control loops) for Unmanned Air Vehicles (UAVs). The system is supposed to operate with vehicles being developed in FEUP and in the Air Force Academy.

The system will be conceived having in mind its integration with an onboard autopilot. The formal model of this system will be based in dynamic networks of hybrid automata. This model will allow the encapsulation of the operational logic in modules of supervision and manoeuvre that include the several details of operator interaction. Each module is described as a hybrid automaton that interacts with a dynamic network of hybrid automata.

The system is also going to be integrated with Neptus – platform for planning, command and control of multiple autonomous vehicles – in development at the Underwater Systems and Technology Laboratory (USTL).

Project plan

The work plan is based on six steps:

- requirements analysis;
- study phase;
- formal system specification;
- implementation;
- experimental setup (simulation with software and hardware in the loop);
- module integration and testing.
List of acronyms

AGV: Autonomous Ground Vehicle
ASF: Autonomous Surface Vehicle
AUV: Autonomous Underwater Vehicle
CAN: Controller Area Network
COTS: Commercial Off-The-Shelf
CPU: Central Processing Unit
DGPS: Differential GPS
FEUP: Faculty of Engineering of University of Porto
GPS: Global Position System
GS: Ground Station
HWIL: Hardware-In-Loop
OI: Operator Interface
OS: Operating System
PC: Personal Computer
RC: Radio control
ROV: Remotely Operated Vehicle
RSSI: Receive Signal Strength Indicator
SWIL: Software-In-Loop
UAV: Unmanned Aerial Vehicle
USTL: Underwater Systems and Technology Laboratory
WP: waypoint
AsasF Summary/Goals

The main goal of the AsasF project, which is currently being developed at the Faculty of Engineering of the University of Porto, is to involve students in the design and implementation of a system of coordinated unmanned air vehicles.

The project has five main thrusts:

- design and build a family of aircrafts in composite materials for performance and durability;
- equip these vehicles with required hardware to make them autonomous;
- develop a control architecture for the system;
- implement the distributed control architecture;
- testing and evaluation.

Hence, each vehicle will be equipped with a commercial autopilot (Vanglentti et al., 2004). This allows research to be focused on the upper levels of the architecture, such as coordinated control of multiple UAVs and mission specification. The long-term goals of the project consist of developing a framework for the mixed initiative (human operators are part of the control loops) control of teams of Unmanned Aerial, Ground, Surface and Autonomous Underwater Vehicles (AUVs) with high levels of autonomy. Therefore, the following higher-level goals are to be achieved:

- Integrate Piccolo autopilot with the Neptus framework for planning and control of multiple autonomous vehicles
- Extend the Neptus framework to include UAV models and operational interfaces
- Evaluate mixed initiative operations

The AsasF project is divided in two complementary subprojects from two different areas:

1. Build, instrument and operate three individual UAVs, setting up the system that will make UAV integration, testing and control possible (Mechanical and Automatic Control Engineering);

2. Develop and implement a distributed control architecture for the system. Devise the interface between the Piccolo avionics (the chosen commercial autopilot) and the Neptus framework (Electrical and Electronics Engineering). These communications are to be achieved through a middleware publish/subscribe system: Seaware.

As already mentioned at the beginning of this document, the focus of this work is on the second subproject.

July 2006
1. Introduction

1.1 Motivation

In European Mediterranean countries, a great concern is raised every summer, with the forest fires. The potential of Unmanned Air Vehicles (UAVs) for fire detection is considerable but it has not yet been fully evaluated in these countries. This is one of the reasons why UAVs are attracting the attention of both academician and fire fighters. However, the potential applications of UAVs are not limited to fire detection. For instance, the surveillance of the Portuguese Economic Exclusive Zone represents a major challenge for the operation of unmanned vehicles, such as UAVs. This situation presents an enormous potential for the development of expertise on this hi-tech research area in Portugal.

Moreover, in the last few years, there has been a trend in the military towards autonomous air and underwater vehicles to perform coordinated missions requiring some communicated information among them. Within this sphere, UAVs are currently used to perform missions where human intervention is dangerous or complex.

One of the main advantages of UAV use is that the number of pilots involved in hazardous scenarios is much lower, which decreases the risk taken by people. Moreover, there are no constraints due to body dimensions or resistance and need for life support equipment, which allows the designers to develop much more task optimized vehicles. This has a greater importance when we take into account the fact that the pilot is the main restriction to mission limits. This is true for aggressive manoeuvres, mission duration and level of contamination. UAVs can also be tasked to perform “dirty”, “dull”, and “dangerous” operations, such as border patrol and maritime search and rescue.

UAVs are attracting a great amount of current interest in the navigation and control communities. Their design and control facilitate the exploration of many exciting new research areas in control theory, ranging from low-level flight control algorithm design to high-level multiple aircraft coordinated mission planning.

There are many applications where “coordinated” control of UAVs is desirable, such as the coordinated operation of several UAVs, where teams of vehicles are

Nowadays, UAVs are having a growing importance since they can perform a great number of low-cost tasks, which can be divided in two main categories:
• Rescue, civil protection & commercial applications
  - flying eye for rescue services
  - hazard inspections
  - vigilance
  - aerial photographs, terrain mapping
  - farming monitoring
  - atmospheric phenomenon research
  - communications
• Military & governmental applications
  - flying vanguard
  - short range reconnaissance
  - search mission
  - traffic control
  - border control
  - military vigilance
  - critical area monitoring
  - fire prevention
  - maritime and aquatic life surveillance

1.2 Related work

Research and development on autonomous systems is an emergent area, leaving a lot of room for new ideas and projects. Currently, research is focused on applications involving either autonomous ground vehicles (e.g. the DARPA Grand Challenge), Autonomous Underwater Vehicles (AUVs), autonomous surface vehicles (ASV) and, of course, on Unmanned Air Vehicles (UAVs). These applications share common goals, such as:
  • Development of autonomous systems with an user-friendly operator interface, which minimizes the need for specialized people on the field;
  • To allow a single-user to control a whole fleet of vehicles;
  • Research on distributed task assignment;
  • Mixed-initiative (human operator in the control loop) environments;

Existing UAVs such as the Predator and GlobalHawk require a team of more than a dozen trained operational people, including pilots, navigators, sensor managers, intelligence officers etc. to operate a single UAV. However, recent advances in UAVs research have already presented one system to have demonstrated in-the-air effective task allocation and collaboration (Ryan et al, 2006). The capabilities displayed by this
Multi-UAV command and control system

system include single-user control of a fleet of aircraft, distributed task assignment, and vision-based navigation. Ongoing efforts are focusing on demonstrating similar technology for hundreds of vehicles.

Research in (Lee et al, 2003) shows a strategy of path-planning for an UAV to follow a ground vehicle. This ground vehicle may change its heading and vary its speed. Here, the UAV will maintain a fixed airspeed and will manoeuvre itself to track the ground vehicle. (Girard et al, 2004a) uses selected case studies as a motivation, and examines emerging results in networked multi-vehicle systems. Some current issues common to networked multi-vehicle systems are presented, as well as their tackling approach.

In (Girard et al, 2004c), a hierarchical control architecture for a system that does border or perimeter patrol using UAVs is proposed.

At the Faculty of Engineering of University of Porto (FEUP) there is a considerable expertise in the coordination and control of ground, underwater and surface autonomous vehicles. This expertise has been achieved through extensive experimental work in operational deployments. There is also a considerable expertise in task planning and execution control for teams of UAVs. This has been done in cooperation with leading US and EU universities, and also with the Portuguese Air Force Academy. Last year, two teams of FEUP students from NAAM (Nucleus of Aeronautics, Aerospace and Modelling) designed and built two aircrafts for the Portuguese Air Cargo Challenge. The Asasfl project aims to combine the expertise acquired with aircraft design with the existing expertise on the coordination and control of multiple vehicles to develop an innovative program of research and development on cooperative air vehicles.

The Underwater Systems and Technology Laboratory (USTL) from FEUP has developed a mixed-initiative environment for the coordination and control of teams of multiple autonomous and semi-autonomous vehicles, called Neptus (Dias et al, 2005). In the context of this work, the kind of operations defined involve a wide variety of possible interactions between the pilot (human or automated) and the vehicles, including: pre-mission setup and preparation of a vehicle (or multiple vehicles) mission; real-time data acquisition and visualization; pilot intervention during the mission execution (mixed initiative operation); coordinated control of multiple vehicles (fleet control); and post-mission review and data analysis. Neptus is intended to be used in several scenarios with different types of vehicles.

Important steps to increase the expansibility and modularity of a multiple vehicle system were also taken at USTL by the development of Seaware. This is a
Multi-UAV command and control system

publish/subscribe middleware toolkit designed with heterogeneous, dynamic, real-time communications application environments in mind. Seaware uses the RTPS (Real Time Publish Subscribe) industry standard as main underlying network support and provides a flexible high-level application programming and configuration interface (Marques et al, 2006).

1.3 Outline of the report

This report is organized as follows: Chapter 2 describes the FEUP AsasF project, the proposed work, requirements and structure in addition to our contributions to this project. Crucial aspects of this type of project are also discussed. Chapter 3 introduces the technologies and tools required as well as some background theory. This will lay the foundations for the proposed architecture in Chapter 4. Chapter 5 demonstrates how this architecture was implemented. In Chapter 6 we concentrate on testing and evaluation. Finally, Chapter 7 discusses the project results and suggests future enhancements. In Appendix A we describe the aspects of the cooperation between FEUP and University of California, Berkeley which proved relevant to our developments. Appendix B includes a Guided Tour of the system developed in this project.

It should be pointed out that the line of development of the project did not exactly follow the structure of this report. Instead, we did it in an incremental fashion and in an iterative manner. We started by defining goals and requirements as well as by learning details of the equipment to use throughout the project. After that, we went through a study phase in order to design the architecture for the system. The next steps were to develop an object oriented software implementation. The test and evaluation process came last. At the end, we extended the system capabilities with a manoeuvre to reach a set with guaranteed results in the presence of disturbances. This was done in the context of results from the theory of differential games.
2. The AsasF project

2.1 System requirements

The Systems Engineering Process (IEEE, 1999) is being used in this project. In this process, each stage of the systems life cycle is divided in three activities: Requirements analysis; Functional analysis; and Synthesis and design. System life cycle stages range from “System definition” to “Costumer support”, including “Subsystem definition” and “Production”.

This incremental development process (in every iteration the requirements are harder and the system has more functionalities) has the advantage of going mostly through “easy steps”, testing every new functionality thoroughly to minimize the number of unpredicted problems and have a faster development time.

Every engineering project in its early stages starts with the definition of the system concept and functional requirements. On these first iterations of the development cycle, we are not going to make hard demands on the system. Thus, our UAV will not have long periods of operation and will be manually deployed and landed. The most important functional requirements for the system at this stage are as follows:

- Minimum payload of 4Kg
- Modular, flexible, robust
- Low cost
- COTS components
- Easy control and operation
- Autonomous and remotely operated operation
- Air data (Altitude, Air pressure, airspeed, GPS, RPMs)
- Video acquisition and real time transmission
- Operational console capable of specifying new waypoints for coordinated flight

Some of these requirements are met by the autopilot, which has already built-in interfaces for sensors to be used in the estimation of air data.

Note that we want our system to use as much as possible Commercial-Off-The-Shelf (COTS) hardware and software components in order to increase flexibility and portability of developed UAVs. Moreover, it creates the opportunity to easily port what is done from one platform (UAV) to another.
2.2 System Breakdown Structure

Platform: For the first tests we have a commercial aircraft model, which guarantees the needed controllability and reliability (fig. 1a). The other two aircrafts are part of NAAM projects. One of them was built last year by the Brutus team and presented good stability and a significant capability to carry payload (fig 1b). The third aircraft is the next version of Brutus which we expect to have better performance and reliability (fig 1c).

