Graphical user interface development for ocean models

Noel, A.

Publisher's version / Version de l'éditeur:
https://doi.org/10.4224/8895453
Student Report (National Research Council of Canada. Institute for Ocean Technology); no. SR-2006-04, 2006

DOCUMENTATION PAGE

REPORT NUMBER	NRC REPORT NUMBER	DATE
SR-2006-04		April 19, 2006
REPORT SECURITY CLASSIFICATION		**DISTRIBUTION**
Unclassified		Unlimited

TITLE

GRAPHICAL USER INTERFACE DEVELOPMENT FOR OCEAN MODELS

AUTHOR(S)

Adam J. G. Noel

CORPORATE AUTHOR(S)/PERFORMING AGENCY(S)

PUBLICATION

Institute for Ocean Technology

SPONSORING AGENCY(S)

Institute for Ocean Technology

IOT PROJECT NUMBER	NRC FILE NUMBER
PJ2019	

KEY WORDS	PAGES	FIGS.	TABLES
Graphical user interface, industry company profile, software, manoeuvring	17 + app.	0	0

SUMMARY

An overview of the field of ocean research is presented, including a summary of organizations that are active in the Canadian Atlantic region. The National Research Council's Institute for Ocean Technology (NRC-IOT) is described with respect to its capabilities to conduct research and its primary focus areas. Finally, the role of the author throughout the Winter 2006 work term is discussed, outlining the development of a graphical user interface (GUI) for ocean software models.

ADDRESS National Research Council
Institute for Ocean Technology
Arctic Avenue, P. O. Box 12093
St. John's, NL A1B 3T5
Tel.: (709) 772-5185, Fax: (709) 772-2462

National Research Council Canada
Institute for Ocean Technology

Conseil national de recherches Canada
Institut des technologies océaniques

GRAPHICAL USER INTERFACE DEVELOPMENT FOR OCEAN MODELS

SR-2006-04

Adam J. G. Noel

April 2006

ACKNOWLEDGEMENTS

Gavin Earle, Ping Xiao, Jiancheng Liu, and Dr. Wayne Raman-Nair provided in-house software models. Their assistance in integrating their work with the author's is greatly appreciated.

Wayne Pearson, Greg Janes, Jim Millan, and Bruce Quinton provided advice on various aspects of the Matlab environment, and a thank-you goes to them as well.

Finally, the author also thanks Dr. Michael Lau for continuous support and mentoring throughout the Winter 2006 work term.

ABSTRACT

An overview of the field of ocean research is presented, including a summary of organizations that are active in the Canadian Atlantic region. The National Research Council's Institute for Ocean Technology (NRC-IOT) is described with respect to its capabilities to conduct research and its primary focus areas. Finally, the role of the author throughout the Winter 2006 work term is discussed, outlining the development of a graphical user interface (GUI) for ocean software models.

TABLE OF CONTENTS

APPENDICES

1.0 INTRODUCTION

This report is the author's industry company profile (ICP), completed as required for Memorial University of Newfoundland's Faculty of Engineering and Applied Science. The ICP is a requirement for work term 2 and will be graded by Co-operative Education. Secondly, this report satisfies the requirement of a technical report for the National Research Council of Canada's Institute for Ocean Technology (NRC-IOT), where the author completed the Winter 2006 work term.

An overview of the field of ocean research is presented, including a summary of organizations that are active in the Canadian Atlantic region. IOT is described with respect to its capabilities to conduct research and its primary focus areas. Finally, the role of the author for the work term is discussed. This component outlines the development of a graphical user interface (GUI) for ocean software models.

A number of other reports were completed in conjunction with this report that focus on the creation and use of the GUIs. The "User's Manual for MTD3D" (Noel, 2006) and "User's Manual for OSIS (Ocean-Structure Interactions Simulator) v0.5 – Version 1" (Noel and Lau, 2006) are general use references for two versions of the GUI. The development and programming details of OSIS is discussed in much greater detail in "GUI Programmer Manual for OSIS (Ocean-Structure Interactions Simulator) – Version 1" (Noel and Lau, 2006). Please refer to these documents for additional information.

2.0 PROFILE OF OCEAN RESEARCH

2.1 Origins and History

Human beings have been using the oceans of the world for different purposes for thousands of years. Sailing vessels throughout history were designed for transportation, warfare, and the gathering of resources.

Primitive humans began ocean travel by vessel, likely using simple rafts. By 6000 BC, hallowed logs were being used for fishing off the coasts of northern Europe. War craft have existed on the Nile since 9000 BC (Woodman, 1997). From these early times, vessels have undergone significant evolutions.

The fishery was the predominant resource for many centuries, however the use of natural gases for transportation and heating, combined with rising oil prices, has led to the importance of the offshore oil and gas sector. Onshore oil wells have been present in North America since the nineteenth century (McKenzie-Brown *et al.*, 1993), however it was not until 1947 that offshore production began by American companies in the Gulf of Mexico (Pratt *et al.*, 1997). The rising cost of oil has steadily driven the industry out to deeper and harsher waters in environments that are more hostile, such as those in northern climates.

Much of the field of ocean research today is thus somehow related to the offshore oil and gas industry, which has a very strong presence in the Atlantic region. Offshore wells have been drilled in this region for over forty years, however it is only within the last fifteen years that the commercial extraction of crude oil has begun in Atlantic Canada (Locke, 1999). Ocean research has investigated many aspects of this process, including the dynamics and design of oil platforms, tankers, risers, and emergency craft.

2.2 Presence of the Field Today

A number of organizations today are involved with ocean research. Due to geographical considerations, there is a strong presence in the Canadian Atlantic region, particularly in Newfoundland. A significant proportion of the research and development performed is in relation to operations in harsh environments, such as extreme climates, deep waters, and the presence of ice. Studies continue to be made, not only in relation to offshore oil and gas, but also with general cargo transport, fishing and aquaculture, competitive racing, and surveillance.

The Newfoundland Ocean Industries Association (NOIA) was formed in 1977, with a broad mandate to be a collective body of companies with interests in ocean industries. Today the NOIA has about five hundred member companies in Eastern Canada, focussed on the oil and gas industry. The Association members have been involved with all facets of the industry, including development, production, construction, transport, processing, and research and development. The NOIA itself is a non-profit organization providing lobbying and networking opportunities to the industry (Newfoundland Ocean Industries Association, 2006).

Petroleum Research Atlantic Canada (PRAC) aims to foster research and development in the petroleum industry by supporting inter-disciplinary work. Beginning operations in 2002, the PRAC is a public-private partnership providing funding via grants and contract research. Completed projects cover the entire spectrum of fields in the industry, from numerical modelling and safety tests to geological studies and socio-economic benefits (Petroleum Research Atlantic Canada, 2006). Its members include oil and gas corporations such as Petro-Canada and ExxonMobil, provincial and federal government departments, and Atlantic Canada academic institutions such as Memorial University of Newfoundland (MUN) and Dalhousie University. It should be noted that PRAC is a member body of the NOIA.

A local corporation based in St. John's that is a key partner in ocean research is C-CORE. Formed in 1975, C-CORE has facilities on the MUN campus and they have a long history of working with the offshore oil and gas industry. As a corporation, their work is conducted via funded projects and contracts for research and development and the applications of new technologies. C-CORE has expanded its interests to include other fields such as mining and pulp and paper. While initially working almost exclusively with large corporations, downsizes in the 1980s prompted a shift to partnering with small and medium size enterprises, a trend that continues to this day (C-CORE, 2006). C-CORE is also a member body of the NOIA.

It would be more than reasonable to assume that any industry involving research will have extensive connections to the academic world. While the aforementioned organizations have strong ties to academia, there are others that are even more integrated with post-secondary institutions. The Centre for Marine Simulation (CMS) is an operational unit of the School of Maritime Studies at the Marine Institute. Its advanced technology was created for the training of navigators and other mariners for harsh environment, yet has evolved to play a role in prototyping, development of work procedures, and testing of new equipment. For example, CMS has been used for simulating the towing of the Hibernia gravity-based structure and for Coast Guard training programs (Centre for Marine Simulation, 2006). Secondly, the Ocean Engineering Research Centre (OERC) is an integrated part of MUN's Faculty of Engineering and Applied Science Ocean and Naval Architectural program. The OERC has its own towing tank for conducting research efforts, and its facilities are available to students as well as faculty and contract clients (Faculty of Engineering and Applied Science, 2006).

Other corporations have fewer ties to the academic world. One such corporation is Newfoundland Aerospace. The company has an expansive range of operations, from controlling

the province's primary flight carrier to conducting surveillance of Canada's oceans for the federal government. Extensive research is undertaken for the development of their surveillance craft for the east and west offshore regions of the country (Scott, 2006). Another corporation, Marport Ltd., is creating sophisticated sensors for trawlers and attempting to introduce wireless broadband for offshore environments (Riggs, 2006). These companies are connected via OceansAdvance, an ocean technology cluster development forum. OceansAdvance seeks ocean-related industry growth as a whole by improving communications between different companies, academic institutes, and government (O'Reilly, 2006).

2.3 Continuing Evolution

It suffices to say that the field of ocean research in the Atlantic region is one that is continuously evolving. The development of new technologies is creating many opportunities for the field. As a whole, the ocean/marine technology sector is the fastest growing in the province, with growth of 15-20% annually from 1998 to 2003 (O'Reilly, 2006).

As the technologies driving marine simulations and design advance, the importance of teamwork grows. The field will expand with collaborations of groups and individuals of different technical backgrounds. Organizations such as the NOIA and OceansAdvance will foster partnerships so that projects can be completed in as effective a manner as possible. Within companies, it will become more important to have individuals from various fields of engineering, arts, economics, and the sciences.

The prominence of the offshore oil and gas industry is expected to continue for at least the next twenty years; society's need for these resources will fuel research projects in this area. This includes the development of oil and gas projects in northern environments.

3.0 IOT PROFILE

3.1 Background

The Institute for Marine Dynamics (IMD) was opened in 1985 by the National Research Council of Canada (NRC). The Institute was created with the goal of supporting ocean technology industries across the country by combining experience with extensive facilities. As a government facility, the Institute has provided a broad range of services and products to public, private, and sporting ocean industries. The Institute was renamed the Institute for Ocean Technology (IOT) in 2004 to better reflect its achievements and goals.

3.2 Facilities

The facilities located and maintained at IOT are among the most extensive in the world for ocean research. Each component offers unique services and opportunities.

The three major facilities are the Offshore Engineering Basin, the Towing Tank, and the Ice Tank. The Basin can produce waves, current, and wind, re-creating the harsh ocean environment for testing oil platforms and other models. Ship models can be up to 4.5 metres in length, while platform structures can be up to 6 metres in diameter. The Towing Tank is 200 metres in length and has its own wave-maker. It can support ship models up to 12 metres in length or platforms up to 4 metres in diameter. The Ice Tank, 90 metres in length, can use ice sheets of length up to 76 m, the longest in the world, while also doubling as a second tow tank.

Many other resources complement the major facilities. There is a cavitation tunnel for performing tests on propellers and conducting other force and pressure tests. Two cold rooms, one large and one small, can obtain temperatures as low as –40 degrees Celsius and create ice for the ice tank while being able to perform their own tests for crystal structure and other model ice

properties. Various pieces of equipment are available for test measurements, from capacitance probes for waves and dynamometers for propeller shafts, to more advanced mechanisms. The Yacht Dynamometer was designed specifically for measuring lift and drag on models for the America's Cup competition. The Marine Dynamic Test Facility can monitor underwater vehicles moving with six degrees of freedom. The Planar Motion Mechanism (PMM) is used for manoeuvrability studies, particularly those of models in ice conditions.

IOT also has the capability of fabricating the models used at its facilities, with specialized equipment for working with small-scale construction. The computerized five-axis milling machine can make wooden and high density foam models up to 12 metres in length.

3.3 Organizational Structure

As noted, IOT is one of the Institutes of the National Research Council of Canada (NRC). NRC is a federal government agency, so IOT receives its funding directly from the federal government.

The head of the Institute is the Director General, a position currently held by Dr. Mary Williams. The rest of the Institute is divided into seven major categories: research, facilities, quality and planning, human resources, communications, finance, and business development, each with their own respective director. The majority of the employees work in the research and facilities divisions, since the focus of IOT is as a research facility. The facilities division can be further divided into the ice tank, electronics, computer systems, software engineering, open water test facilities, design and fabrication, and engineering support and maintenance.

Much of the research conducted is project based, with each project having a supervisor and other members. Members can be directly involved with multiple projects at once, depending on the time required and the individual expertise they bring. Projects may or may not be directly

applicable to industry, however most can be placed into one of the five focus research areas that IOT has identified.