We will use the same engine for the three aircrafts: a 15cc, 2.9HP, 2 stroke engine, OS 91-FX. This will allow us to have a takeoff payload weight larger than 5kg.

![Figure 1: a) Lusitânia b) Brutus v1 c) Brutus v2](image)

Avionics: We choose the Piccolo Autopilot from CloudCap (fig 2) (Vaglienti et al., 2004) for our developments. It is really as small as it looks with its 212g and dimensions: 12.2x6.1x3.8cm. This is an autopilot for small aircrafts and includes everything it needs to perform autonomous navigation and dynamics control, such as:

- Possibility of controlling 10 servos (to control ailerons, elevators, rudder and engine)
- 3 gyroscopes, 2 two axis accelerometers (to measure aircraft attitude)
- static pressure sensor
- total pressure sensor
- temperature sensor (these last three are used to measure altitude and airspeed)
- GPS

![Figure 2: Piccolo avionics](image)

With this autopilot, a single user can control a number of Piccolo-equipped UAVs using the groundstation and accompanying operator interface software. The operator interface displays telemetry from each UAV and sends low-level commands to the autopilots over a 2400 MHz radio link.
Piccolo’s communications System Development Kit (SDK) provides a set of functions that allow packets to be exchanged with the ground station. These packets include sending aircraft telemetry such as air pressure and velocity, or receiving commands such as waypoints.

**Ground Station:** The ground station (fig 3), which comes with the Piccolo Autopilot handles the data exchanges and communications between the operator interface and the avionics.

![Ground Station Image](image)

**Figure 3: Ground Station**

**Payload:** We have studied several options to incorporate a camera in the system (which can be found in the AsasF website), in order to give the operator a video feed in real-time. The system acquired was from wirelessvideocameras.com, which includes a small transmitter, a directional receiver and a small camera (36mm x 36mm x 33mm). This is done through a reliable wireless transmission system in the 2.4GHz frequency, with a range of 8Km.

![Camera Image](image)

**Figure 4: Camera**

**Future system architecture:** Piccolo’s development system allows us to perform both software and hardware-in-the-loop (HWIL) simulation. This is a cornerstone of Piccolo’s development environment, since the simulator allows the aircraft control laws and mission functionality to be tested without risking hardware in real flight test. It also provides an ideal training tool that can be used in the laboratory. Although HIL simulation cannot replace flight-testing, it measurably reduces the likelihood of failure by detecting bugs and deficiencies before the hardware is put at risk.

On the long term, one of the main goals of our UAVs research is to integrate the aircrafts in a team of heterogeneous autonomous vehicles that is also being developed at the University of Porto. The desired architecture to achieve in the near future is depicted in figure 5.
Figure 5: Coordinated control architecture
2.3 Crucial points

Since this is the first research project with UAVs, the main tasks are oriented to understand the whole system and provide a background that will make any system module implementation as simple as possible.

As with any other system, the vehicle navigation subsystem is subject to failure. In order to make it as much as possible immune to unexpected events, we need to analyze where these failures can occur and eliminate or reduce their occurrence. This analysis is part of the Systems Engineering process. Communication with manufacturers and experienced people on the area along with extensive testing of components, subsystems and systems is essential for this purpose. Thus, in spite of a little increase on the initial development time, this should reveal a great advantage on the long term by minimizing the cost and re-development time in case we face faulty behaviours.

Radio frequency interference: Care should be taken on the possibility of interference on the transmission bandwidth. When the vehicle is being manually controlled from the ground, interference on the communication channel can be fatal for the aircraft. However, to prevent this from happening, the autopilot uses a radio with frequency hopping spread spectrum system, which means that it doesn’t use a single frequency, but instead hops along a pattern of frequencies. One radio in any network is a master (always the ground stations) and slaves in the network synchronize their hopping to match the master. In order to operate multiple ground stations in close proximity they must be using different hopping patterns to avoid interference.

Platform reliability: Having in mind the low reliability of new platforms and the lack of experience in UAV integration, there is the need to create some tasks to validate the aircrafts operation capability. We have to take into account that two of the platforms are recent designs, which can have undesirable behaviours. These unexpected situations can emerge due to motor vibrations effects on the structure or on electronic devices, material interferences with electronic devices or any other unpredicted design fault.

Experimental plan: A list of experimental steps is being developed in order to build a roadmap for the first autonomous experimental flight. This draft roadmap is showed next:

- Flight controller validation through software-in-loop and hardware-in-loop simulation
• Aircrafts behaviour evaluation on different mechanical, aerodynamic and electrical environments
• Aircraft testing with manual piloting
• Thoroughly follow Piccolo’s pre-flight checklist
• Aircraft testing with autopilot
• First autonomous flight
3. Tools

In this section we describe the tools and technologies used to develop the Apollo system. This is the name we gave to our developed system and is described later. We used 3 tools in our developments. The Picollo auto-pilot and its development system. The Seaware middleware framework for real time publish subscription. The Neptus command and control framework. Finally, we used Seaware to provide the communication support and integrated Apollo in Neptus.

3.1 Piccolo

Piccolo is the name of the CloudCap autopilot, which goes onboard the aircraft. It includes all necessary sensors and communication links to perform an accurate control of the vehicle. Communications between the ground station and Piccolo are established through a wireless 2.4 GHz link. On the ground, the Ground Station (GS) that allows access to all onboard information. This is achieved through a computer running the Operator Interface.

3.1.1 Operator Interface

The Operator Interface (OI) provided by Piccolo manufacturer allows monitoring, command and control everything going on in the autopilot. A detailed guide can be found in (Vaglioni et al, 2005a). The image below on the left shows the main OI screen (Telemetry) and the provided information. The image on the right shows the Map tab where you can assign waypoints, routes and watch the aircraft behaviour. You can also add a georeferenced image to see any aircraft current position on the map.
3.1.2 HWIL and SWIL

Hardware-In-the-Loop

A cornerstone of Piccolo’s development environment is the hardware-in-the-loop (HIL) simulator. Aircraft control laws and mission functionality can be tested without risking hardware in flight test. Although HIL simulation cannot replace flight-testing, it measurably reduces the likelihood of failure.

Piccolo’s HIL simulator is based upon the external CAN interface (Figure 7: Exchanged information in HWIL, 9). Piccolo sends servo control commands (actuators) over the CAN bus, and accepts external sensor data (attitude, behavior) on the CAN bus.

![Diagram showing the exchange of information in HWIL](image)

To perform HWIL, you need a computer connected to a CAN interface, running a real-time simulator that accepts servo commands. Piccolo is connected through a wireless link to the groundstation which, in turn, is connected through a serial port to the...
operator interface computer (see Piccolo Quick Setup Guide, 2002, for more information on HWIL).

![Figure 8: Schematic for HWIL](image)

![Figure 9: CAN interface (left) and Piccolo (right)](image)

**Software-In-the-Loop**

Software in loop simulation (SWIL) provides the same functionality as the hardware-in-loop simulation, with the difference that the Piccolo and groundstation firmware run on a PC instead of the actual avionics and groundstation hardware. Therefore, the method has the advantage that besides one or two networked PCs, no other hardware is necessary.

- The GroundstationPC and PiccoloPC software applications will replace the groundstation and Piccolo avionics.
- A network TCP/IP socket on port 3000 replaces the CAN-bus connection between the Simulator and the avionics.
- The wireless connection between the avionics and the groundstation is replaced by a multicast network connection to address 224.0.0.1 on port 4004.
• The serial connection between the groundstation and the Operator Interface is replaced by a CommSDK network connection on port 2000.

For a Guided Tour on how to run the Piccolo system see appendix B.

3.1.3 Calibration

Orientation
In order to mount Piccolo in the airframe, you should follow the instructions in (Vaglienti et al, 2005b).

However, one critical detail when mounting Piccolo is its orientation. This orientation with respect to the aircraft has to be orthogonal, and the position where all Euler angles are zero is the one indicated in the picture below. Thus, the Piccolo External Interface points backwards, and its larger side is pointing up. Note that the Euler angles can be adjusted in the Sensors tab. Here, a simple drawing illustrates the Piccolo orientation in respect to the aircraft.

![Euler transformation diagram]

Figure 10: Zero Euler angle orientation between Piccolo and aircraft
Surfaces

After mounting it you need to calibrate the aircraft surfaces. This is done in the Surfaces tab (Fig. 12, left). For each surface, you should send a pulse signal in microseconds and measure the corresponding angle deviation in degrees. The sequence of steps to follow is:

1. Select a surface (ailerons, rudder, elevators, throttle, etc)
2. Send a 1500us pulse
3. Mechanically adjust the servo so that to this pulse corresponds 0 (zero) angle
4. Send other pulses and measure the angle deviation having in mind that a down deflection is positive. Identically, a right deflection is also positive.
5. Fill the table with the measured values

Piccolo also has the possibility of controlling aircrafts with a V tail. You just need to specify whether the tail is an upright V or an inverted V (figure 12, bottom right square).
Gains

One of the most critical parts when inserting Piccolo in a new aircraft is to tune its control loop gains. This can be done in the Gains tab (Fig. 13 on the right). The procedure for tuning should follow the steps described in (Initial Flight Test Cards, 20005). To enable/disable the control loops you should go to the Commands tab (Fig. 13, left).

The first step before changing the gains is to trim the vehicle. When the vehicle is flying straight, at a constant altitude and airspeed, press the Capture Trim button (Fig. 13, bottom right). Afterwards, you may begin the fine tuning. This procedure is exactly the same whether the vehicle is in the air or is doing HWIL.
Harness

The picture below shows the Piccolo harness and its connections. It has a 12V and a 5V input, servo outputs, a CAN interface for HWIL, a programming cable and a payload serial port.

Figure 14: Piccolo External Interface and harness
3.2 Seaware (Middleware)

Seaware is a middleware that addresses the problem of communications in heterogeneous environments with diverse requirements. Seaware (Marques et al, 2006) adopts a publish/subscribe based messaging, defined by anonymous message exchange between data subscriptions and publications. Each application dynamically registers itself, specifying the topics it wishes to publish and subscribe, without the need to know in advance who its peers are or where they are located. Figure 15 illustrates this network environment. On a system like this, the peers do not need to know each other.

![Publish/Subscribe Middleware](image)

Figure 15: Publish/Subscribe Middleware

A few important Seaware concepts are going to be detailed next:

- **Messages** in Seaware have to be identified by a topic.
- **Topic** is a subject name used for coupling peers in a network.
- In order to place a message on the network, there must be a topic message issuer (publisher) and a topic message receiver (subscriber).
- A **topic domain** is an identifier for the name space of one or more topics. A publication-subscription pair couples together only if they refer to the same topic and belong to the same topic domain.
- A **node** consists of a data exchange module defined as having a set of publishers and subscribers. For instance, a node can be a UAV, publishing its telemetry information and subscribing altitude commands.
- Finally, an **application** refers to the address space of a process with network connectivity and on which a number of nodes may be active at any moment.
The coupling of these entities is shown in fig. 16. In the sample model there are two applications App1 and App2, each with two active nodes: App1 – N1 and N2; App2 – N3 and N4, operating with three topics (A, B and C) and three topic domains (D1, D2 and D3).

3.3 Neptus

The Underwater Systems and Technology Laboratory (USTL) from FEUP has developed a mixed-initiative environment for the coordination and control of teams of multiple autonomous and semi-autonomous vehicles, called Neptus (Dias et al, 2005). The execution of operational missions with the USTL vehicles is the main motivation for the development of the Neptus framework. In the context of my work, there is a wide variety of possible interactions between the pilot (human or automated) and the vehicles. These include:

- pre-mission setup and preparation of a vehicle (or multiple vehicles) mission;
- real-time data acquisition and visualization;
- pilot intervention during the mission execution (mixed initiative operation);
- coordinated control of multiple vehicles (fleet control);
- post-mission review and data analysis.

Thus, Neptus framework was designed to fulfil the operational mission requirements listed below:

- An application to define the environment of the operational mission. Including navigation references, bottom profile, obstacles, and mark points;
- Allow for the mission programming;
Multi-UAV command and control system

- A console that establishes the link between the support computer and the on-board computer. In some vehicles, the console should enable users to send joystick commands and visualize sensors' data in real-time;
- A simulation platform to allow users to verify the conformity of the mission program. This tool will give the user the chance to debug its own mission plan;
- A tool for mission review and analysis of sensors data;
- A repository for gathered data with associated query services.

Neptus is intended to be used in several scenarios with different types of vehicles. Thus, the framework has to be designed such that some portion of the framework (modules) may be used separately of the other modules. These modules must be designed to be easily integrated in already existing software.

Hence, in light of middleware concepts, Neptus is going to work as an application that will communicate to, command and control vehicles through Seaware. Therefore, Neptus can have as many middleware nodes as needed, depending on the number of vehicles involved in the mission. For instance, Neptus can be controlling 2 UAVs, a AUV and a ASF. Thus, it will create four nodes, one for each vehicle. Each of these nodes is characterized by a topic domain identifying the vehicle to allow for a set of messages to be exchanged between each vehicle and Neptus.

Screenshots of Neptus are shown below, where you can see its usage for underwater vehicles (ROV and ISURUS). Figure 17 shows some vehicle information (left) along with a console demonstrating real-time underwater images and a map of the mission area (centre and right). On figure 18 we find the mission specification as a set of manoeuvres as well as 3D views of ISURUS.
Multi-UAV command and control system

Figure 17: Neptus (Underwater vehicles)

Figure 18: Mission planning

July 2006
3.4 Hybrid Systems

In order to pursue with the coordinated control of multiple UAVs we can think of UAVs as being vehicles to which we want to apply a high level coordinated control, and no longer need to worry about individual controllers. The control structure of this application is going to be modelled in the framework of dynamic networks of hybrid automata (Henzinger, 1996), (Girard et al, 2004b, c). These are formal models for systems exhibiting hybrid behaviour (hybrid systems), consisting of continuous-time phases separated by discrete-event transitions. These systems combine time-based signals with sequences of events.

The basic motivation behind formal methods is that any rigorous reasoning about systems must be based on an unambiguous description. This is particularly true if one wants to automate such reasoning. The key point of formal methods is their mathematical exactness. It is unambiguous how the system is going to “behave”.

Control processes on networked vehicle systems typically run on one or more microprocessors. In turn, the activities of the environment in which the vehicles operate can be discrete or continuous (the aerodynamics of an aircraft are continuous, while error signals sent by manoeuvre controllers would be modelled as a discrete event). The overall system thus forms a so called hybrid system, which is characterized by the coexistence of discrete and continuous behaviours. When there are several possible modes of control, a supervisory agent monitors the condition of the system and switches between control blocks as appropriate.

Hybrid systems are usually represented as a set of states and a description of how to switch from one state to another. Switching from one state to another is called a transition. During the continuous phase (in each state), we need to define how the continuous variables evolve. This can be done by using differential equations or just by specifying a continuous-time system (e.g. monitoring).
Formal model

A HybridSystem is a 5-tuple (Varaya, Lee, 2001):

\[
\text{HybridSystem} = (\text{States}, \text{Inputs}, \text{Outputs}, \text{TransitionStructure}, \text{initialState})
\]

Where \text{States} = \text{Modes} \times \text{RefinementStates} is the state space. \text{Modes} is the finite set of discrete states or modes and \text{RefinementStates} is the state space of the time-based system.

\text{Inputs} = \text{InputEvents} \times \text{TimeBasedInputs} is the space of inputs. \text{InputEvents} are discrete input events while \text{TimeBasedInputs} are the space of input values to which the time-based system reacts.

\text{Outputs} = \text{OutputEvents} \times \text{TimeBasedOutputs} is the space of outputs. \text{OutputEvents} are discrete output events while \text{TimeBasedOutputs} is the space of continuous output values.

The transition structure determines how a mode transition occurs and how the continuous-time system changes over time. Transitions are usually labelled with three distinct items: a guard, an action and a reset assignment.
Multi-UAV command and control system

- A transition can only be taken if its guard is true.
- The action associated with the transition is executed if the transition is taken.
- The reset assignment can assign arbitrary values to the continuous variables. A common use of the reset assignment is resetting clocks to zero.

In addition, states are labelled with invariants. If an invariant is violated, a transition out of the corresponding state must be taken. For example, consider a simple algorithm for a UAV manoeuvre controller (goTo). The control algorithm has two states, A (go), and B (loiter), as shown in figure 11. The goal of the control system is to make the UAV reach the commanded waypoint and then circle around it. The kinematic model to represent the aircraft in each of the states is explained below. The invariants of the system are given by: Inv(A)=(distance>60), Inv(B)=(distance<60). The guards are (A,B)→DONE.

![Figure 20: Coordinate system](image)

The directions x and y are in a frame that is fixed with respect to the ground. The control variable is only $u_2$, the rotational speed. For simplicity, we have fixed the longitudinal speed and use only the rotational speed $u_2$ as a control variable. Moreover, we are restricting the aircraft movement on the horizontal plane only. This is a good assumption, as wireless communications between aircraft are much more reliable if all aircraft are in the same geometric plane. The hybrid automaton\(^1\) that models this system is represented next:

![Figure 21: Hybrid automaton](image)

\(^1\) A hybrid system is a system in which discrete and continuous dynamics co-evolve. A hybrid automaton is an abstraction that models hybrid systems. For example, a hybrid automaton may model a manoeuvre controller or a supervisor.
On the example shown, the automaton goes from A to B when $distance<60$ and sends an output to the supervisor\(^2\) above saying it has finished (DONE). The system goes from B to A if somehow happens that $distance>60$ or a new waypoint command is given.

It is useful to think of networked vehicle systems as dynamic networks of hybrid systems. The "dynamic" nature of the problem allows one to create and/or destroy components in real time. For example, this is useful to model vehicles joining or leaving a group of cooperating vehicles. The system designer gains a lot of flexibility. This is important because typically in networked vehicle systems, the formation changes dynamically, and so do the communication patterns between vehicles. The ability to create and destroy components and communication links between components at runtime is therefore of paramount importance.

**Architecture of hybrid systems**

In general, the system architecture for networked vehicle systems is organized in hierarchical layers. Each layer provides different kinds of services and behaviours. Here, the lower layers of the hierarchy consist of continuous controllers that interact with the sensors and actuators to produce the desired positioning and tracking performance. Higher layers of the hierarchy will be modelled by discrete event systems used for manoeuvre coordination, fault identification and reconfiguration. Most of the communication at this level is in the form of “one-time” questions and answers. The answers to the questions help determine what the global behaviour of the system must be.

It is interesting to see that the Apollo system follows the steps described in (Girard, 2002 e). A detailed description of the hierarchical system architecture is going to be presented in Chapter 4.

\(^2\) By supervisor we mean the controller that concerns itself with the high-level goals, control and communication protocols for the vehicle formation.
3.5 Reach sets

3.5.1 Definitions

Consider the following model of a system whose state $x$ evolves in $\mathbb{R}^n$

$$\dot{x} = f(t, x, u), u \in U(t) \subset \mathbb{R}^p$$  \hspace{1cm} (1)

where $f$ satisfies the conditions for existence and uniqueness of the ordinary differential equation and $u$ is our control.

**Forward reachability**

Forward reachability concerns the set of states that can be reached from an initial state/set of states. For instance, consider the system described by equation (1).

**Definition 1 (Reach set starting at a given point)** Suppose the initial position and time \{x_0, t_0\} are given. The reach set $R[\tau, t_0, x_0]$ of system (1) at time $\tau$, starting at position and time \{x_0, t_0\} is given by:

$$R[\tau, t_0, x_0] = \bigcup \{x[\tau], u(s) \in U(s), s \in (t_0, \tau]\}$$ \hspace{1cm} (2)

where $x[\tau]$ denotes the state of the system at time $\tau$ when driven by a given control $u(\cdot)$. This expression means that the set of reachable states starting from \{x_0, t_0\}, is the union of all possible states at time $\tau$, $x[\tau]$ which can be reached under feasible controls $u(\cdot)$ ($u(s)$ in U) when the time goes from $t_0$ to $\tau$.

**Definition 2 (Reach set starting at a given set)** The reach set at time $\tau > t_0$ from set $X_0$ is defined as:

$$R[\tau, t_0, X_0] = \bigcup \{R(\tau, t_0, x_0) \mid x_0 \in X_0\}$$ \hspace{1cm} (3)

This expression means the same as the previous one when the system starts from a set ($X_0$).

**Definition 3 (Reach set under adversarial behavior)** Consider now the case of adversarial behavior:

$$\dot{x} = f(t, x, u, v), u \in U \subset \mathbb{R}^u, v \in V \subset \mathbb{R}^v$$ \hspace{1cm} (4)

where,

- $u$ is our control.
- $v$ is controlled by an adversary. We don’t know what the adversary will do (you may assume the worse case scenario).
The reach set $R[\tau, t_0, x_0]$ of system (4) at time $\tau$, starting at position and time $\{x_0, t_0\}$ is given by:

$$\forall u(.), u(s) \in U(s), \forall v(.), v(s) \in V(s): R[\tau, x_0, t_0] = \bigcup \{x[\tau], s \in (t_0, \tau)\}$$  \hspace{1cm} (5)

**Backward reachability**

Backward reachability concerns the set of initial states that could originate from the final set. Until now we have discussed the problem of ‘forward’ reachability – we integrate the differential equation ‘forward’ in time. Now we briefly discuss ‘backward’ reach sets.

**Definition 4 (Reach set ending at a given set)** Given a set $X_f$ and the dynamic system (1) the reach set $R[\tau, t_f, x_f]$ of system (1) at time $\tau$, ending at position and time $\{x_f, t_f\}$ is given by:

$$R[\tau, t_f, x_f] = \bigcup \{x[\tau], u(s) \in U(s), s \in (\tau, t_f)\}$$  \hspace{1cm} (6)

This means the set of possible states from which the system could have started is $R[\tau, t_f, x_f]$. Thus, if the system ends at $\{x_f, t_f\}$, the starting set is the reunion of all possible states, $x[\tau]$, under control $u(.)$ at time $\tau < t_f$.

We can also introduce definitions for the backward reach set under adversarial behavior.

**Reach sets, invariance, and control**

We are interested in understanding the geometry of reach sets and the relationships between reach sets, control, and invariance. In what follows we introduce some simple examples which illustrate these relations in a geometric setting.

**Example 1** Consider:

- The linear system ($B$ is the identity $2 \times 2$ matrix), whose state $x \in \mathbb{R}^2$:
  $$\dot{x}(t) = Bu, \; u \in U = \{(u_1, u_2) \in \mathbb{R}^2: u_1 \geq 0, u_2 = 0\}$$
- The closed unit ball in $\mathbb{R}^2$, centered at the origin, $B_1(0)$.
- The initial state $x_0$ at time $t_0$: $x_0 = (0, 1)$.

Is there any control $u \in U$ that is able to drive the state of the system to the interior of $B_1(0)$ when departing from $x_0$?

The differential equations that guide the behavior of the system are:
\[
\dot{x}(t) = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} = Bu = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}
\]

Since \(u_2 = 0\) and \(u_1 > 0\), we get:

\[
\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} = \begin{bmatrix} u_1 \\ 0 \end{bmatrix}, u_1 > 0
\]

Thus, if we draw the set of possible velocities from point \(x_0\), we get the following graph:

As we can see, there are no velocities with negative vertical components in order to drive the state of the system to the interior of \(B_1(0)\). So, the answer is no.