The first research area is that of ships and structures in ice, with the goal to predict ship and structure performance in ice conditions using a combination of numerical and physical models. A more general area is that of performance evaluation in the ocean environment, which investigates performance at sea but not necessarily in ice. The area of underwater vehicle systems is developing the means for exploration and manipulation of sub-sea environments. This area is similar to that of applied hydrodynamics, which predicts structure dynamics for sub-sea resource discovery. The marine safety area tests lifesaving systems and safe operation of smaller vessels. All of these focus areas cover the broad scope of supporting ocean technology industries. Additionally, unique ideas can be allocated short-term efforts to determine the feasibility of further study.

3.4 Client Base

As a research facility, IOT maintains a unique client base with connections to both academia and industry. It has formed partnerships with a number of organizations, both directly and indirectly.

The facilities over the years have been used for testing models of various ferries, oil platforms, ice-breakers, racing yachts, life craft, ships, and submersibles. Clients for these tests include crown corporations such as Marine Atlantic, large partnerships such as the Hibernia development project, and competitors in the America's Cup races. These services have generally been arranged via contract. The internal project groups for research, who can in turn offer their results to companies in the industry at large and have their work presented to the academic community, use a significant portion of the facilities.

Currently, most of the external contracting is conducted via Oceanic, which was created as a collaboration of IOT and Memorial University of Newfoundland. Formed in 1998, Oceanic offers commercial and public enterprises access to all of IOT's testing facilities, and their clients have included designers of commercial ships, fixed and floating offshore platforms, government ships, passenger vessels, and sub-sea systems.

IOT helps start-up companies with their ventures via OTEC, the Ocean Technology Enterprise Centre. Various programs within OTEC, such as the Young Entrepreneurs Program and the Ocean Technology Co-Location Program, offer funding and/or facilities to new companies in the field. Current co-locating companies include Madrock Marine Solutions, a developer of marine safety products, LewHill Testing Technologies, a developer of temperature and pressure calibration systems, NavSim Technologies, a designer of electronic navigation systems, and Shearwater Geophysical, who conduct seismic consulting. Current participants in the Young Entrepreneurs Program include Virtual Marine Technology and WES Power Technologies.

3.5 Future Development

IOT will continue to perform as a local, national, and worldwide leader in ocean technology research. It will adapt to the evolution of industry needs as required, and foster the formation of other organizations within the St. John's ocean engineering technology cluster.

Since most of the facilities within the Institute are now over twenty years old, a significant effort is in place for infrastructure replacement. High maintenance costs, decreased reliability, and the use of many components beyond their expected operational life are driving the need to replace equipment. A five-year plan formed in 2004 has been dedicating hundreds of thousands

of dollars annually to this effort. The infrastructure replacement is expected to assist in the Institute's goal to offer state-of-the-art facilities.

IOT will remain supportive of OTEC, which will continue to promote growth and development of companies in the field. This will be key in ensuring that the field of ocean research will maintain growth in the province. As noted, ocean technology is the fastest growing sector in Newfoundland and Labrador, so it is expected that there will be a growing market to benefit from ocean research.

4.0 STUDENT'S ROLE

4.1 Primary Roles

One of IOT's primary research areas is that of ships and structures in ice conditions, with the goal to make predictions from models to determine the reaction and performance of ships and structures in ice. Software development project PJ2019 is part of this research initiative and includes the development of a model for a moored cone in ice, eventually expanding to other model cases. The software model requires an effective graphical user interface (GUI), whose design and implementation was the primary focus of this author's experience.

The GUI faced a number of requirements for its design. It was necessary to create an interface that would be accessible to users who would not be expected to fully understand the models upon which the software was based. The interface needed to be sufficient for use without having to enter or modify any programming code, contrary to many models currently available. The interface itself could not be limited to a single model, but rather act as a platform for different models that could be integrated as the software evolved. Finally, there had to be sufficient documentation included so that other programmers could easily pick up and expand the code.

For the duration of the work term, it was expected that the GUI would be developed and made available as proof of concept to showcase the capabilities that an intuitive and integrated interface could have when applied to pre-existing and newly developed models. An attempt would be made to implement a variety of ideas. Matlab was chosen as the development language because of its GUI-creating tools, its high-level language format, and its presence as a language for model design.

In addition, time was also dedicated for the editing of various research reports that were in different stages of development, from initial drafts to final compiled copies. The format to be followed was that specified in the "IOT Publications Manual" (LeBlanc *et al.*, 2004), which provided guidelines for font, heading styles, figures, references, and appendices. Many of the reports required re-formatting to adhere to the guidelines.

4.2 Other Duties

A number of other tasks and activities were completed by the author. Meeting times and locations were arranged and booked, and minutes were created and sent to attendees for all project group meetings. A research video collection was in the process of being upgraded to DVD; the video inventory was updated, re-organized, and the videos were placed into the order specified by the inventory. The collection was comprised mostly of model tests, including those captured during the tests of the *Kulluk* scale model as part of PJ2019, as well as other tests throughout the last twenty years. About two hundred discs made up the inventory.

4.3 The Work Environment

The author's work term was conducted at the National Research Council's Institute for Ocean Technology in an office environment. The GUI design and implementation was performed independently, however there was considerable interaction with other individuals, including all of those involved with the software development project PJ2019. The project supervisor, Dr. Michael Lau, was in continued contact throughout the semester via email and regular meetings. The other project members, Gavin Earle and Ping Xiao, were also in regular contact. They were involved with providing feedback on aspects of the GUI, especially regarding models they had developed or modified for integration.

The GUI was created completely in Matlab Version 7 (Release 14, Service Pack 2), in part using its GUI Development Environment (GUIDE). An Intel Pentium 4, Hyper-Threading 3.0 GHz processor was used with 512 MB of RAM and running Windows XP Professional.

4.4 Job Challenge and Educational Enhancement

The work conducted required numerous skills to be completed in an effective manner, the most apparent being able to learn independently as necessary. The author had no previous experience with Matlab, however he had created a preliminary GUI within two weeks of exposure. Working in Matlab was a continuous process of learning, especially when adding new features to the platform and de-bugging the code, while also providing an outlet for creativity when designing the interface and organizing the platform. A portion of related coding was performed in Fortran, with which the author also had no previous experience. Having completed two courses in programming with C++ was certainly beneficial to learning Matlab and Fortran, but it was also necessary to reference official documentation, educational texts, and online help services. It was also useful to speak with other individuals at the institute who worked with the Matlab and Fortran environments. The ability to independently learn computing languages will be an invaluable asset, not only from a programming perspective but also for learning in general.

The report editing required strong grammatical knowledge, an eye for identifying discrepancies, and a desire to ensure that a given paper was as consistent as possible. The author gained exposure to technical writing in a research format while also learning about the field of structure-in-ice interactions. These abilities extended to the compiling of the reports for the GUI platform, including the user manuals and programming documentation, in addition to the creation of minutes.

It is expected that the work term will have a substantial influence on the author's long-term goals. The team-focussed project environment will lend itself to any other professional experiences, particularly those requiring communication and organizational abilities. The programming aspect will be beneficial to any future need to acquire more programming skills. Finally, the documentation experience will go a long way to develop the writing abilities needed as a professional engineer, academically or otherwise.

4.5 GUI Development

The process of GUI creation for PJ 2019 and related projects was, as noted, the primary focus of the author's work. This process included interface design, implementation of exporting capabilities, and integration with pre-existing models. The GUI evolved a number of times to adapt to the requirements. Refer to Appendix A: GUI Figures for a series of figures highlighting the changes made to the GUI. All of the details of the development history are provided in "GUI Programmer Manual for OSIS (Ocean-Structure Interactions Simulator) – Version 1" (Noel and Lau, 2006).

4.5.1 Moving Triangle Demonstration

The first iteration of the GUI was to create a simple interface for a user to specify simulation parameters, view the results as they were generated, and conduct analysis with the results. The program was named Moving Triangle Demonstration, and was created within a few weeks of being introduced to Matlab. The simulation plotted a triangle moving on a two-dimensional plot. The plot could be exported as a figure file, and its data could be exported to a simple Excel spreadsheet or text file. A previous progress report on this iteration is described in Appendix C of "User's Manual for MTD3D" (Noel and Lau, 2006).

4.5.2 Moving Triangle Demonstration in Three Dimensions

The next logical step was to expand the simulation to the third dimension and modify the interface to display it. Aptly named Moving Triangle Demonstration in Three Dimensions (MTD3D), the program plotted a cone object moving on a three-dimensional plot.

Gradually, over the period of a month and a half, many other features were introduced to MTD3D. AVI video generation, settings files, and customized figure editing were among the capabilities introduced. The controls for the settings became organized into different windows based on category. Readouts displayed the current co-ordinates of the cone while its position was re-calculated. The simulation performing the calculations changed from arbitrary trigonometric formulas to a preliminary Simulink model designed by Gavin Earle. A separate interface from within MTD3D was created to write the input file required by the "mooring_system" Fortran program. The user manual for MTD3D is "User's Manual for MTD3D" (Noel and Lau, 2006).

4.5.3 Ocean-Structure Interactions Simulator

MTD3D was found to have inherent limitations that prevented it from fulfilling the GUI requirements, although it was very useful for internally showcasing what an interface could do. There was no way to easily implement multiple models, i.e. create a platform, without re-designing the program. The nature of models to be included did not lend well to displaying results as they were calculated; the Simulink model, for example, needed to run in full before results became available.

The GUI was re-built to address these needs, in some cases recycling much of the code. It was named the Ocean-Structure Interactions Simulator (OSIS) to be more inclusive of its capabilities. Its implementation occupied much of the final month and a half of the author's term. OSIS separates simulations from analysis and uses custom text files for settings, simulation

output, and video configurations. The text files are created such that they can be read and understood if opened external of Matlab. A user can specify the model to be used in a specific simulation so that a relevant configuration window appears. The analysis allows plotting of any output data created by the simulation. AVI videos can be previewed and created to plot a specified object with given settings.

OSIS has already been used as a platform for ocean software models. Donald E. Nevel's Fortran "Cone" (Nevel, 1992) program was compiled by the author for use with Matlab and then completely integrated with OSIS. Dr. Wayne Raman-Nair's model for a steel riser in water (Raman-Nair and Baddour, 2005), created in Matlab, has had parameters accessible to the OSIS interface for input and output analysis. As part of PJ 2019 and PJ 2114, a combination of models created by Ping Xiao and Jiancheng Liu to calculate time-domain forces and displacements were introduced to OSIS for manipulation.

All of the code components of OSIS have been extensively documented in "GUI Programmer Manual for OSIS (Ocean-Structure Interactions Simulator) – Version 1" (Noel and Lau, 2006). This report also includes development history and the process of adding new models and object types. General use of OSIS is described in "User's Manual for OSIS (Ocean-Structure Interactions Simulator) v0.5 – Version 1" (Noel and Lau, 2006).

OSIS will be expanded even more as PJ 2019 continues. It already includes a significant number of features without a user needing to modify any code by providing continuous interaction with an interface. Appendix B: Matlab Programming and GUI Fundamentals was created as a reference for any programmer new to Matlab or GUI design, and should be a helpful supplement to the official Matlab documentation.

5.0 REFERENCES

C-CORE, 2006. "C-Core's History." Accessed 30 Mar. 2006. URL: http://www.c-core.ca/about/history.php

Centre for Marine Simulation, 2006. "About Us: Capability Statement." Accessed Mar. 31, 2006. URL: http://www.mi.mun.ca/cms/cms_capability_statement.htm

Faculty of Engineering and Applied Science, 2006. "Ocean Engineering Research Centre (OERC)." Memorial University of Newfoundland. Accessed 31 Mar. 2006. URL: http://www.engr.mun.ca/research/centres/OERC/

LeBlanc, T., Green, E., and R.E. Baddour, 2004. "IOT Publications Procedures Manual." NRC/IOT Report LM-2004-32, Institute for Ocean Technology, National Research Council of Canada, St. John's, NL.

Locke, W., and Community Resource Services, 1999. Harnessing the Potential – Atlantic Canada's Oil and Gas Industry. Strategic Concepts, Inc.

Mckenzie-Brown, P., Jaremko, G., and D. Finch, 1993. The Great Oil Age: The Petroleum Industry in Canada. Calgary: Detselig Enterprises Ltd., pp. 14-15.

Nevel, D.E., 1992. "Ice forces on cones from floes", Proceedings of IAHR '92, 11th International Symposium on Ice Problems, Banff, Alta., Vol. 3, pp. 1391–1404.

Newfoundland Ocean Industries Association, 2006. "About NOIA." Accessed 10 Apr. 2006. URL: http://www.noianet.com/about/

Noel, A., and Lau, M., 2006. "GUI Programmer Manual for OSIS (Ocean-Structure Interactions Simulator) – Version 1", NRC/IOT Report LM-2006-02 (Protected), Institute for Ocean Technology, National Research Council of Canada, St. John's, NL.

Noel, A., and Lau, M., 2006. "User's Manual for MTD3D", NRC/IOT Report SR-2006-09, Institute for Ocean Technology, National Research Council of Canada, St. John's, NL.