Consider now the control constraint to be

\[
U = \{ (u_1, u_2) \in \mathbb{R}^2 : u_1 \geq 0, u_2 \in \mathbb{R} \}
\]

Now, it is possible to reach the interior of \(B_1(0)\) if \(u_2 < 0\). The control \(u \in U\) should be:

\[
U = \{ (u_1, u_2) \in \mathbb{R}^2 : u_1 \geq 0, u_2 < 0 \}
\]
Example 2 Going a little bit further with this geometric interpretation, let us find conditions for $\dot{x}(t)$ at a point $x$ at the boundary of a closed set $S$ which ensure that the trajectory of the system penetrates $S$.

\[ \text{For } x \in \text{bdry}(S), \exists \dot{x} \in f(t,x,u), u \in U : \tilde{v}_N \cdot \dot{x} < 0 \]  

(7)

where $v_N$ is the normal outward pointing vector to the set at the point $x_0$ and $\text{bdry}(S)$ is the boundary of the set $S$. This means that if we wish to penetrate a given set $S$, there has to exist a velocity such that it points to the interior of $S$, i.e., the internal product between this velocity and the vector normal to the surface pointing outwards is negative.

The previous condition involves the existential quantifier. If it holds at every point of the boundary of a closed set $S$ then it is possible to find controllers which ensure that there exist trajectories of the system which will never leave $S$. For example, in our example the controller will take the form:

If $x \in S_{\text{int}}$, $U = \{ (u_1, u_2) \in \mathbb{R}^2 : \forall u_1, u_2 \}$

If $x \in S_{\text{boundary}}$, $U = \{ (u_1, u_2) \in \mathbb{R}^2 : \exists u, \tilde{v}_N < 0 \}$

Condition (7) concerns the existence of at least one penetrating trajectory. If we use the universal quantifier instead then we will get conditions which ensure that no trajectory will ever leave the set $S$ if they depart from a point inside the set.

\[ \forall \dot{x}, \tilde{v}_N \cdot \dot{x} < 0 \]

i.e., all velocity vectors have to point to the interior of the set.

3.5.2 Why are we interested in reach sets?

Reach sets of dynamic systems are pervasive in control. This is because the reach set describes the motion capabilities of a dynamic system. Consider a problem of control synthesis abstractly formulated as follows: given a dynamic system and a specification synthesize a controller so that the composition of system with the controller satisfies the specifications.

Consider a problem of verification abstractly formulated as follows: given a dynamic system and a controller check if the composition of the two satisfies a given property. Consider a dynamic system and a closed set $S$. Examples of the properties we are interested in are:
Invariance – the trajectories of the system do not leave S.
Attainability – the trajectories of the system will enter S.
Safety – the trajectories of the system do not enter S.

Checking for those properties amounts to solving reachability problems. There are synthesis techniques which produce controllers with guaranteed properties: this means it is not necessary to go through a verification phase to check if the system and the controller satisfy a given property. Arguments involving reachability concepts are also used to prove results in control theory.

The problem of computing the reach set of a general dynamic system is quite difficult. This is because the trajectories of a dynamic system can be quite complex. The problem of reach set computation is even more difficult for hybrid automata. A hybrid automaton consists of control locations with edges between the control locations. The control locations are the vertices in a graph. A location is labeled with a differential equation, and every edge is labeled with a guard (condition of jump), a jump (action to take when jumping) and a reset relation. The state of a hybrid automaton is a pair $l, x$ where $l$ is the control location and $x \in \mathbb{R}^n$ is the continuous state.

The general problem of reachability for hybrid systems is very difficult.

How to compute reach sets

The computation of reach sets is not a trivial matter because the reach set inherits the behavior of a dynamic system. See the behavior of some nonlinear systems. Several techniques for reachability analysis of hybrid systems have been proposed. They can be (roughly) classified in two kinds:

1. Purely symbolic methods based on (a) the existence of analytic solutions to the differential equations and (b) the representation of the state space in a decidable theory of the real numbers.

2. Methods that combine (a) numeric integration of the differential equations and (b) symbolic representations of approximations of state space typically using (unions of) polyhedra or ellipsoids.
In what follows we describe a technique to compute a special type of reach sets that is based on the existence of analytic solutions to the differential equations. This is because the special case we will deal with allows us to find a closed-form solution.

### 3.5.3 A construction for pursuit evasion games

Consider the motion of point $x \in \mathbb{R}$ described by the following equation with two adversary control inputs $(u, v)$ (Krasovskii, Subbotin, 1988):

$$\dot{x} = f(t, x, u, v), u \in P, v \in Q$$  \hspace{1cm} (8)

where the following hypotheses hold

H1) $f$ is continuous in all variables and $t \in T = (-\infty, \theta]$  
H2) for any bounded region $D$ in $\mathbb{R} \times \mathbb{R}$, $f$ satisfies the following Lipschitz condition:

$$\| f(t, x_1, u, v) - f(t, x_2, u, v) \| \leq \lambda(D) \| x_1 - x_2 \|$$

for any $(t, x_i) \in D(u, v) \in P \times Q$.

H3) for any $(t, x, u, v) \in T \times \mathbb{R} \times P \times Q$ the following inequality where $\sigma$ is a constant, is valid:

$$xf(x, t, u, v) \leq \sigma (1 + \| x \|^2)$$

H4) for any $(t, x) \in T \times \mathbb{R}$ and $s \in \mathbb{R}$ and the so-called ‘saddle point condition in a small game’ is valid:

$$\min_{u \in P} \max_{v \in Q} sf(x, t, u, v) = \max_{v \in Q} \min_{u \in P} sf(x, t, u, v)$$

Consider the set $M$:

$$M = \{(t, x) \in T \times \mathbb{R} : t = \theta, l_1 \leq x \leq l_2\}$$

The adversarial aspect of control is captured in an optimization problem where $u$ seeks to control $x$, the motion of the system, to enter $M$ at time $\theta$, i.e., $x(\theta) \in [l_1, l_2]$. The objective of $v$ is exactly the opposite. More formally, consider the following cost functional:

$$\gamma(x(t_0, x_0, U(\cdot), V(\cdot))) = \begin{cases} 
0 & \text{if } x(\cdot) \text{ intersects } M \\
1 & \text{otherwise}
\end{cases}$$

where $U(\cdot)$ is a control function for the first player and $V(\cdot)$ is a control function for the second player. Note that $\gamma(x(t_0, x_0, U(\cdot), V(\cdot))) = 0$ means that the trajectory $x(\cdot)$ departing from $(t_0, x_0)$ under controls $U(\cdot)$ and $V(\cdot)$ enters the target set $M$ at time $\theta$.

The adversarial aspect of control is captured in an optimization problem where $u$ seeks to minimize $\gamma$ and $v$ seeks to maximize it. Under the hypotheses (H1-4) this game has a value, or equivalently:
Multi-UAV command and control system

\[
\inf_{\mathcal{U}} \sup_{\mathcal{V}} \gamma(x(t_0, x_0, U, V)) = \sup_{\mathcal{V}} \inf_{\mathcal{U}} \gamma(x(t_0, x_0, U, V))
\]  \hspace{1cm} (9)

Moreover, the game has a saddle point, i.e., there exist strategies \(U^*, V^*\) such that:

\[
\gamma(x(t_0, x_0, U^*, V^*)) \leq \gamma(x(t_0, x_0, U^*, V^*)) \leq \gamma(x(t_0, x_0, U^*, V^*))
\]  \hspace{1cm} (10)

This is summarized in the following theorem.

**Theorem 1** For any closed set \(M\) and for any initial position \((t_0, x_0)\), one and only one of the following assertions is valid:

- The value of the game is 0. Any \((U^*, V^*)\) is a saddle point \((V^*\text{ is any feedback strategy})\).
- The value of the game is 1.

This is basically the theorem on an ‘alternative’ from (Krasovskii, Subbotin, 1988). Moreover, the optimal strategies \(U^*\) and \(V^*\) are feedback strategies. This is an important result, because it states that we cannot obtain improvements in the performance if we consider additional information (other than the state information) in the optimal strategy.

The notion of \(u(v)\)-stable bridge is a very important one in this setting. Informally, a \(u(v)\)-stable bridge \(W_0(W)\) is the set of all points \((t_0, x_0)\) such that there exists an optimal strategy \(U^*(V^*)\) that keeps the motion of the system departing from \((t_0, x_0)\) inside (outside) \(W_0(W)\) until \(M\) is reached (avoided). Notice now the relation to reach sets. We want to find the set of all states such that there is a control strategy which will ensure that the trajectories of the system will attain the target set no matter what the adversary does. So this is a problem of backward reach set computation under disturbances.

Now consider this problem setup for one-dimensional dynamics, i.e., \(x \in \mathbb{R}\). In this problem setup it is possible to derive a closed form for the \(u\)-stable bridge. First consider the following expressions:

\[
f_1(t, x) := \max_{u \in \mathcal{P}} \min_{v \in \mathcal{Q}} f(t, x, u, v) \tag{11}
\]

\[
f_2(t, x) := \min_{u \in \mathcal{P}} \max_{v \in \mathcal{Q}} f(t, x, u, v) \tag{12}
\]

Now consider solutions \(w_1\) and \(w_2\) to the following differential equations:

\[
w_1'(t) = f_1(t, w_1(t)), w_1(\theta) = l_1 \tag{13}
\]

\[
w_2'(t) = f_1(t, w_2(t)), w_2(\theta) = l_2 \tag{14}
\]

The \(u\)-stable bridge is the set:

\[
W_0 := \{(t, x) \in T \times \mathbb{R} : t \in T, x \in [w_1(t), w_2(t)]\}
\]

where
\[ T_* = [\tau_*, \theta] \]

Here, we are integrating backwards. Thus, since \( w_2 \) is above \( w_1 \) at the beginning, if they intersect then it means that from that point backwards the stable bridge is empty.

This construction has a simple and appealing geometric interpretation. Keep in mind that the state evolves in \( \mathcal{R} \). Equations (13, 14) describe the evolution (in reverse time) of the boundaries \((l_1, l_2)\) of the target set \( M \) when both players adopt optimal control strategies (given by the \( \text{argmax}^3 \) and \( \text{argmin}^4 \) in equations (11,12)). Now, consider an initial state \((t,x)\) in the relative interior of \( W_0 \) (if the interior is empty this means that the initial state is at the boundary of \( W_0 \)). Then, apply any control strategy \( U \) until the state reaches the boundary of \( W_0 \). From this point onwards apply the optimal control strategies to both players. Then, by construction of \( w_1 \) and \( w_2 \) the state slides along one of the boundaries of \( W_0 \) (\( w_1 \) or \( w_2 \)) until it reaches \( l_1 \) or \( l_2 \), respectively, at time \( \tau \).

Suppose the following problem. Two players are evolving in a 1-dimensional setting. Player_1 is trying to get within a range \( r^* \) of Player_2 at time \( \tau \) (set to 0). Player_2 is oblivious to Player_1.

The relative velocity between the Players depends only on the control strategies of both Players. Thus, the dynamics of this system is linear:

\[
\frac{dx}{dt}(t) = r_1(t)u(t) + r_2(t)v(t),
\]

\[ |u(t)| \leq \beta_1, |v(t)| \leq \beta_2, T = [0, \theta] \] (15)

The set we wish to reach is given by: \( M = \{(t,x): t = \theta, |x| \leq r^*\} \) (fig. 22). Therefore, \( u \) will win if \( x \) reaches \( M \), while \( v \) wins if \( x \) does not reach \( M \). This generates the following problem:

- What is the set \( W \) of positions \((t,x)\) such that there is a winning strategy for \( u \) (safe set for \( u \))?  

---

3 argument with which the function reaches its maximum  
4 argument that makes the function reach its minimum
Multi-UAV command and control system

![Diagram](image)

**Figure 22: the set M to reach**

If we imagine one player in the upper bound of M and that each player does its best to win the game, when we integrate equation (15) backwards in time we get:

$$\rho(t) = r + \int_t^\theta (b_1|r_1(\tau)| - b_2|r_2(\tau)|)\,d\tau$$

Here, we are assuming symmetry in dx/dt, as well as a simplicity by considering $|r_1(t)| = r_1$ and $|r_2(t)| = r_2$. The set of states (W) from which $v$ looses the game ($u$ wins) is:

$$W = \{(t,x) : t \in T^*, |x| \leq \rho(t)\} \text{ where}$$

- $T^* = T$ if $\rho(t) \geq 0$, for all $t$ in $T$
- $T^* = [t^*, \theta]$ where $t^* = \sup \{t \in T : \rho(t) < 0\}$

![Diagram](image)

**Figure 23: Backward reach set**

The backward reach set for this problem is shown in figure 22.
This construction can be extended to a Multiple-Input-Multiple-Output (MIMO) problem when the target set is a closed ball centered at the origin since this involves working with a norm (1-dim function).
4. Apollo’s proposed Architecture

4.1 Hierarchical control architecture

When we introduced the hybrid automata theory, we also discussed some issues about the architecture for a system of this type. It should be a hierarchical system, where higher layers would take discrete-event decisions while the lower layers would be responsible for the continuous control of the vehicles. This architecture provides a modular and easily expansible system allowing for an incremental development. Hence, we can always add new modules for unpredicted situations that may arise.

Before detailing the architecture, some specification concepts are clarified next:

**Manoeuvre**: a prototype of an action/motion description for a single UCAV and the atomic component of all specification concepts.

**Mission**: an array of manoeuvres to be executed sequentially. Other mission structures are possible, but are not implemented.

**Task**: a prototype of an action/motion description for a group of vehicles (team).

To each type of specification concept there corresponds one type of controller. There are two types of controllers: vehicle controllers and team controllers. Vehicle controllers control mission and manoeuvre execution. Team controllers control task execution.

Figure 24 shows the architecture we wish to develop. It consists of five layers, where the lowest layer is the vehicle to control, while the highest is a team controller for coordinated missions.
We will now detail the several levels of this structure:

**Platform** is the physical vehicle to receive commands and send sensor or telemetry data. **Manoeuvre controller** supervises the execution of a vehicle manoeuvre. It sends commands to the Platform and gets the current status from it. It accepts abort and configuration commands from the vehicle supervisor and sends status messages to it. **Vehicle supervisor** supervises all of the UAV operations. It receives manoeuvre specifications through a link to either the Mission Supervisor during a mission execution or to a team controller during a team task execution. Launches the corresponding manoeuvre controller, monitors its execution and accepts configuration commands from an external controller. For example, it should be possible to change the link to a team controller. This means that it can be possible to move the vehicle among teams. The vehicle supervisor is the same throughout the life span of the UCAV. If there is no link to a team controller and no mission to be executed the vehicle supervisor commands the execution of a default manoeuvre (in our case, loiter manoeuvre).

**Mission Supervisor** supervises the execution of a mission. **Team controller** supervises the execution of a task. It commands and monitors the execution of vehicle manoeuvres to execute the task specification. It does this by exchanging messages with the vehicle supervisors in the team. It also provides for a definition of a team as a set of UCAVs under the control of a team controller. The team controller also accepts configuration and task execution and abort commands.
Each layer can be modelled through a hybrid automaton with continuous and discrete states, organizing the system in a totally modular way.

Combining the proposed architecture with the capabilities of the Piccolo autopilot, we can build a system where the vehicles can be real and/or simulated, due to capability of the ground station to communicate with multiple vehicles. Thus, even if we have only one aircraft, we can always test our coordinated control algorithms using simulated vehicles (figure 25).

![Figure 25: Real/Simulated vehicles](image)

With the proposed architecture, we can have two different hardware structures. On one hand, the controllers can be running on a ground computer and the commands are continuously sent to the avionics (figure 25). On the other hand, the controllers can be running onboard the vehicle without overloading the communications link. Thus, while Piccolo manages low-level flight control, this computer (PC-104 in this example) manages the execution of a larger number of higher-level tasks, such as vision processing, trajectory planning, communicating between vehicles, etc. Additionally, the onboard computer may have an independent radio link to transmit this high level information. This second structure is depicted in figure 26.

![Figure 26: Hardware structure with extra onboard processing](image)
4.2 Expansibility and Modularity

The proposed system architecture has the property of being easily expanded in order to be used with multiple UAVs to carry out coordinated missions. Moreover, this architecture provides for a modular system, which is one of the most important features when dealing with dynamical systems of multiple vehicles. This allows us to build a system composed of different types of entities, such as consoles to monitor and control vehicles, physical or simulated vehicles or even cameras or other type of hardware, in a transparent way to the network. Consequently, this system allows us to build an environment where different kinds of vehicles are present (AUVs, ASVs, UAVs), which is precisely what we desire.

All that was said can be seen in fig. 27, where all communications between vehicles and their control consoles are done through Seaware.
4.3 Apollo

Fig. 28 is a representation of the Apollo system where an emphasis is given on messages exchanged through middleware. Apollo is responsible for translating Piccolo’s received data into messages to publish (send) in Seaware as well as to translate commands subscribed (received) from Seaware (such as altitude, waypoint, joystick or mission commands) into data structures that the autopilot understands.

![Diagram of Apollo system]

Figure 28: Piccolo – Seaware - Neptus schematic

Hence, Apollo should create a MiddlewareNode with a suitable topic domain to carry out the communications over the middleware (Seaware). This node publishes in the middleware the aircraft state (EstimatedState, Motor), Manoeuvre status or error signals (Errors). It subscribes high/low level Commands, Joystick movements, Mission plans or Abort signals. This exchanged information is done through messages identified by a topic (in bold). More details on the implementation of the Send Data and Receive Data methods are given in the “Development” chapter.

Neptus, on some other place of the network, creates a MiddlewareNode with the same topic domain to execute the opposite sequence of steps of the interfacing node. EstimatedState, Errors or Manoeuvre messages are subscribed while Commands, Joystick, Mission or Abort messages are published on the network.
The HWIL preparation with Neptus is shown in figure 29. The Neptus console connects to the GS through the operator interface computer which is running as a server. However, we should not think that this is a rigid structure. Neptus can run on the same computer as the OI or on a separate one. The same applies to the interface, so that every combination is possible.
5. Apollo’s development

As already mentioned before, Apollo is the name of the mixed-initiative command and control system developed and its development process is described next. The Apollo system is based in dynamic networks of hybrid automata that represent the modules of supervision and manoeuvre and include the several details of operator interaction. Each module is described as a hybrid automaton that interacts with the hybrid system network. We also explain the system implementation software based on an object oriented programming. Moreover, some error detection issues are discussed together with reach set theory applications.

5.1 Hybrid System

In order to devise the interface with the Piccolo autopilot, it was necessary to go through a study phase of both its communications protocol and simulation environment. The communications System Development Kit (SDK) provided by CloudCap is free for download at the CloudCap website (see References), and assists in starting a customized communications interface.

With the hybrid system’s concepts in mind, we tried to develop a system that, besides following the expansibility and modularity already explained, would also follow the hybrid system formalization with the hierarchical control structure previously described.

The Apollo system architecture proposed in section 4.1 was developed until the mission supervision level. Thus, we obtain a composition of synchronized automata in exchanged commands and events like the one presented in figure 30.

The figure shows the several hybrid automata that compose Apollo and how they are synchronized with each other. The transitionStructure is also represented.

The system starts in the Mission Supervisor. It is at this level that Apollo interfaces with an operator through Neptus. This automaton will read the mission specification file and start the execution of the plan. The outputs to the vehicle supervisor are the name of the manoeuvre together with the specification that defines it. The inputs from the vehicle supervisor is an acknowledgement signal, saying that the manoeuvre is currently executing or an end signal saying the manoeuvre has finished its execution. This automaton has also the ability to send cancel commands, which terminate the current manoeuvre and plan.
The vehicle supervisor is responsible for executing the orders coming from the Mission Supervisor. It launches and terminates manoeuvres and monitors their execution.

goTo, loiter and teleoperation and the Manoeuvre Controllers implemented. They accept launch commands and keep executing until the manoeuvre objective is achieved, an error or a timeout have occurred, or a terminate command is sent.

In order fully characterize the Apollo system, we also need to define its inputs and outputs to external applications. These can be found below:

**Inputs:**
- Messages from Seaware: Commands, Plans, Joystick
- Packets from Piccolo: Telemetry, Control

**Outputs:**
- Messages to Seaware: EstimatedState, Motor, ManoeuvreState, ManoeuvreErrors
- Packets to Piccolo: Commands

For more details about the tasks involved in each of these levels, see next section.
5.2 Hybrid Controllers

5.2.1 Mission Supervisor

The hybrid automaton that describes this supervisor can be seen in figure 31. This automaton starts at the Init state, where it reads the mission specifications file. If this file is present, it will create a vector of manoeuvre specifications. In each position we can find all the parameters that define a specific manoeuvre. On the exec state, we give manoeuvre commands and wait for their finish until no manoeuvres are left. If no plan is available, we send to the vehicle supervisor the default manoeuvre specification and wait for a new plan to arrive.

![Diagram of Mission Supervisor](image)

Figure 31: Mission Supervisor

We have presented the transition structure of this automaton. However, to complete its formalization we need to specify its states, inputs, outputs and initialState as shown below:

<table>
<thead>
<tr>
<th>States:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
</tr>
<tr>
<td>Exec</td>
</tr>
<tr>
<td>Idle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inputs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neptus Commands</td>
</tr>
<tr>
<td>(DONE/TERMINATED)</td>
</tr>
<tr>
<td>Mission Plans END</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal(done)</td>
</tr>
<tr>
<td>mnvr specification</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>InitialState:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
</tr>
</tbody>
</table>
5.2.2 Vehicle Supervisor

The vehicle supervisor (figure 32) executes the commands sent by the mission supervisor. Thus, when it receives a new manoeuvre command and its specification, it jumps to the exec state which will launch and monitor the new manoeuvre. The automaton leaves this state to Idle when the manoeuvre has finished or when a terminate command is received. Here, if the waiting time for new orders reaches a timeout (0.5 seconds), it will launch the default manoeuvre to guarantee that the UAV is under our control. If no orders are found, the automaton launches the default manoeuvre with a default specification. We are not expecting this to happen very often, because the mission supervisor itself assigns the default manoeuvre in case no plan is available.

![Diagram: Vehicle Supervisor]

Figure 32: Vehicle Supervisor

To complete the formalization of this automaton, we need to specify its states, inputs, outputs and initial state:

<table>
<thead>
<tr>
<th>States</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Initial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>action</td>
<td>configure(mnvr, specification)</td>
<td>Init</td>
</tr>
<tr>
<td>Exec</td>
<td>mnvr</td>
<td>execute(mnvr)</td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>specification</td>
<td>end(mnvr)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MANOEUVRE_END,</td>
<td>signal(END_MNVR)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TERMINATEcommand</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

July 2006 53-84
5.2.3 Manoeuvre Controllers

In what follows we present a generic manoeuvre controller (figure 33). We don’t go into the details of how each manoeuvre is implemented due to their similarities. Thus, the only difference to be considered is in the controller responsible for executing the manoeuvre.

When these automata start, they try to create the controller. If there was an error on this operation, an error signal is sent and the manoeuvre ends. If no error was detected, the controller executes, by sending commands to Piccolo and the current manoeuvre state to Neptus. When the manoeuvre finishes due to completion of its objective or due to external termination order, it sends to the vehicle supervisor a DONE signal and then ends.