Noel, A., and Lau, M., 2006. "User's Manual for OSIS (Ocean-Structure Interactions Simulator) v0.5 – Version 1", NRC/IOT Report LM-2006-01 (Protected), Institute for Ocean Technology, National Research Council of Canada, St. John's, NL.

O'Reilly, L., 2006. "OceansAdvance Forum – An Opportunity to Exchange Ideas." Institute for Ocean Technology. St. John's, NL, 6 Apr. 2006.

Petroleum Research Atlantic Canada, 2006. "What is PRAC?" Accessed 30 Mar. 2006. URL: http://www.pr-ac.ca/prac_4245.html

Pratt, J.A., Priest, T., and C.J. Castaneda, 1997. Offshore Pioneers: Brown & Root and the History of Offshore Oil and Gas. Houston: Gulf Publishing Company, p. xii.

Raman-Nair, W., and Baddour, R.E., 2003. "Three-Dimensional Dynamics of a Flexible Marine Riser Undergoing Large Elastic Deformations", Multibody System Dynamics, Vol. 10, pp. 393-423.

Riggs, N., 2006. "OceansAdvance Forum – Marport Ltd." Institute for Ocean Technology. St. John's, NL, 6 Apr. 2006.

Scott, D., 2006. "OceansAdvance Forum – Newfoundland Aerospace." Institute for Ocean Technology. St. John's, NL, 6 Apr. 2006.

Woodman, R., 1997. The History of the Ship: The comprehensive story of seafaring from the earliest times to the present day. London: Conway Maritime Press, p. 12.

Appendix A

GUI Figures

The following appendix serves as a general timeline of the graphical user interfaces (GUIs) created in the development of MTD, MTD3D, and finally OSIS. The first five figures, which show the windows of MTD, were captured from Matlab's Graphical User Interface Development Environment (GUIDE) and then converted to '.png' pictures. The remaining figures were captured as the respective program executed, so they are more representative of what the window actually looked like. Please note that some of the final MTD3D figures were captured after the original OSIS figures were created.

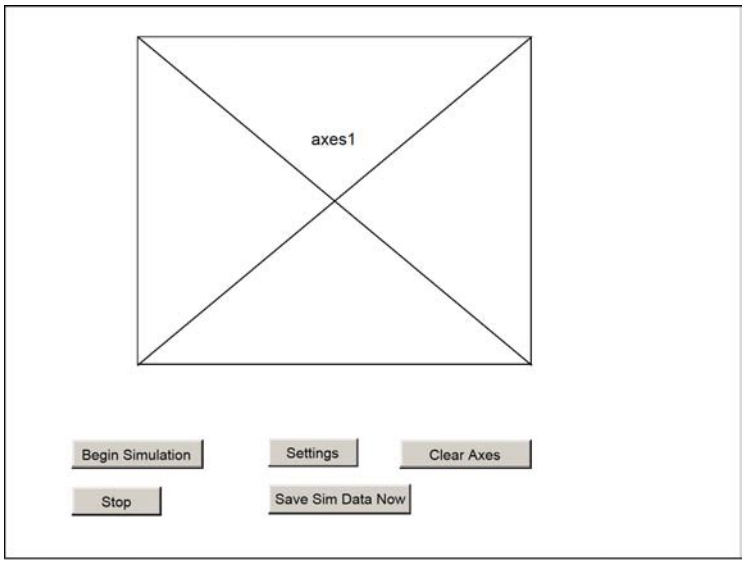

Figure A-1: MovingTriDemo window (January 16, 2006)

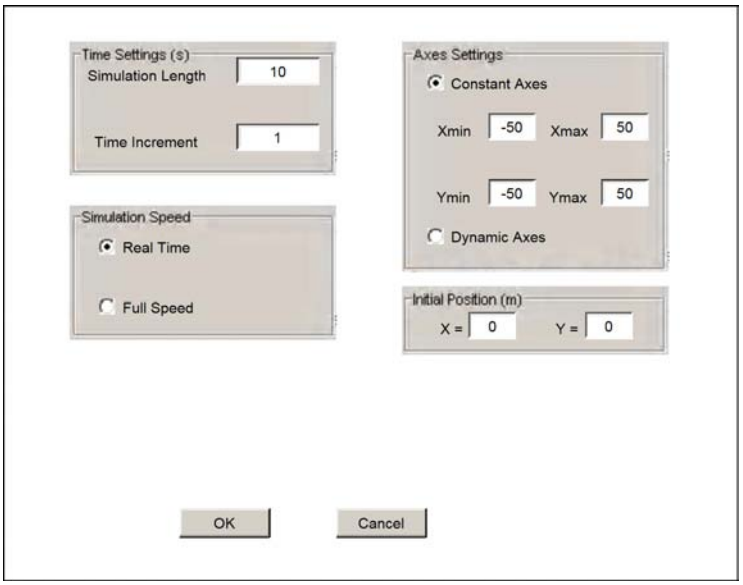

Figure A-2: MTDSettingsGui window (January 16, 2006)

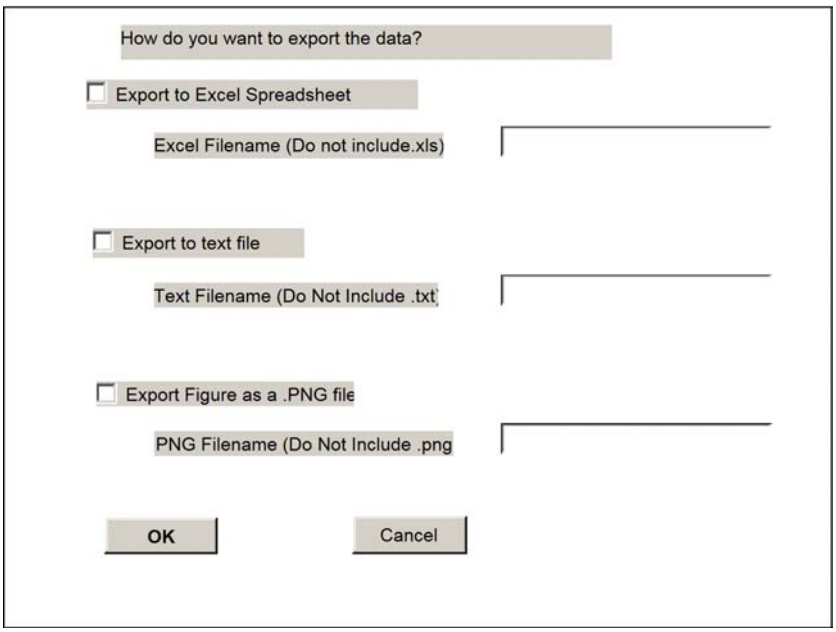

Figure A-3: DataExport window (January 16, 2006)

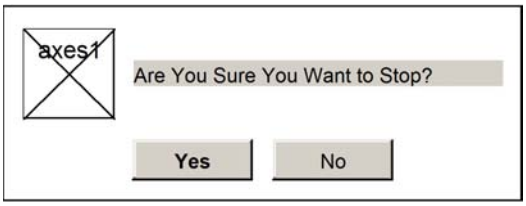

Figure A-4: confirmStop window (January 16, 2006)

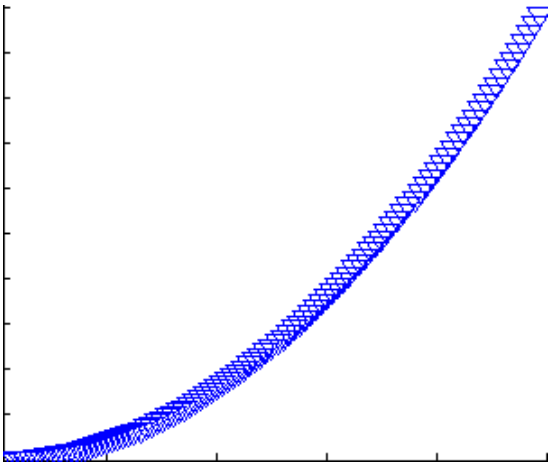

Figure A-5: Sample plot of y = x^2 by MTD (January 16, 2006)

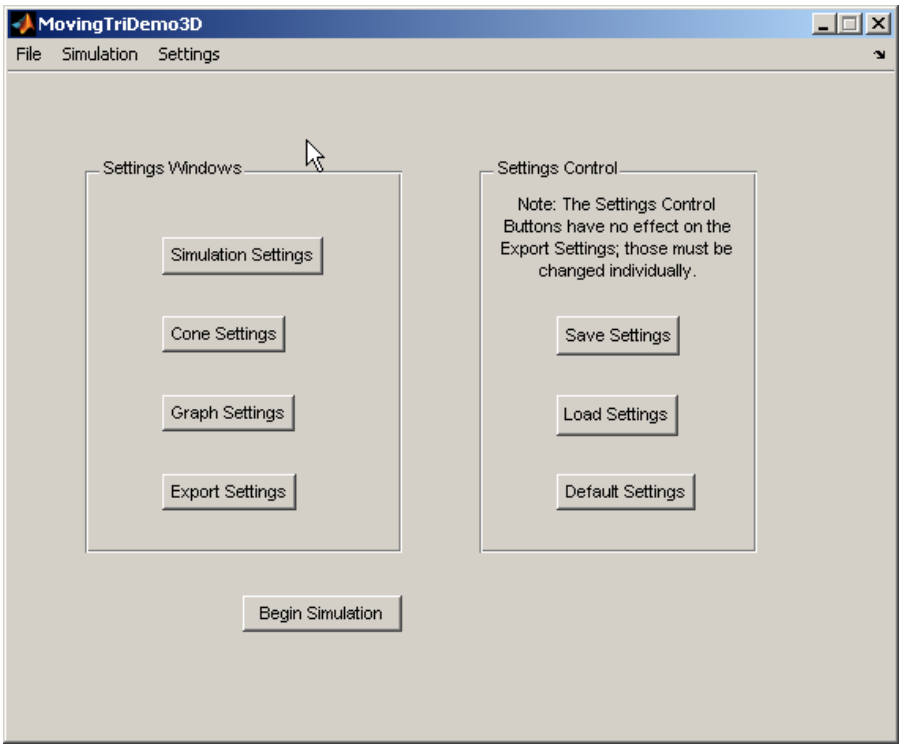

Figure A-6: MovingTriDemo3D opening window (January 31, 2006)

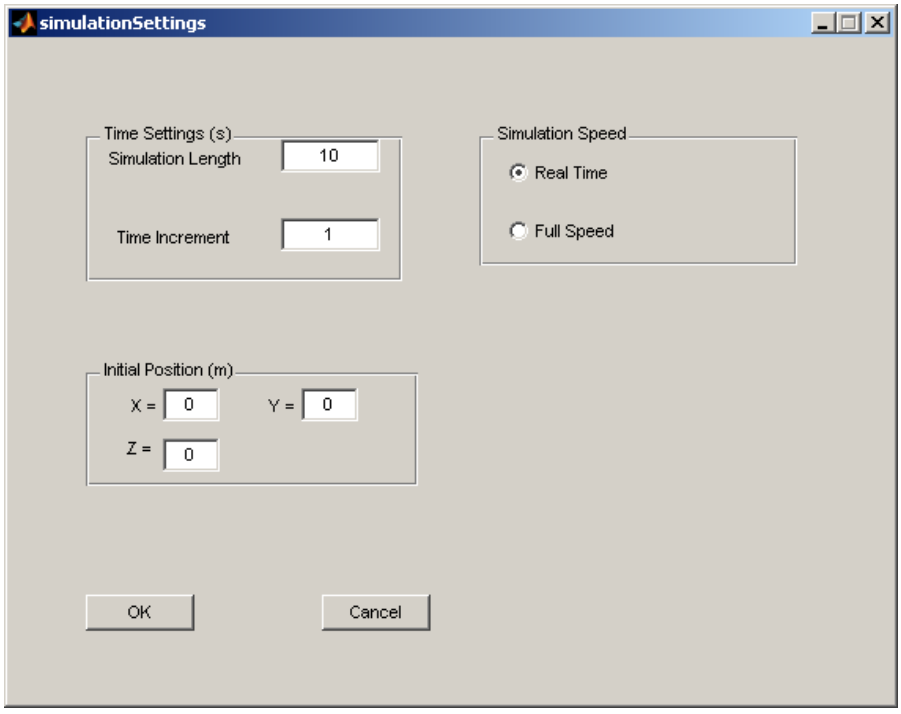

Figure A-7: MTD3D Simulation Settings window (January 31, 2006)

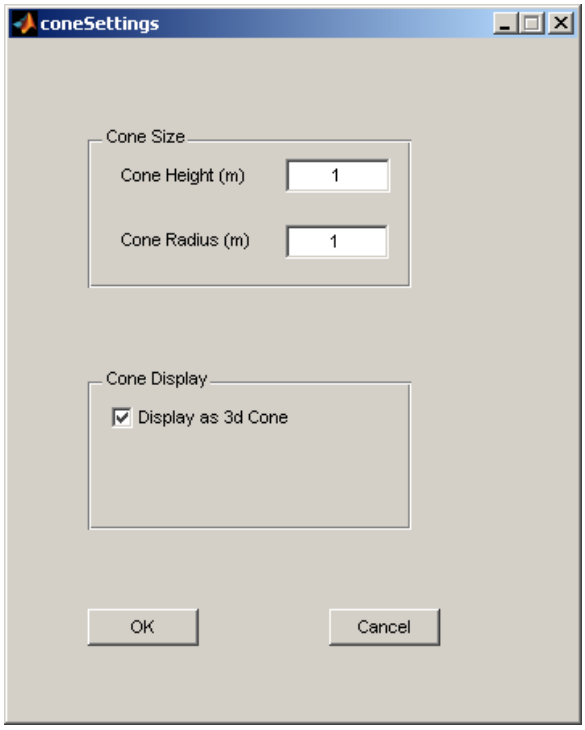

Figure A-8: MTD3D Cone Settings window (January 31, 2006)

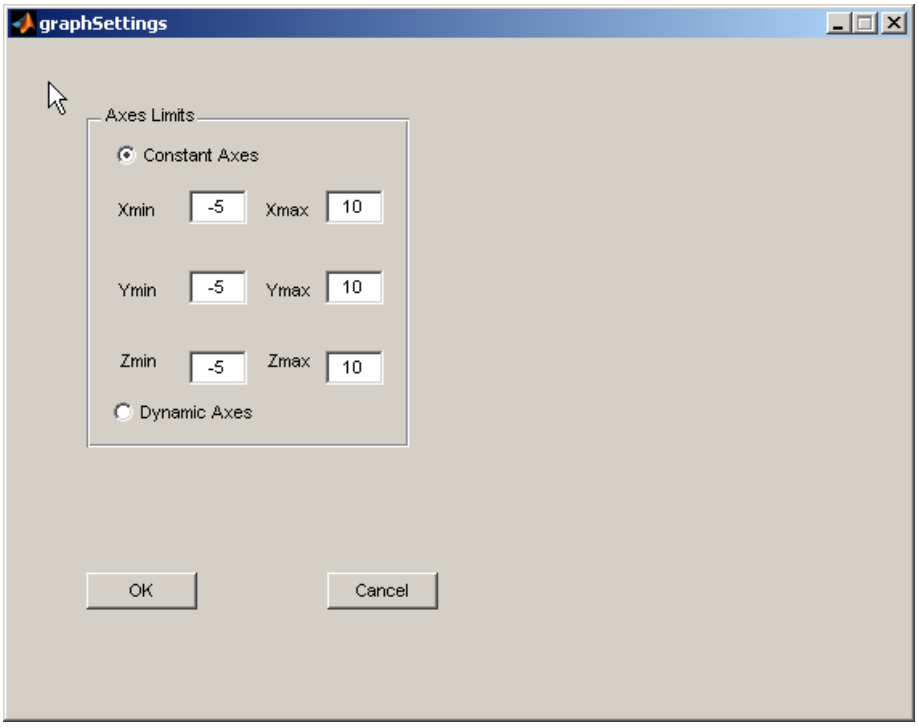

Figure A-9: MTD3D Graph Settings window (January 31, 2006)

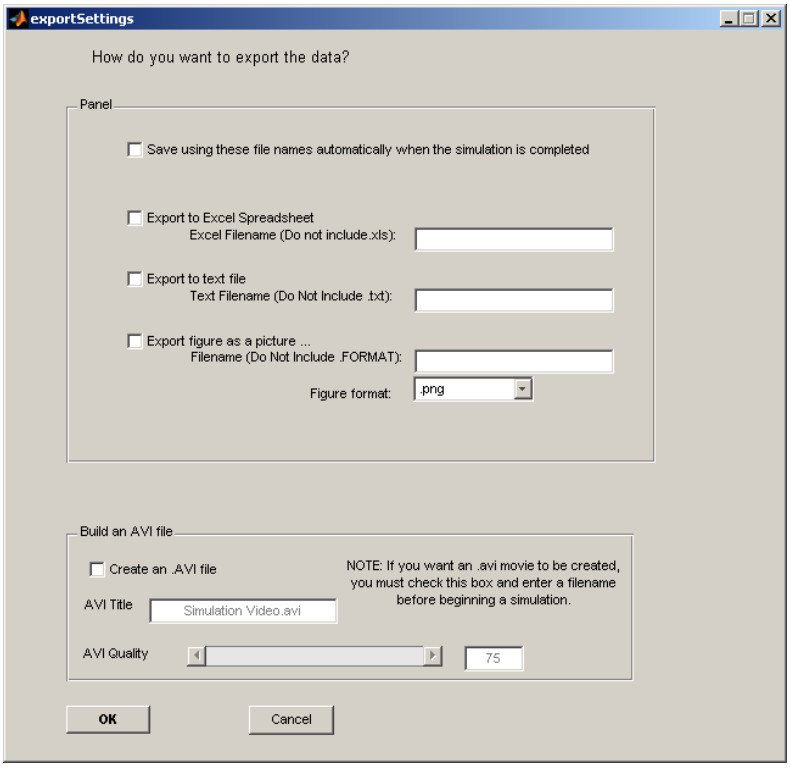

Figure A-10: MTD3D Export Settings window (January 31, 2006)

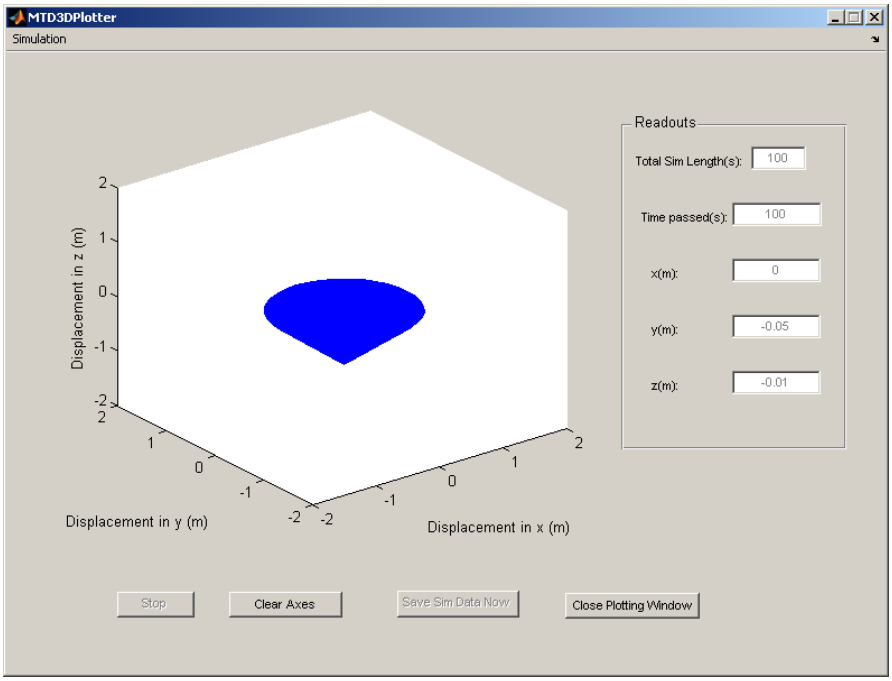

Figure A-11: MTD3D Plotter window (January 31, 2006)

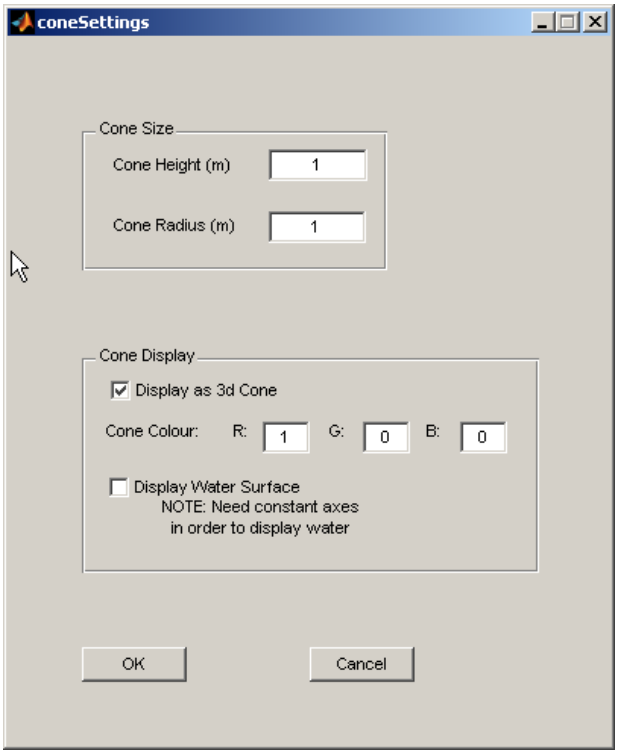

Figure A-12: MTD3D Cone Settings window (February 9, 2006)

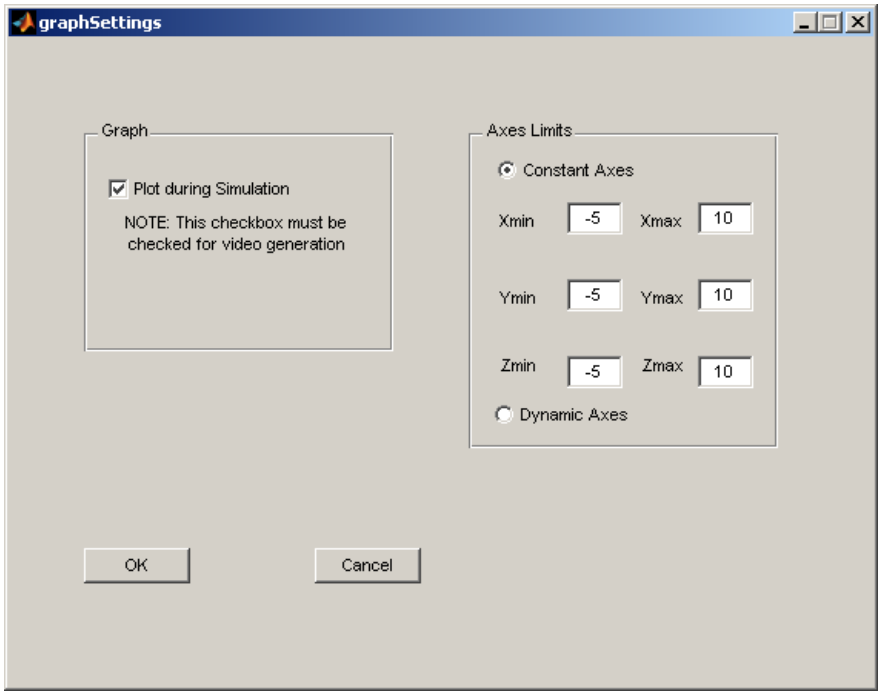

Figure A-13: MTD3D Graph Settings window (February 9, 2006)

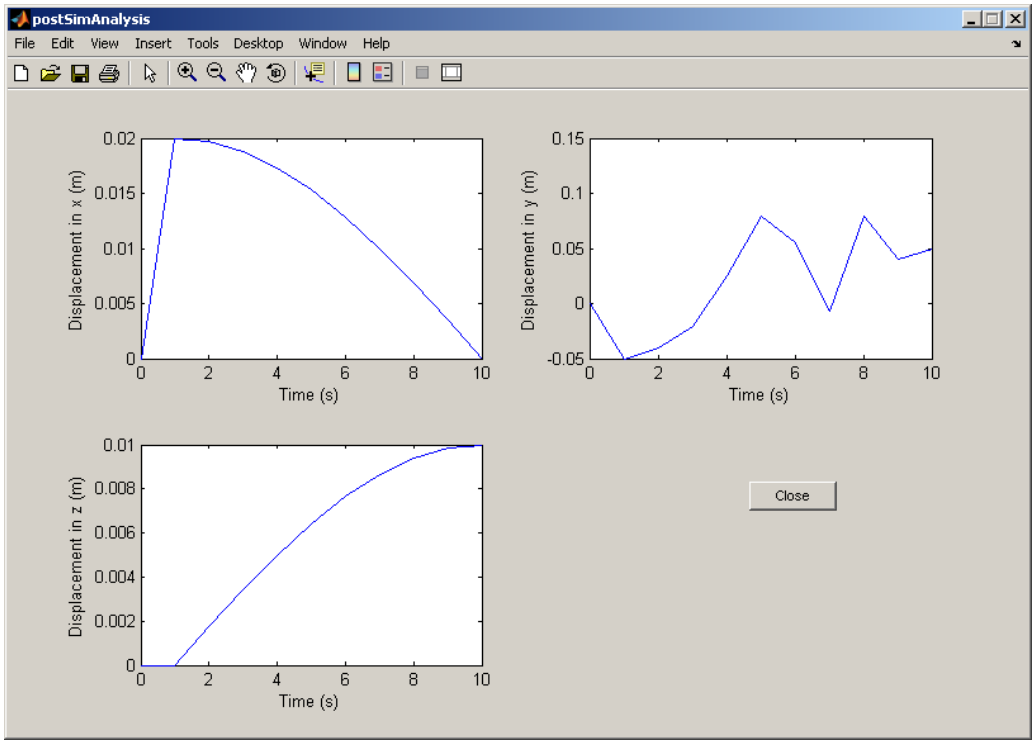

Figure A-14: MTD3D Post-Simulation Processing window (February 9, 2006)