![Manoeuvre controller](image)

Figure 33: Manoeuvre controller

The complete formalization of these automata can be found next:

**States:**
- Initialization
- Execute controller
- Error
- Done

**Inputs:**
- configure(specification)
- EXECUTE
- TERMINATEcommand

**Outputs:**
- send_Piccolo(commands)
- send_Neptus(state)
- send_error(code)
- send_vehicleSupervisor(DONE)

**InitialState:**
- Initialization

The manoeuvre controller mentioned before, obeys to the following block diagram representation:
Thus, the actuator commands that this system will output to Piccolo are in the form of high level commands that the Piccolo controllers understand (Turn rate, waypoint, Altitude or Speed). Currently, our manoeuvres are only using waypoint commands. However, in case we implement a *track_path* manoeuvre, the best way to do it is to use turn rate commands.
5.3 Object Oriented implementation

The hybrid structure previously described was implemented using object oriented programming. The basic structure of the program can be seen in the diagram below and it consists of creating four executing threads to perform the following tasks:

- Message exchange in Seaware
- Packets reception from Piccolo (telemetry, attitude, GPS, etc)
- Vehicle supervisor
- Mission supervisor

![Diagram](attachment:image.png)

Figure 35: System implementation

To implement the manoeuvres, we defined a top-level abstract class (*manoeuvre*) containing most of the member-functions and member-objects that the manoeuvres will need. Thus, any new manoeuvre will inherit the structure of the *manoeuvre* class. Thus, in case we wish to create new manoeuvres we only need to devise the manoeuvre controller and write this code in the *CodeExecute* function. This inheritance structure is depicted in the UML diagram of figure 36.
Figure 36: manoeuvre classes declaration
5.4 Error detection

As can be seen from the automata presented before, an error tackling strategy is not implemented yet. However, the program is already capable of doing error detection. It can detect errors on manoeuvre initialization and execution.

An interesting feature is the capability of masking errors. There can be situations when the operator does not want the system to detect certain types of error. Then he/she can just send a command to mask that error. For instance, suppose the operator does not wish that manoeuvres terminate due to timeout and wants them to continuously keep trying to achieve their goal. It is just sets this variable to zero. To disable the number_of_tries detection, just set enableTries to false.

It follows the list of errors that can be detected in each implemented manoeuvre:

<table>
<thead>
<tr>
<th>Manoeuvre</th>
<th>Mask to enable error</th>
<th>Errors detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>teleoperation</td>
<td>enableInit=true</td>
<td>Error in initialization</td>
</tr>
<tr>
<td></td>
<td>enableError=true</td>
<td>Vehicle not obeying manual commands</td>
</tr>
<tr>
<td>goto</td>
<td>enableInit=true</td>
<td>Error in initialization</td>
</tr>
<tr>
<td></td>
<td>Timeout!=0</td>
<td>Vehicle is taking too long to reach waypoint</td>
</tr>
<tr>
<td></td>
<td>enableTries=true</td>
<td>Number_of_tries to reach waypoint has been exceeded</td>
</tr>
<tr>
<td>loiter</td>
<td>enableInit=true</td>
<td>Error in initialization</td>
</tr>
<tr>
<td></td>
<td>Timeout!=0</td>
<td>Vehicle is taking too long to reach circle</td>
</tr>
</tbody>
</table>
5.5 Guaranteed manoeuvres

The reach set theory presented before fits perfectly into the capabilities of the Apollo system. For instance, based on those theoretical results it is possible to execute manoeuvres with guaranteed success. There are some different approaches to reach sets usage on Apollo:

1. the mission supervisor verifies if the current state is part of the backward reach set and takes the decision of executing or not. This way, we can guarantee the assignment of reachable points only;
2. the verification of belonging to the backward reach set can be done inside the manoeuvre (with a mask variable, as we did with errors) which then communicates it to the supervisors;
3. we can create special kind of manoeuvres with this property.

The property of guaranteed success is especially important when time is a critical factor. For instance, let us consider the example depicted in figure 37. Two UAVs are evolving in a 2-dimensional setting (flying at the same altitude). UAV$_1$ is trying to rendezvous with (get within a range $r^*$ of) UAV$_2$ at time $\tau$ (set to 0). UAV$_2$ is oblivious to UAV$_1$ (UAV$_2$ might be performing some task, or actively trying to stay away from UAV$_1$).

![Manoeuvre scenario](image)

**Figure 37: Manoeuvre scenario**

We have adopted the standard practice in control design and used simplified equations of motion. The equations of motion for each UAV are taken to be as follows:

$$m_\alpha \dot{\alpha} = -\alpha_\alpha \dot{\alpha} + u$$

(16)
where \( m_i \) indicates the mass of \( \text{UAV}_i \) and \( a_i \) is a friction coefficient for \( \text{UAV}_i \). The terms \( u \) and \( v \) will be used to denote the control terms for \( \text{UAV}_1 \) and \( \text{UAV}_2 \) respectively. The control terms are subject to the constraints:

\[
\| u \| \leq \beta_1, \quad \| v \| \leq \beta_2
\]

where \( \beta_1 \) and \( \beta_2 \) are given. In state-space form, if we denote the position of \( \text{UAV}_1 \) by \( (x_1, x_2) \),

\[
\begin{align*}
\dot{x}_1 &= x_3 \\
\dot{x}_3 &= (-\alpha_1 x_3 + u_1)/m_1 \\
\dot{x}_2 &= x_4 \\
\dot{x}_4 &= (-\alpha_1 x_4 + u_2)/m_1
\end{align*}
\]

The equations for \( \text{UAV}_2 \) are identical if the following substitutions are enacted: \( m_1 \) and \( a_1 \) are replaced by \( m_2 \) and \( a_2 \), \( u_1 \) and \( u_2 \) are replaced by \( v_1 \) and \( v_2 \), and the states are numbered from 5 to 8 instead of 1 through 4. The target set for \( \text{UAV}_1 \) is given by:

\[
\mathcal{M} = \{ (\theta, x) = (0, x) : x \in \mathbb{R}^8, (x_1 - x_5)^2 + (x_2 - x_6)^2 \leq r^2 \}
\]

\( \text{UAV}_1 \) and \( \text{UAV}_2 \) evolve in the set:

\[
\mathcal{N} = (-\infty, 0] \times \mathbb{R}^8
\]

Note that we are only interested in the relative motions \( (x_1 - x_5) \) and \( (x_2 - x_6) \). So, if we write the equation \( x = x_{\text{UAV}_1} - x_{\text{UAV}_2} \) then the equations describing these relative motions can be put in the following form:

\[
\dot{x}(t) = Ax(t) + Bu(t) + Cv(t)
\]

We now make a transformation of variables by using the formula of variation of constants from (Krasovskii, Subbotin, 1988) to obtain an equivalent system of the form:

\[
\dot{y}(t) = B_u u_* (t) + C_v v_* (t)
\]

where the control \( u_* \) and \( v_* \) are equal to \( u \) and \( v \) for almost all \( t \in [0, T] \). Using this transformation, one will not have to consider the state in the maximization process described above for calculation of the u-stable bridge. The transformation is applied using:

\[
y(t) = F(t).x(t)
\]

where
\( F(t) = F \Phi_\ast(t) \)

and \( F \) is a matrix of constants given by (Krasovskii, Subbotin, 1988): \( F = (I, O, -I, O) \)

where \( I \) and \( O \) are the 2-by-2 identity and zero matrices, respectively.

\[ \Phi_\ast(t)(i = 1, 2) \text{ is given by:} \begin{bmatrix} 1 & 0 & \alpha_i(t) & 0 \\ 0 & 1 & 0 & \alpha_i(t) \\ 0 & 0 & \nu_i(t) & 0 \\ 0 & 0 & 0 & \nu_i(t) \end{bmatrix} \text{ with} \]

\[ \phi_i(t) = \left[ 1 - e^{-\frac{\alpha_i}{m_i}t} \right] \frac{m_i}{\alpha_i} \]

\[ \psi_i(t) = e^{-\frac{\alpha_i}{m_i}t} \]

Let us set \( m_1 = m_2 = 1 \) to simplify the notation. In short, the transformation yields:

\[ B_\ast(t) = r_1(t)I, C_\ast(t) = r_2(t)I \]

with

\[ r_1(t) = \frac{1 - e^{\alpha_1 t}}{\alpha_1} \]

\[ r_2(t) = -\frac{1 - e^{\alpha_2 t}}{\alpha_2} \]

That is,

\[ \dot{y}(t) = r_1(t)u(t) + r_2(t)v(t) \] (21)

and, remember, the controls are still subject to the constraints 17. We transform the sets \( M \) and \( N \) as well, which yields:

\[ M_* = \{ (0, y) : y \in \mathbb{R}^2, \| y \| \leq r_* \} \]

and

\[ N_* = (-\infty, 0] \times \mathbb{R}^2 \]

Note that after this simple transformation of variables we will work with modulus of \( y \) (\( \| y \| \)) and the problem can be cast as a one-dimensional problem. This means that we can use the construction presented above.

**Construction of the u-stable bridge**

If \( W_0 \) is the maximum u-stable bridge, the problem of attaining the target set for the first player is solvable if \((t_0, x_0) \in W_0\). Here, the maximal u-stable bridge is given by:
\[ W_0 = \{(t, y) \in \mathbb{T} \times \mathbb{R}^2, \|y\| \leq \rho_0(t)\} \] (22)

And \(\rho_0\) is given by the equation (an extension to \(\mathbb{R}^2\) of the result from equation 13):

\[ \rho_0(t) = r^* + \int_0^t (\beta_1 |r_1(\tau)| + \beta_2 |r_2(\tau)|)d\tau \] (23)

This equation comes from a direct backwards integration of 21, having in mind the final state is \(x(0) = r^*\). Moreover, we also did a maximization process from equation 21 by assuming that each player will do its best. Thus, we replace \(u(t)\) and \(v(t)\) by their maximum control, \(\beta_1\) and \(\beta_2\) respectively. Note that \(\rho_0(t)\) is the diameter of the \(W_0\) at time \(t\). Plugging in for the expressions of \(r_1\) and \(r_2\), and noticing that in our case, \(t\) and \(r_2(t)\) are negative or zero and \(r_1(t)\) is positive or zero, we get:

\[ \rho_0(t) = r^* - \frac{\beta_1/\alpha_1 - \beta_2/\alpha_2}{(1 - e^{\alpha_1 t}) + (\beta_2/(\alpha_2)^2)(1 - e^{\alpha_2 t})} \]

The figure below illustrates several profiles of \(\rho_0\) as a function of the parameters. We set \(r^* = 0\), \(\beta_2 = 24\), and the friction coefficients to be \(\alpha_1 = 3.5\) and \(\alpha_2 = 1\) (the plane that is trying to rendezvous has more friction). We plot the \(\rho(t)\) for different values of \(\beta_1 = 24, 48, 96\). For the least amount of control input, \(\beta_1 = 24\), it will not be possible to guarantee success of the one-sided rendezvous mission (\(\rho_0\) is always negative). For \(\beta_1 = 48\), there is a small set of initial conditions such that the norm of \(y(t_0, x_0)\) is under the curve but above the \(y\)-axis, for which it will be possible to guarantee the success of the mission using the control strategy given below. For \(\beta_1 = 96\), there is a much larger space in which initial conditions will guarantee success of the mission.

![Maximal u-Stable Bridge Construction](image)

**Figure 38: The safe set**

July 2006

62-84
Optimal Strategy

The optimal strategy for the u controller is expressed in terms of y(t,x).

\[
U^o(t,x) = \begin{cases} 
- \beta \frac{y(t,x)}{\| y(t,x) \|} & \text{for } \| y(t,x) \| \neq 0 \\
0 & \text{for } \| y(t,x) \| = 0
\end{cases} \tag{24}
\]

where the components of y(t, x) are obtained using transformation 24:

\[
y_1(t,x) := x_1 - x_5 + x_3(1 - e^{\alpha_1 t})/\alpha_1 - x_7(1 - e^{\alpha_2 t})/\alpha_2 \\
y_2(t,x) := x_2 - x_6 + x_4(1 - e^{\alpha_1 t})/\alpha_1 - x_8(1 - e^{\alpha_2 t})/\alpha_2
\]

This is a powerful result. If we have

\[
\| y(t_0, x_0) \| \leq \rho_0(t_0)
\]

where t0 and x0 designate the initial time and position, and where

\[
\rho_0(t) \geq 0, \forall t_0 \leq t \leq \theta = 0
\]

then using the optimal strategy GUARANTEES that

\[
(x_1 - x_5)^2 + (x_2 - x_6)^2 \leq r^* \tag{25}
\]

If we do NOT have

\[
\| y(t_0, x_0) \| \leq \rho_0(t_0)
\]

then there is no feedback strategy that will guarantee success for the game.