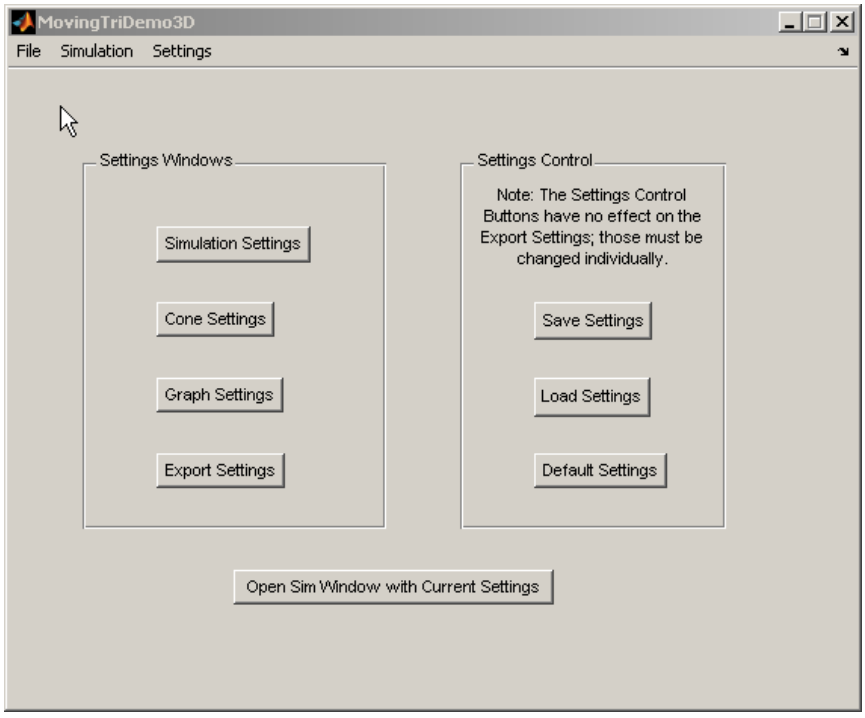

Figure A-15: MovingTriDemo3D opening window (February 13, 2006)

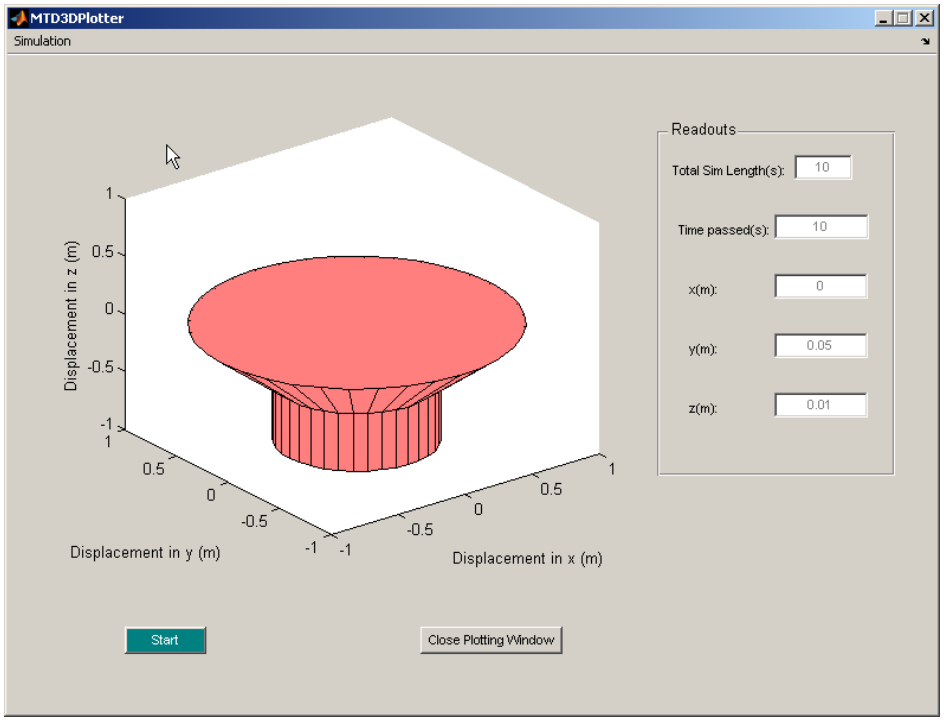

Figure A-16: MTD3D Plotter window (February 13, 2006)

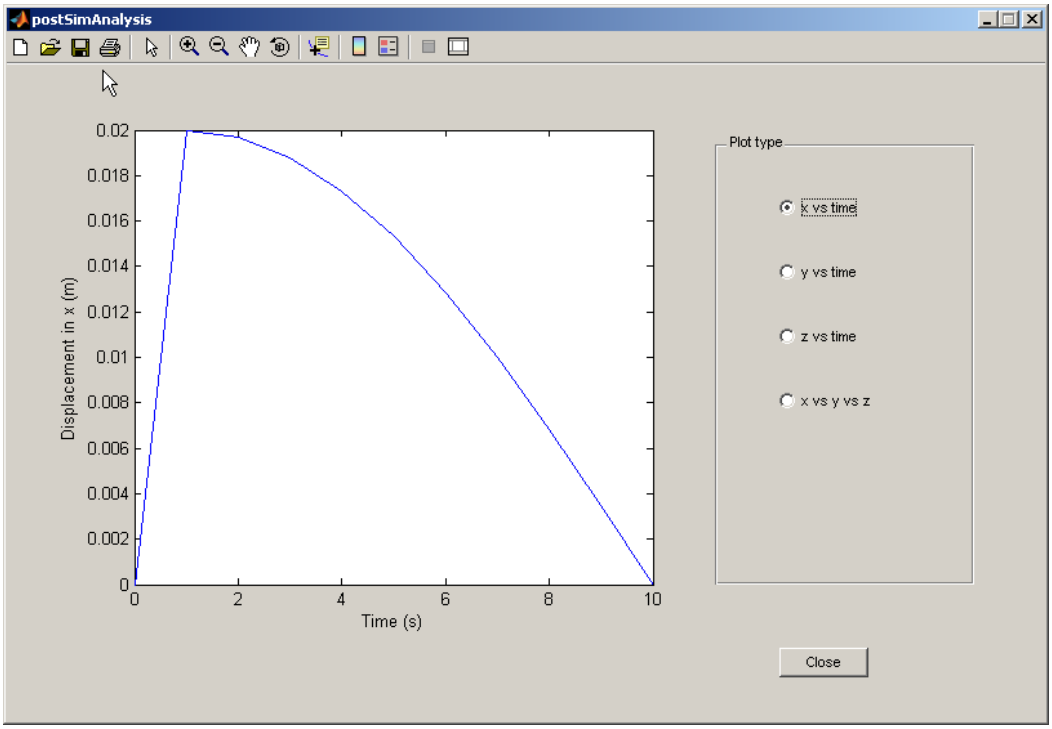

Figure A-17: MTD3D Post-Simulation Processing window (February 13, 2006)

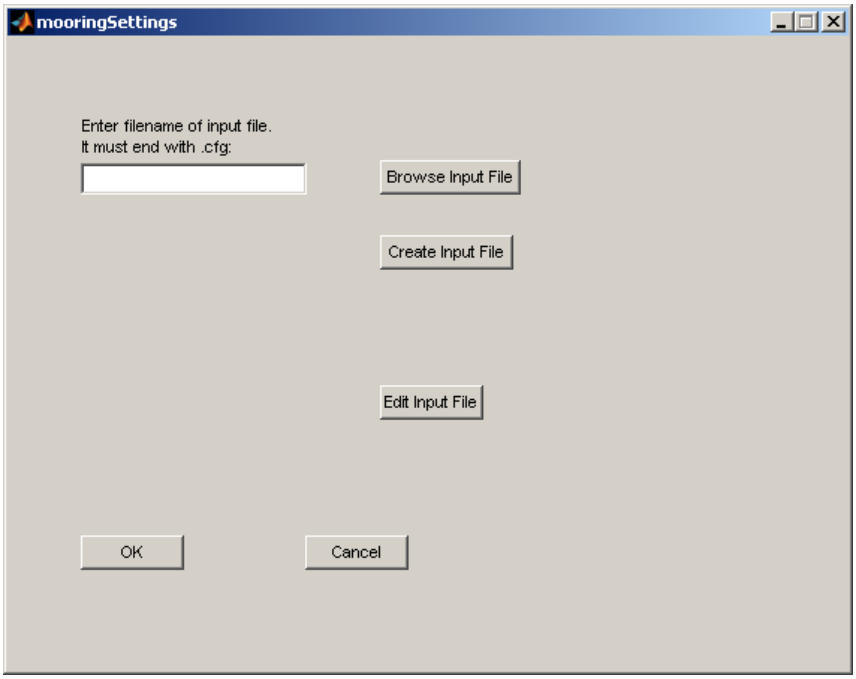

Figure A-18: MTD3D Mooring Settings window (February 13, 2006)

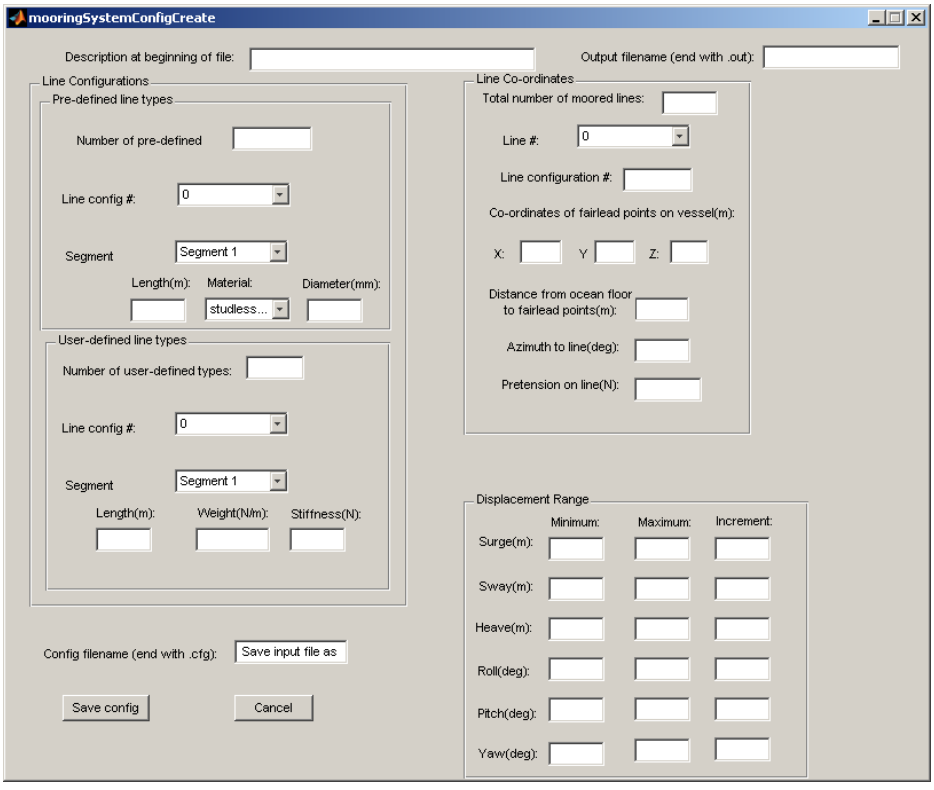

Figure A-19: MTD3D Mooring File Creation window (February 13, 2006)

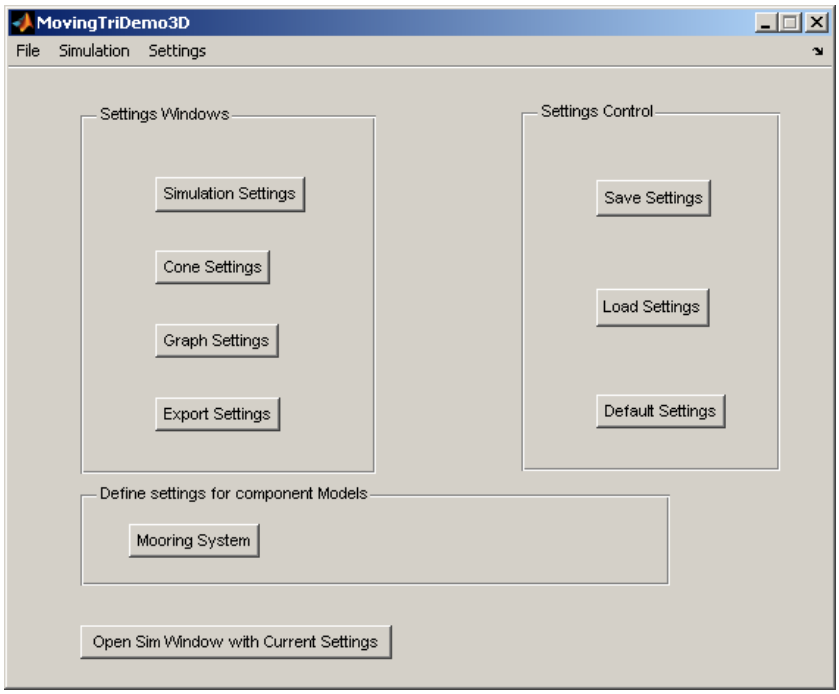

Figure A-20: MovingTriDemo3D opening window (March 10, 2006)

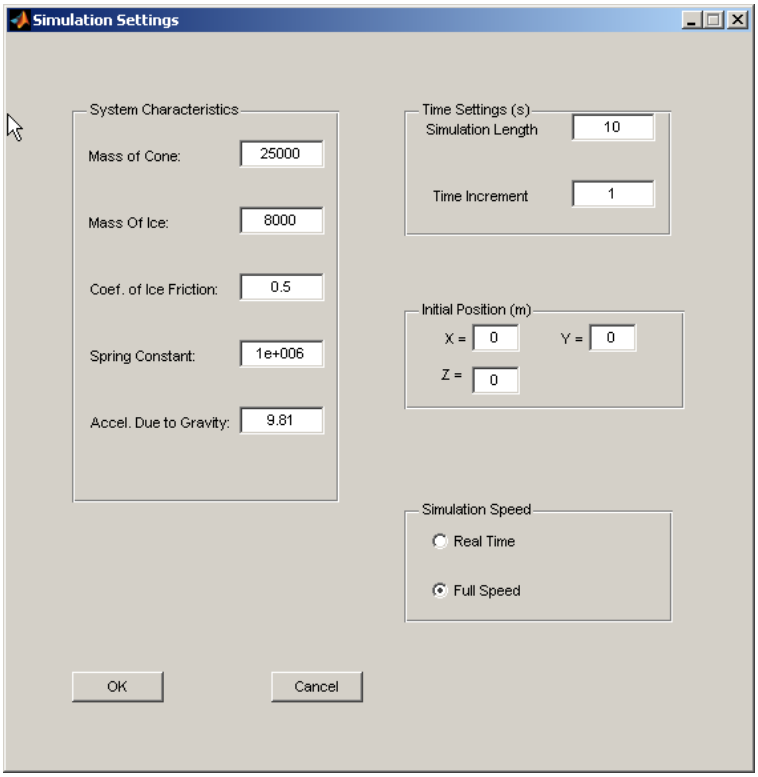

Figure A-21: MTD3D Simulation Settings window (March 10, 2006)

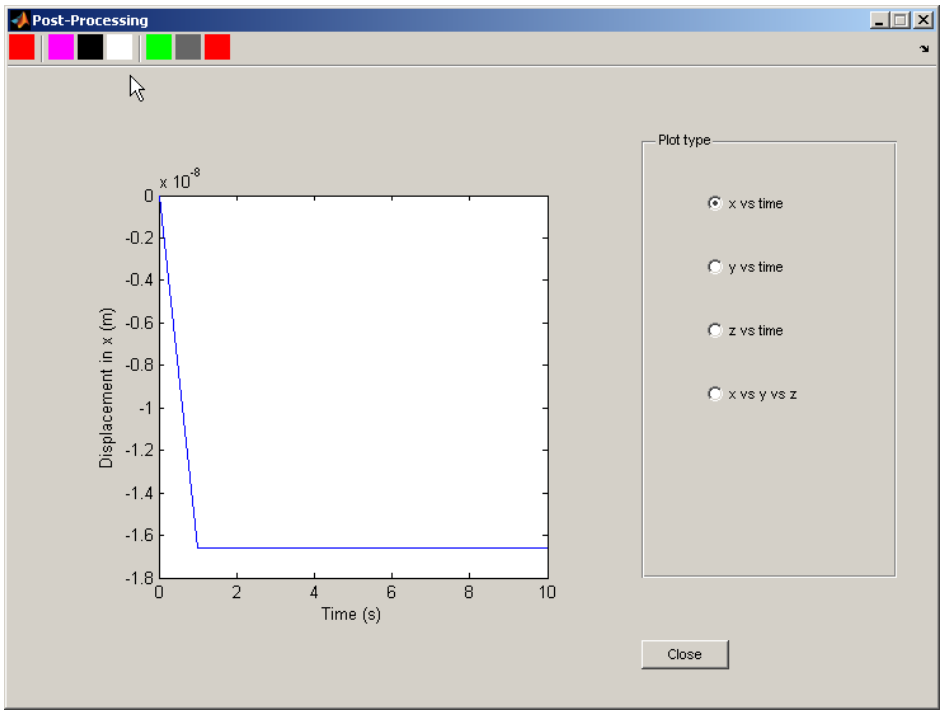

Figure A-22: MTD3D Post-Simulation Processing window (March 10, 2006)

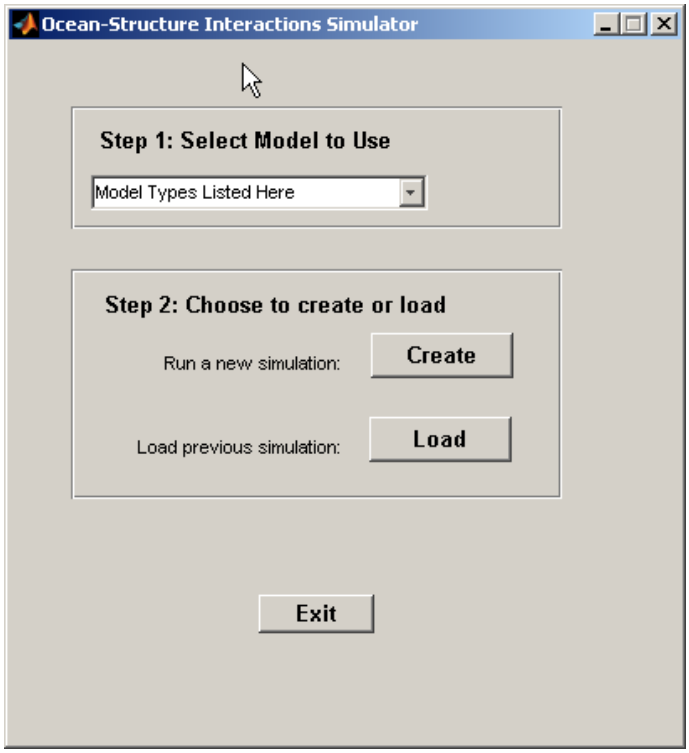

Figure A-23: OSIS Opening window (March 2, 2006)

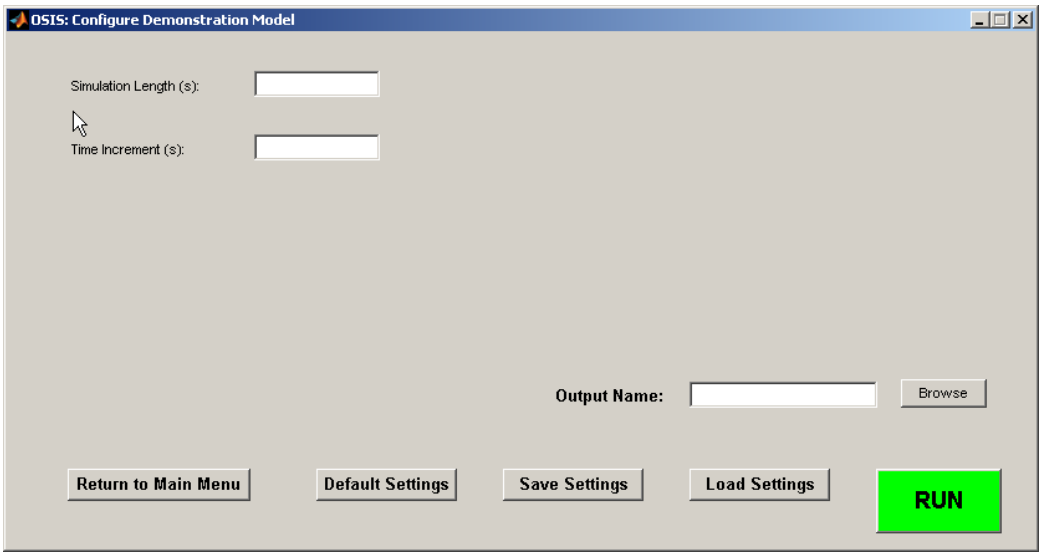

Figure A-24: OSIS Demonstration Configuration window (March 2, 2006)

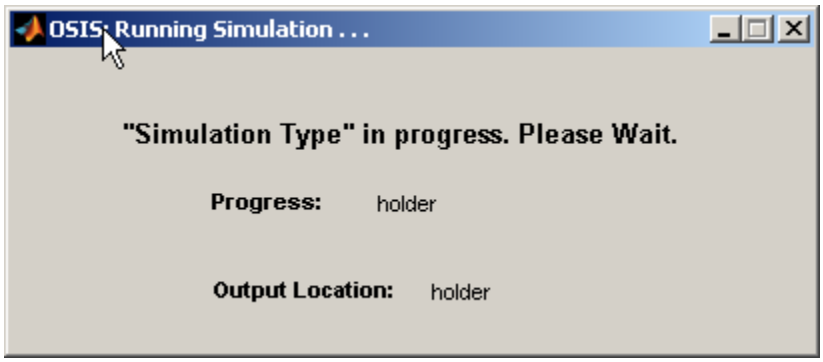

Figure A-25: OSIS Simulation progress window (March 2, 2006)

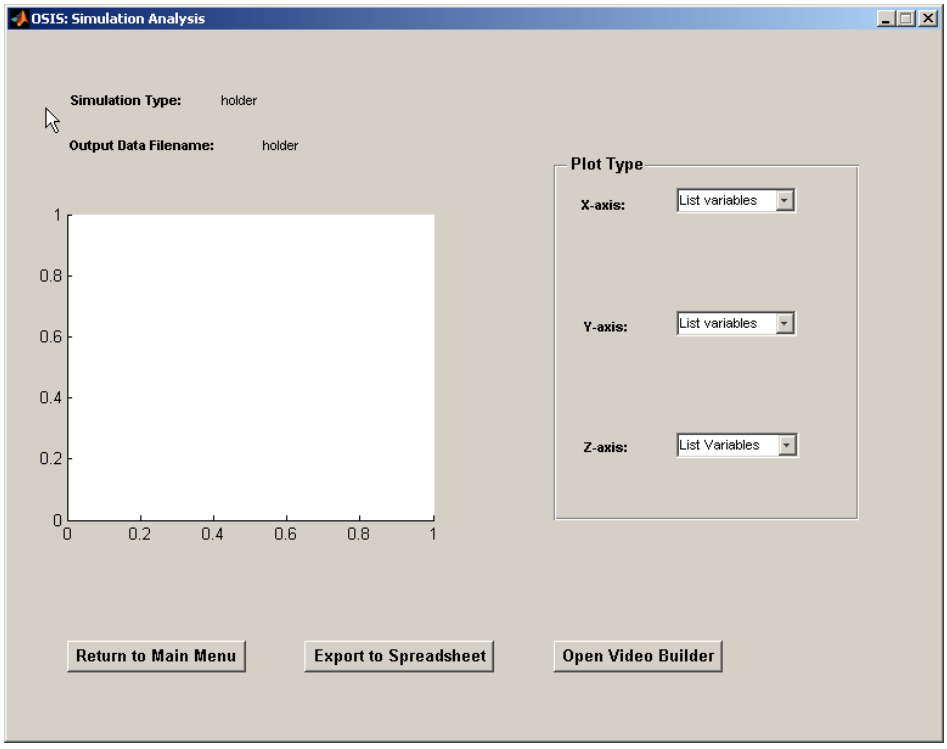

Figure A-26: OSIS Analysis window (March 2, 2006)