The conclusion from this example is that in order to reach the first UAV in a certain amount of time, we have to derive the best control strategy for it. From the equations of motion (16) we should notice that the optimal control obtained is in the form of a force (N). However, Piccolo only accepts velocity commands. Therefore, we should now find the transfer function from force to velocity and our controller is ready to be integrated with Piccolo.
Multi-UAV command and control system

There are also situations where a second UAV is not present, but just a waypoint. Let’s suppose the situation depicted in figure 39 where the wind velocity is $\vec{v}_{\text{wind}}$.

![Figure 39: UAV to reach WP with wind](image)

This can be transposed to a situation where the waypoint to reach acts as the second UAV. We assume no wind velocity and a waypoint with velocity: $\vec{v}_{\text{WP}_1} = -\vec{v}_{\text{wind}}$ (figure 40).

![Figure 40: UAV to reach moving WP with no wind](image)

Thus, even in situations with only one UAV we can use the reach set theory to determine the best control strategy to reach the target in some specified time $\tau$. 

July 2006
6. Apollo tests and results

Since we are using Piccolo’s development system we were able to test Apollo by both SWIL and HWIL simulation. As expected, the behaviour of the system was the same. This allows us to assert that the performance of the SWIL simulation is exactly as fast as it will be the real system. The results from simulation are perfect in terms of delays when we go from SWIL simulation to experimental flight. However, we have to take into account that we are only using simulated environments which present ideal conditions for the systems and unexpected events will surely happen.

The HWIL setup is shown in figure 41. Through this image it is clear the usage of the three computers. While one runs the simulator and flightGear, another runs the OI while the third runs the Neptus console. Neptus reads data through Seaware while Apollo connects to both Seaware and the GS. It is the OI computer that makes the interface between Apollo and the GS. The GS is wirelessly linked to a Piccolo through the 2.4GHz frequency.
The system Neptus-Apollo-Piccolo can be seen in figure 42. This screenshot shows the UAV Console (top left) the Apollo system (right) and the OI (bottom right). On the UAV console, we see a 3D image of the aircraft as well as a 2D map of the hovering area. The small window besides the OI is the joystick window for us to visualize the joystick position.

Figure 42: UAV Console, Piccolo OI

For a Guided Tour on how to run the Piccolo system together with instructions on how to run the interface see appendix B.
7. Conclusions and future work

UAVs have been deserving an ever growing importance in both civil and military operational environments as well as an increased complexity of cutting-edge technology. Existing UAVs are high density technology systems, with a heavy automated behaviour able to perform a wide variety of missions.

The developed Apollo system consists of a mixed-initiative command and control system (human operator in the control loops) for Unmanned Air Vehicles (UAVs). It was conceived having in mind its integration with Piccolo and also with the Neptus framework. Its requirements and subsystems development have been carefully planned and the results were shown. A meticulous study of hybrid systems in order to model this system based in dynamic networks of hybrid automata has proven extremely useful. This way, the Apollo system is modular, hierarchical and easily expandable encapsulating the operational logic in modules of supervision and control. Nevertheless, a limitation of this system is its one-to-one operation, i. e., it can only communicate between one UAV and one Seaware Node.

Considering the current work status, we can say that we are ready for an experimental flight test. The airframes have been built and RC tested and the software has gone through all SWIL and HWIL tests. Thus, the Apollo system is ready for real test. Moreover, the Piccolo autopilot is going to exist very soon. However, to prepare for these flight tests, it is still necessary to make the simulation model for the all airframes, mount Piccolo autopilot in each of them and make all the necessary instrumentation. Only after this we are ready to integrate all systems and carry out an autonomous flight demonstration.

Future work

Future work extends along several directions. For instance, there are still some aspects of the Apollo system that should be developed or improved. The work to be done consists of:

- solving the one-to-one limitation by automatically detect several UAVs and creating a Seaware node for each of them. Remember that Apollo is an application that can have any number of MiddlewareNodes.
Multi-UAV command and control system

- Developing more complex manoeuvres and controllers, such as *track_person*, *track_vehicle*, *search*, *patrol*, *follow_path*, *sense_area*, *coordinated_sensing*, etc.
- Make error detection and handling with exceptions in C++. These are a powerful C++ capability that should be exploited to the fullest in order to simplify code.
- Improve communications with Neptus. Enlarge the messages scope for UAVs
- Write all hybrid automata systems as classes to improve system modularity. Currently, *Mission supervisor* and *vehicle supervisor* are written as threads.
- Improve the capabilities of the Mission supervisor to perform the following functions:
  - Store more than one plan file
  - Check reachability (if enabled by operator)
  - Change current mission
  - Change manoeuvres in current mission
  - Abort current plan by sending single manoeuvre command
- Make extensive unit tests for the classes developed
- Develop the Apollo system for Windows. It currently works only under Linux due to library conflicts in Windows. However, since we using generic code, there should be no problem porting the system from one OS to another, except for minor details.

Following the incremental development cycle, future steps for the AsasF project in order to improve the system performance are already identified. One is related with altitude measurement. Even though on takeoff and normal flight, an altitude error should be tolerable, on the landing manoeuvre it is a critical factor, since a little imprecision can cause an aircraft loss. Since Piccolo’s altitude measurements can have significant errors (pressure sensors and GPS), the use of a sonic altimeter connected to the autopilot on the landing manoeuvre, in order to achieve better results is going to be tested. Other improvements to the system are listed next:

- Increase of autonomy for at least two hours
- Able to perform day or night missions
- Autonomous takeoff and landing
- Emergency signal for UAV localization in case of crash
- Onboard computer system for data storage and redundancy
- Gimbal video camera (which has the property of rotating in the three axis)
- Infrared camera (for fire detection)
Multi-UAV command and control system

- Image processing in the onboard computer (does not overload the communication link) or on a ground computer (less onboard payload weight)
- *FailSafe* system in case of Piccolo failure (e.g., due to low batteries)
- System to keep the sparkplug hot in order to avoid unexpected engine shut down.

It should be mentioned that together with the developed work one paper was publish in Encontro Científico de Robótica, 2006 (Almeida *et al*, 2006) a one other has been submitted to the 1st First IFAC Workshop on Multi-vehicle Systems.
Aknowledgements

This project was supported by the POCI 2010 scholarship which promotes the following:

- Increase cooperation between universities and companies from several areas of economic activity, as a way for a better interaction between the students’ skills and the demands of the market;
- To ease the insertion in the job market of graduated people
- To have a learning experience in the context of the job market

This work was performed at the Instituto de Sistemas e Robótica - Porto between February, 1st, 2006 and July, 31st, 2006. Gil Manuel Gonçalves was the advisor from the Institute of Systems and Robotics (ISR-Porto) and João Borges de Sousa was the supervisor from FEUP.

Thus, I am especially indebted to both my advisors, Gil Manuel Gonçalves and João Borges de Sousa, for their valuable guidance, patience and encouragement throughout my research, as well as for their financial support.

I would like to thank Ricardo Bencatel from Mechanical Engineering who worked with me during this period and helped to attain a good relationship.

Special thanks to NAAM people for their time and insights shared with the AsasF project.

I would also like to acknowledge the support given by FEUP, as well as by the Underwater Systems and Technology Laboratory (USTL) and Institute of Systems and Robotics (ISR).

The PESC program (Projectar, Empreender, Saber Concretizar) also deserves an acknowledgement for the financial support.

Finally, I would like to give a special thanks to Professor António Torres Marques and Professor Fernando Lobo Pereira for their contribution to the AsasF project.
References

Almeida, Pedro; Benacatel, Ricardo; Gonçalves, Gil M.; Sousa, João B.; Multi-UAV Integration for Coordinated Missions, Encontro Científico de Robótica, Guimarães, 2006


Girard; Anouck R.; Sousa, João; Hedrick, J. Karl; A Selection of Recent Advances in Networked Multi-Vehicle Systems; Proceedings of the Institution of Mechanical Engineers, Part I, 2004 (a)


Girard; Anouck R.; Sousa, João; Hedrick, J. Karl; “Elemental Maneuvers and Coordination Structures for Unmanned Air Vehicles”, Proceedings of the IEEE Conference on Decision and Control, Paradise Island, Bahamas, 2004 (c)


Girard, Anouck Renée; Hybrid System Architectures for Coordinated Vehicle Control PhD dissertation, Spring 2002 (e)


Multi-UAV command and control system

Krasovskii, N. N.; Subbotin, A.I.; Game-theoretical control problems, Springer-Verlag, 1988

Lee, Jusuk; Huang, Rosemary; Vaughn, Andrew; Xiao, Xiao; Hedrick, J. Karl; Zennaro, Marco; Sengupta, Raja; Strategies of Path-Planning for a UAV to Track a Ground Vehicle, In Proceedings of the 2nd annual Autonomous Intelligent Networks and Systems Conference, Menlo Park, CA, June 2003. AINS

Marques, Eduardo; Gonçalves, Gil; Sousa, João; Seaware: A Publish/Subscribe Middleware for Networked Vehicle Systems, 7th Conference on Maneuvering and Control of Marine Craft (MCMC’2006), Lisboa, Portugal, 2006 (submitted)

Rauw, Marc; FDC 1.2 - A Simulink Toolbox for Flight Dynamics and Control Analysis, 2nd edition, May 10, 2001

Ryan, Allison; Xiao, Xiao; Rathinam, Sivakumar; Tisdale, John; Zennaro, Marco; Caveney, Derek; Sengupta, Raja; Hedrick, J. Karl; Collaborative flight demonstration of three unmanned aerial vehicles, Submitted to the AIAA Conference on Guidance, Navigation, and Control, Keystone, CO, 2006

Sousa, João Borges; Notes on reachability and control for continuous-time systems (Draft), December 7, 2004

Varaiya, Pravin; Hierarchical Control of Semi-Autonomous Teams Under Uncertainty (HICST), May 2004

Varaya, Pravin; Lee, Edward; Structure and interpretation of signals and systems, University of California, Berkeley, 2001

Video transmission system:
www.wirelessvideocameras.com

AsasF project website: http://whale.fe.up.pt/asasf

C3UV group website: http://www.calceit.org/c3uv/

FlightGear installation and other files:
http://www.cloudcaptech.com/download/Piccolo/FlightGear/
Piccolo References

Initial Flight Test Cards, April 2005, CloudCap documentation website (version 1.3.0): http://www.cloudcaptech.com/download/Piccolo/Version%201.3.0/Docs/


Vaglienti, B.; Hoag, R.; Miller, T.; Piccolo Aircraft Integration Guidelines, July 2005 (b)

CloudCap website: http://www.cloudcaptech.com/

CloudCap documentation website (version 1.3.0):
http://www.cloudcaptech.com/download/Piccolo/Version%201.3.0/Docs/
Appendix A - Cooperation with UC Berkeley

I. Introduction

In this appendix I will briefly describe the C3UV group at UC Berkeley and the cooperation being held as a background for the description of my work with this group.

II. C3UV

The Center for Collaborative Control of Unmanned Vehicles (C3UV) at the University of California, Berkeley is an interdisciplinary group focusing on the fundamental theoretical developments necessary to allow teams of unmanned vehicles to operate autonomously, without extensive monitoring and intervention by human operators.

They have focused primarily on unmanned aerial vehicles (UAVs) and have developed a small fleet of UAVs outfitted with commercial avionics packages, PC104 microcomputers, wireless communications, and an array of sensing capabilities. The experimental platform provides a means to validate a number of fundamental issues related to autonomous operation and collaboration of heterogeneous teams, such as sensing and control, path planning, obstacle and collision avoidance, communications, task decomposition and allocation, cooperative execution, and information sharing.