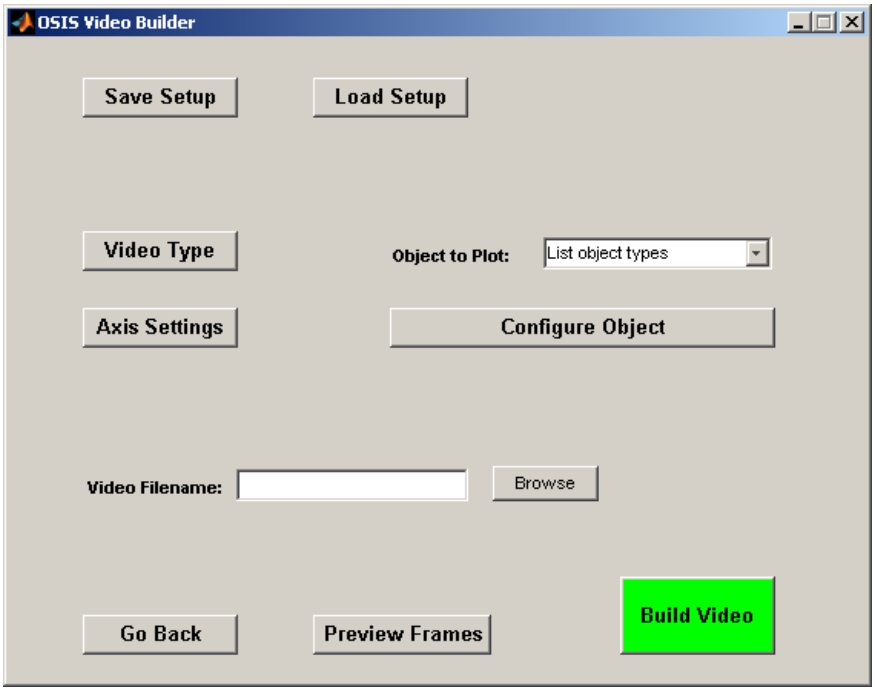

Figure A-27: OSIS Video Builder window (March 2, 2006)

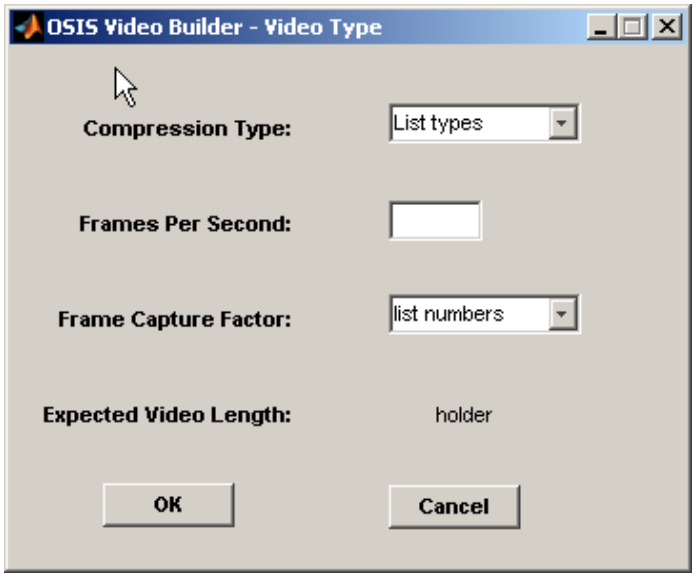

Figure A-28: OSIS Video Type window (March 2, 2006)

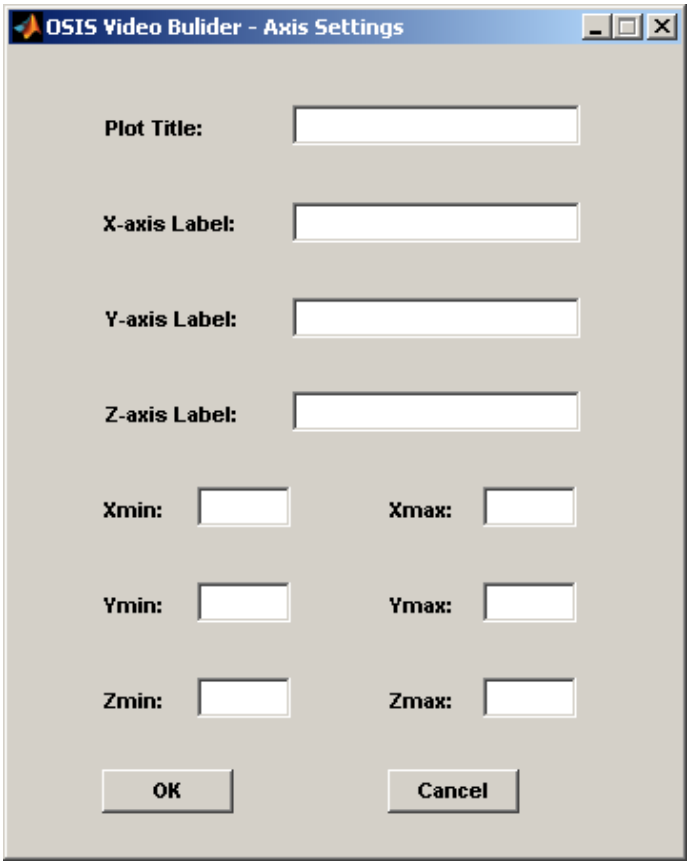

Figure A-29: OSIS Video Axis window (March 2, 2006)

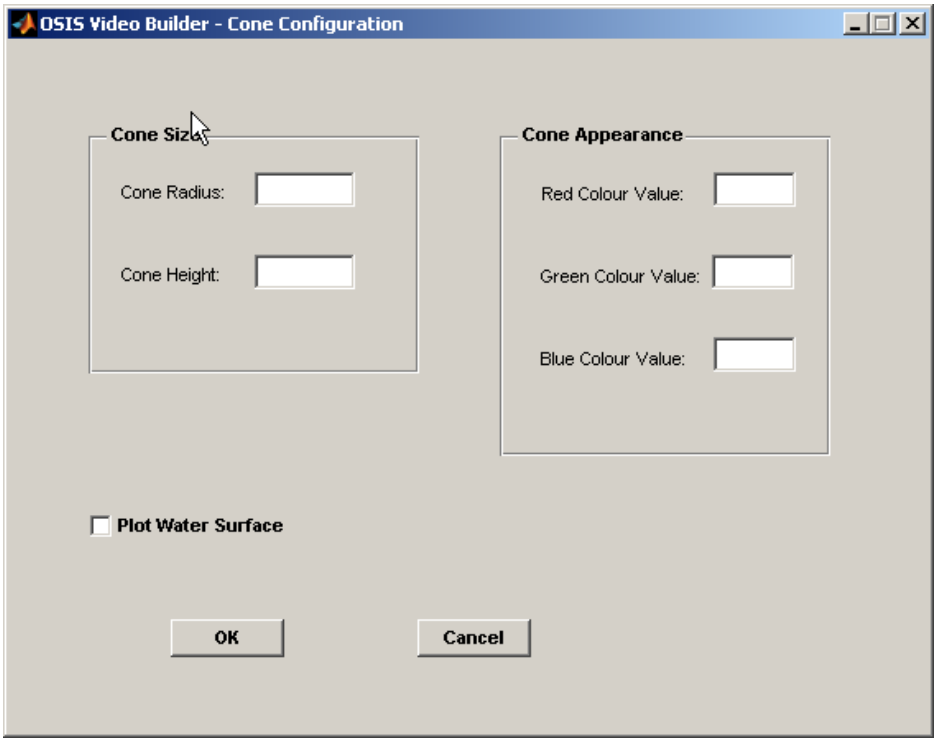

Figure A-30: OSIS Cone Video Configuration window (March 2, 2006)

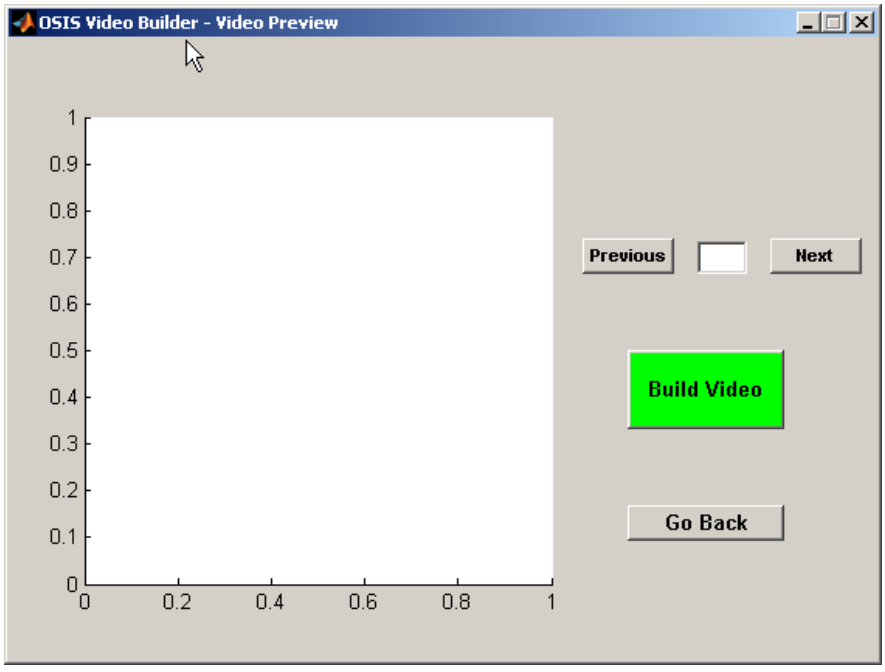

Figure A-31: OSIS Video Preview window (March 2, 2006)

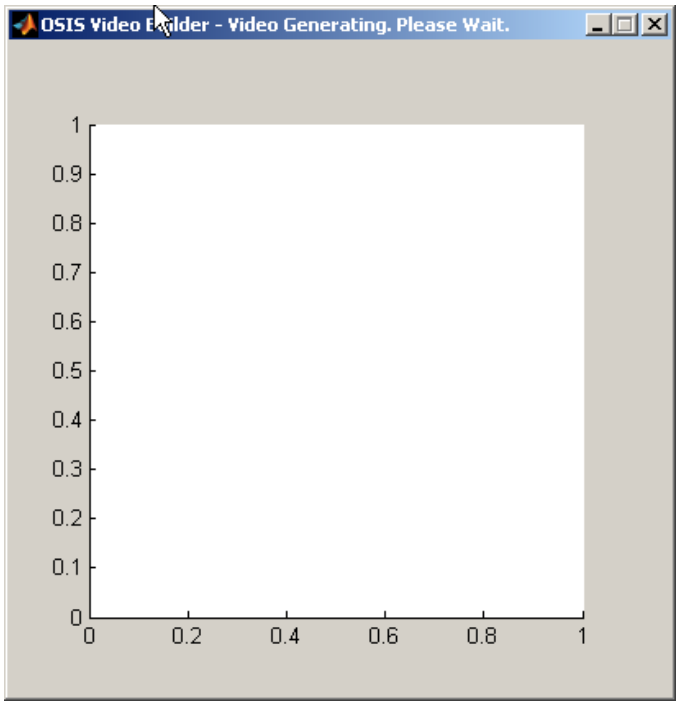

Figure A-32: OSIS Video Generating window (March 2, 2006)

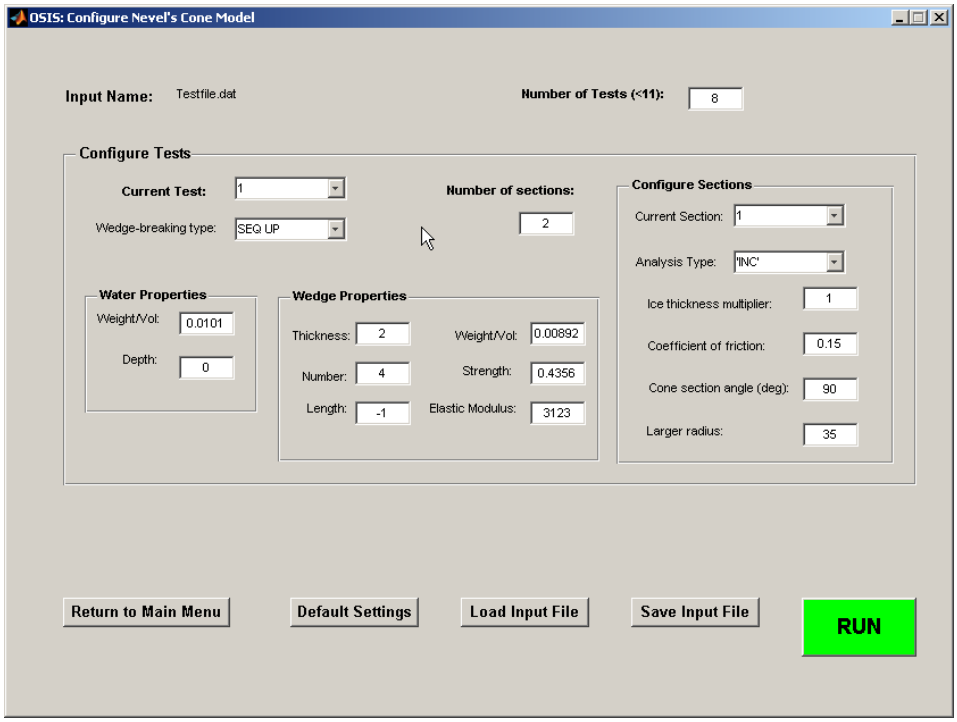

Figure A-33: OSIS Nevel Cone Configuration window (March 29, 2006)