This group has been doing research on UAVs since 2002 and their first demonstration of an autonomous flight was in 2003.

It is noticeable that there are common thrusts shared by both the C3UV group and the AsasF group. A few of them are enumerated next:

1. The autopilot used by both groups is the same (Piccolo). Therefore, we both need to calibrate and tune in addition to build simulation aircraft models.
2. We both wish to have onboard processing and communication between vehicles. At C3UV they are using PC104 for processing and 802.11a wireless cards to communicate between vehicles in the air.
3. Develop a control architecture
4. Design an user interface for multiple vehicle control
5. Onboard payload (any electronic equipment onboard communicating with Piccolo)
6. Piccolo mounting issues, preflight check-list, tests and evaluation
Hence, these similar goals as well as their interest in the USTL technology are the main reasons for establishing cooperation links between the two groups. The two groups have been collaborating since the early days of C3UV. In light of this collaboration, I spent three-weeks at UC Berkeley to learn the low-level details of operation with Piccolo as well as insight on risky situations or even on constructive issues when mounting Piccolo or performing experimental tests. Another important issue is that it allowed us to test the system with HWIL simulation and validate the solution implemented.

In spite of the similarities there are differences concerning the long-term goals for each group, such as:

- AsaSf is doing research to build a control architecture for heterogeneous teams of vehicles (AUV, ASV, UAV), while they are only using UAVs for their studies.
- Our interface supports control at different hierarchical levels. At high-levels we can do task-planning, supervision or analysis while at the low level we can perform manual control of the vehicles. At Berkeley, they are only interested in higher level manoeuvres

III. August demonstration

Currently, the C3UV group is preparing a demonstration flight for the 9th of August 2006, with the following goals:

1. To have 5 aircrafts flying autonomously in a coordinated way.

2. Test new vision system to follow rivers. In previous years, this group has already tested vision-based control for following aqueducts and roads. The group is now interested in improving the vision system to follow and detect rivers. This is more difficult since we are dealing with a less structure world: rivers do not have a fixed width, the borders can be extremely different and sunlight reflex is also an issue.

3. Use new aircrafts with the following characteristics (see pictures below):
   - Custom designed aircraft (BAT-4) from MLB (http://www.spyaircrafts.com)
   - Weight (no payload): 23 Kg; Max takeoff weight: 45Kg; wingspan: 4m
   - Autonomy: 8 hours (7.6 litres of fuel)
   - Max velocity: 45 m/s; Min velocity: 20 m/s
   - The motor also works as generator for all batteries onboard (12V and 5V).
Multi-UAV command and control system

IV. Test days

During my stay we carried out field tests during two days. On one of them, both the servos and Piccolo’s gyros behaviour were tested while the engine was running. A communications test at a 0.8km distance was also successfully performed. Figure 44 shows the setup.

Figure 44: Experimental setup

The goal of the second test day was to tune Piccolo gains. The procedure for tuning is as explained in Initial Flight Test Cards.

However, on this test, the plane crashed (figure 45). On the ‘conclusions’ we state the reasons for this.
V. Conclusions

After the crash there were strong suspicions that the state of the battery was the main reason for the crash. Piccolo log file indicated that the cause of failure was in the 5V battery. In fact, they were rated for 1A*hr and it was being drawn 1.04 A. Moreover, the servo voltage immediately before the crash was 3.5V while its datasheet specifies that they stop responding at a voltage below 4V.

Thus, two things went on:

1) the servos were drawing more current than expected (almost 1 A while in flight)  
2) no attention was paid to how long the aircraft had been in flight or how long the batteries were plugged in.

This allowed us to extrapolate some conclusions for the AsasF project. We are more aware of possible causes of errors and have already studied some suitable prevention strategies. For instance, errors may be Piccolo malfunctioning or a too low throttle. The prevention solutions were already presented in Chapter 7.

There are some interesting points worth mentioning about work at C3UV. The communication with manufacturers is easy and fast. Moreover, the decision for buying new equipment as well as the time it takes for the material to arrive is considerably fast. The reason behind this is their bigger budget together with no export issues, since most manufacturers are located in the United States. This was a problem for the AsasF problem since the Picollo system is ITAR. This means that it is subject to strict export regulations.
One other particularly important concern on which we should focus our attention are plane crashes. As we could learn, the C3UV group makes mistakes despite the fact that they have been flying autonomous vehicles for 3 years. Although it was a predictable human error, it still occurred, which alerts us to the fact that we cannot predict all possible errors. Consequently, although doing the best for error prevention, it is important to be aware of the possibility of having a plane crash.
Appendix B - Guided Tour

Running Software-In-the-Loop

To run the SWIL, the main computer can be used to run the Simulator, PiccoloPC, GroundstationPC, and the Operator Interface. Alternatively, if a second computer is available, you can run the Simulator and PiccoloPC on the main computer, and run the GroundstationPC, Operator Interface, and the FlightGear visual display on the secondary computer.

- If a Windows-compatible joystick is available the GroundstationPC application will use it as Pilot console.
- Multiple PiccoloPCs can be run by using the command line switch –SN= to set the serial number of each software avionics. Thus, in the command line you would type:

  PiccoloPC –SN=3

To run the Piccolo with the serial number 3. If you just double click the application, the default serial number is 1.

Here are the steps to follow when setting up the software-in-loop simulation:

1. On the main computer, launch the PiccoloPC application.
2. On the main computer, launch the GroundstationPC application (just double click).
3. On the second computer, launch the Operator Interface with the following command line:

   operatorinterface -SERVER=[MAIN_COMPUTER]:2000

where you need to replace [MAIN_COMPUTER] with the network name of the computer running GroundstationPC. Example:

   operatorinterface -SERVER=localhost:2000

An alternative way to do this was to double-click the OperatorInterface.exe where the following menu would appear:
The default option is to connect to the groundstation over the COM1 serial port, at 57600 baud, and run as a TCP/IP server on port 2000. This is when the groundstation is connected to the computer running this OI. In case the groundstation is being simulated through GroundStationPC or you are connecting to an OI on another computer you should check the box “connect via server” and type:

[IP_OR_NAME_OF_COMPUTER]:2000

It should look like the image on the right.

4. Make sure that you see the groundstation telemetry, and add the avionics to the network addresses list if necessary.
5. On the main computer start the Simulator, and from the "External" menu select "AP Simulation" such that the simulator sends its data to a TCP/IP socket instead of CAN bus, and specify the computer name as "localhost" if PiccoloPC runs on the same computer. You must also type the serial number of the PiccoloPC that should communicate with this simulator.
6. Make sure that in the Operator Interface you can see the sensor data that is being sent from the Simulator, and in the Simulator you can see the control surfaces that are being sent by the PiccoloPC. If everything looks in order, the software simulation is ready to run.
7. To have a visual representation of the aircraft, you should use FlightGear. You must download it from CloudCap’s website (References: FlightGear) and install it. From this
website you should also download (from version 0.9.8) this file: runfgfsnet-c172.bat.

8. Double-click this file to run FlightGear on the secondary computer. Then, it the Simulator window, from the “External” menu, select “FlightGear 0.9.8”. Write the name or IP address of the computer running FlightGear. You show now be watching an aircraft. To see it moving autonomously, follow the steps described next.

After having the system ready to run, it is necessary to spend some time learning to work with the commands and interfaces. Thus, it follows a list of steps to be taken in order to have an aircraft flying autonomously in simulation:

1. On the “Map” screen of the operator interface, select “New quick plan” from the menu on the top left.
2. Click anywhere in the map and the following menu will appear:

   ![Quick plan](image)

   Figure 47: Quick plan

3. Don’t change anything in “Lat” or “Lon”, leave the default values. Type the values shown above for “Altitude” and “Radius”. Press OK button
4. Now, on the Simulator window, you have to open an aircraft model. Go to File→ Open→CubModel→Cub.txt. Insert values like shown on the picture below.
5. Press *Apply Slew*, then *Start*. Your aircraft should now be following the plan you just specified. You are now running a simulation of a vehicle flying autonomously.

![Simulator window](image)

**Figure 48: Simulator window**

**Interface between Seaware and Piccolo**

The interface is currently only working under Linux Operating System. Thus, in order to integrate it with Piccolo and Neptus you need a third computer. However, you can create a virtual machine in order to simulate this third computer. In a future development, the system will also run under Windows.

**Initiating interface**

The interface can be called in one of two ways:

1). `/pinterface [PiccoloSerialNumber] middleware-domain [GSserver IP:port] [manager IP]
2). `/pinterface [PiccoloSerialNumber] middleware-domain hw [Channel] [BaudRate] [manager IP]

Example:

1). `/pinterface APV 127.0.0.1 1 192.168.105.74:2001
2). `/pinterface APV hw 127.0.0.1 1 2 57600

- **middleware-domain** is the name by which the vehicle is known on the seaware network.
- **manager IP** can be any address, for now (it is not being used).
- **PiccoloSerialNumber** is the serial number of the Piccolo to which we want to communicate
- **GSserverIP:port** is the address and port of computer running the groundstation server to which we want to communicate
- **Channel** is the serial port to which the groundstation is connected to (1/2/3...). Write -1 to autodected USB-serial groundstation port.

- **BaudRate** is the baud rate of the communications in the serial port (should be 57600)

The first usage will initiate communications with seaware and with a GroundStation server running on computer 192.168.105.74.

The second usage will initiate communications with seaware and with the GroundStation connected to the second serial port.

**Exchanging messages with Seaware**

Currently, the messages published on Seaware by this interface are:

- Heartbeat (every second)
- EstimatedState (every 0.5 secs)
- Motor (every 0.5 secs)

The messages subscribed by the Interface are:

- Heartbeat (arrives every second)
- Joystick (arrives every time there are changes in joystick)

In order to have new messages exchanged it is necessary to modify the XML file (RovMiddlewareConfig.xml) and compile it with a java compiler (version 1.5)

**Compiling the code in Linux**

In order to compile this code in Linux, a few OS requirements have to be met:

- libc6 package installed
- g++-3.3 (and gcc-3.3) installed (it is compulsory to compile the code with this version)
- java Runtime Environment version 1.5
- add the following lines to the .bashrc file on the user directory, with the proper modifications.

The generic version is given first, and then an example is shown:

```bash
export JAVA_HOME=[ABSOLUTE_PATH_OF_JAVA_DIRECTORY]
export MW_LIB_HOME=[ABSOLUTE_PATH_OF_SEAWARE]
export PATH=$MW_LIB_HOME/bin:$JAVA_HOME/bin:$PATH
```
Multi-UAV command and control system

Example:

```bash
export JAVA_HOME=/home/asaf/jdk1.5.0_07
export MW_LIB_HOME=/home/asaf/seaware
export PATH=$MW_LIB_HOME/bin:$JAVA_HOME/bin:$PATH
```

Running the interface in Linux with Neptus in Windows (NDDS)

The Windows requirements to make Neptus work correctly are:
- JAVA 3D version 1.4.0
- QuickTime 6

After starting the interface, you should start Neptus. The first thing to do is to specify the RTPS transport to use. Go to Tools→Edit General Preferences→RTPS transport to use. Here, you should write NDDS. To see the aircraft’s console, go to File→Open→APV-3 Console. Now you should be able to see the aircraft attitude in Neptus. To see a camera fixed with the aircraft, press the button. To change camera position, zoom in/out press the button.

Running system with ORTEMANAGER

In case you do not want to use NDDS but want to use ORTE, you need to specify the IP address of the computer running it on the field ORTE middleware manager address.

In this case, you should first start the manager before starting other applications. To start ORTEMANAGER you should use the following command in Windows or Linux:

```bash
./ortemanager -d 1 -n 255.255.255.255 -e
```

To tell the interface to use ORTE you need to change some code in pinterface.cpp and write the correct manager IP address when you start it (see above).