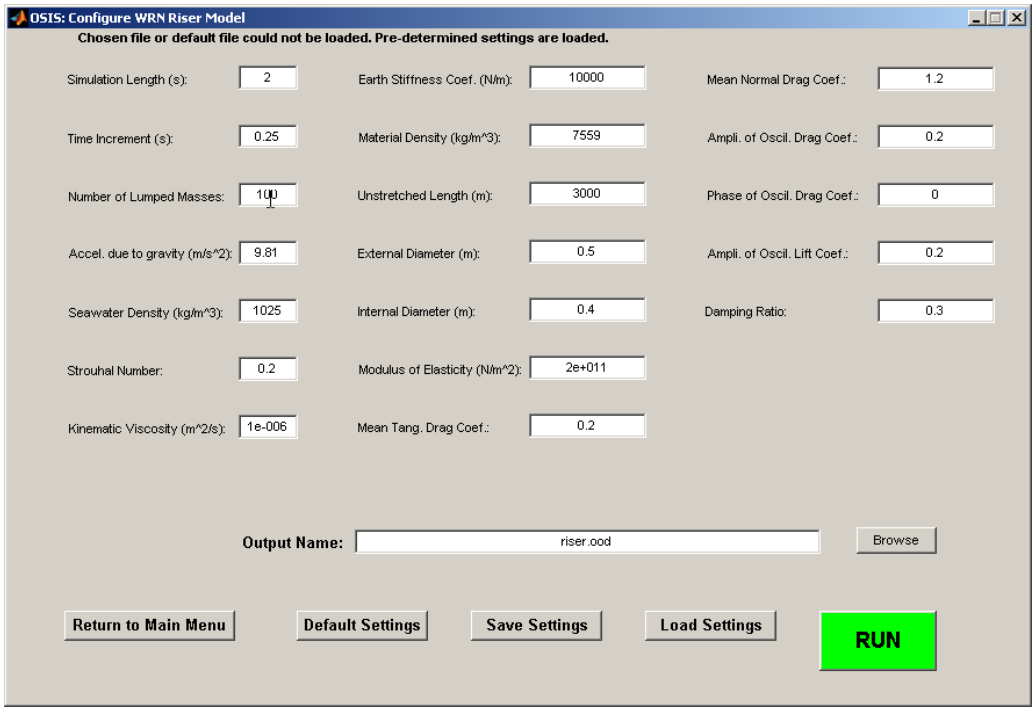

Figure A-34: OSIS Riser Model Configuration window (March 29, 2006)

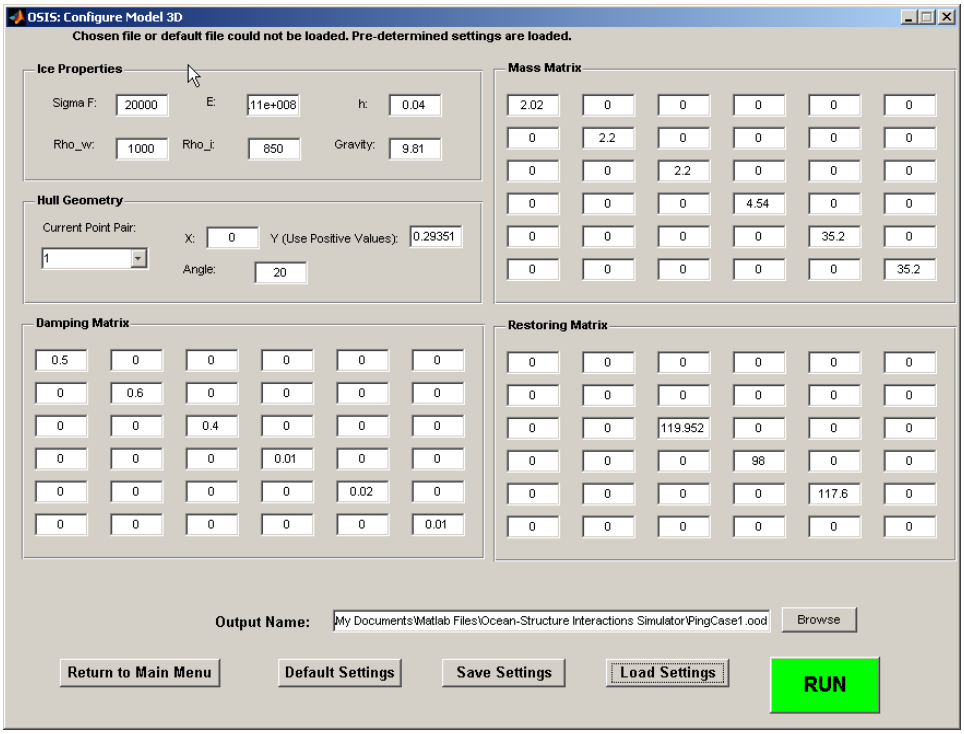

Figure A-35: OSIS Model3D Configuration window (March 29, 2006)

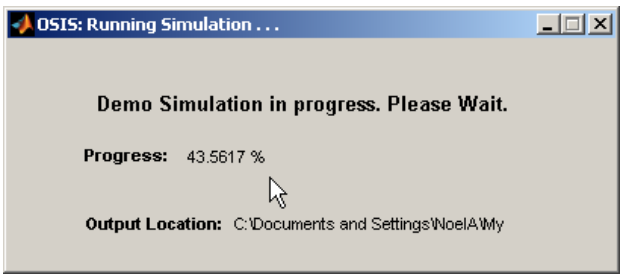

Figure A-36: OSIS Simulation Progress window (March 29, 2006)

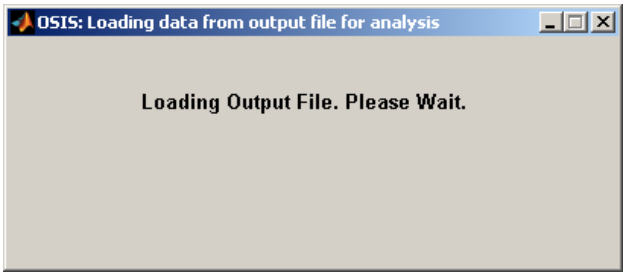

Figure A-37: OSIS Analysis Loading window (March 29, 2006)

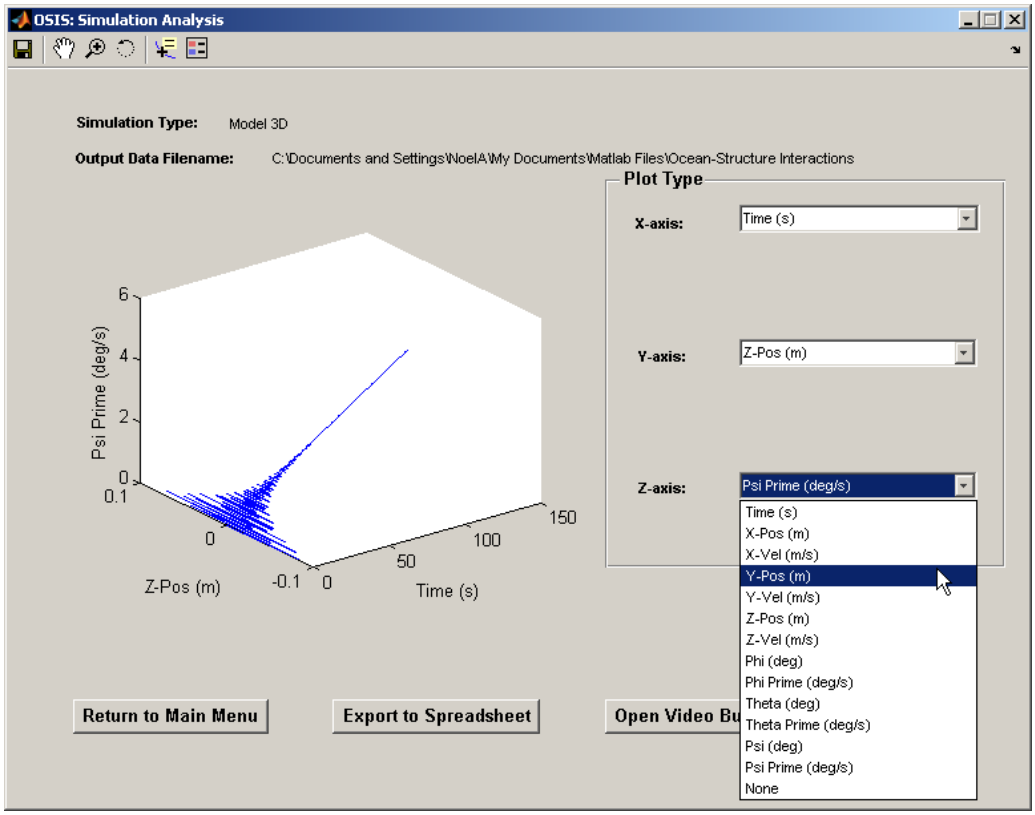

Figure A-38: OSIS Analysis window (March 29, 2006)

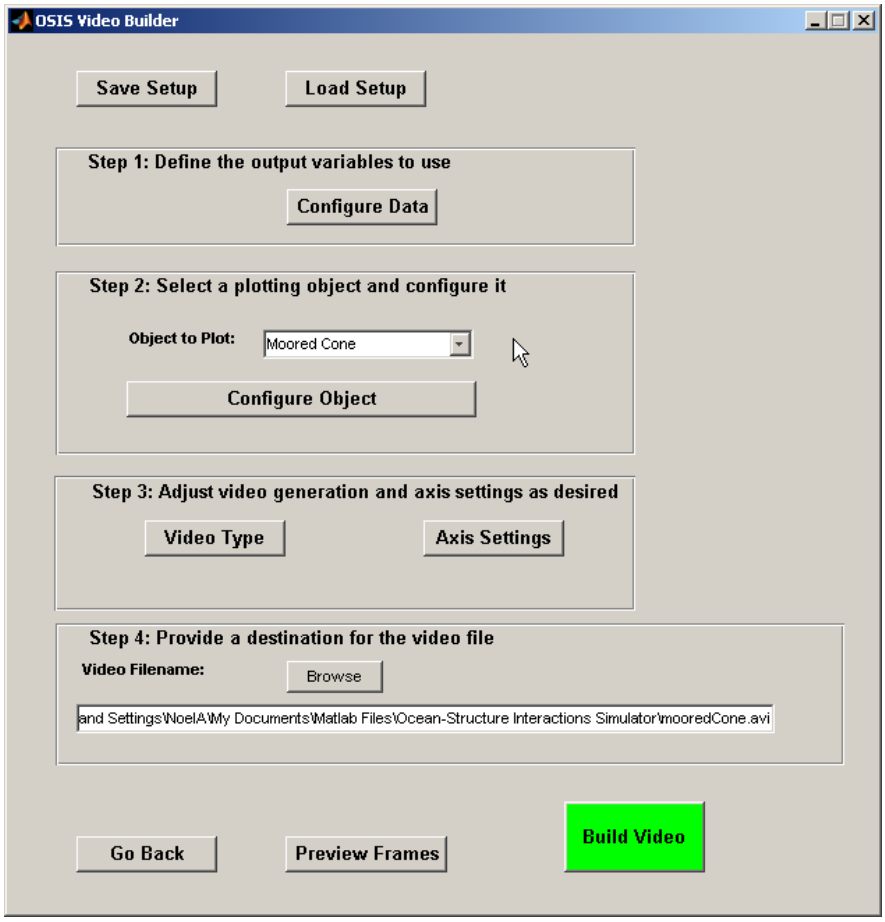

Figure A-39: OSIS Video Builder window (March 29, 2006)

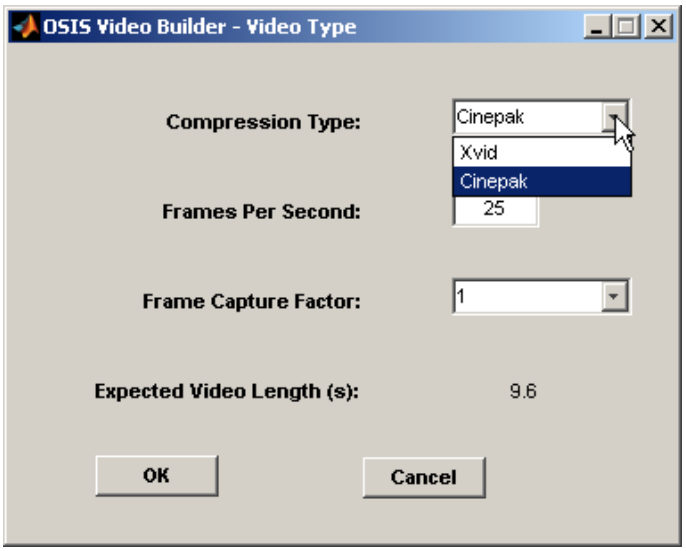

Figure A-40: OSIS Video Type window (March 29, 2006)

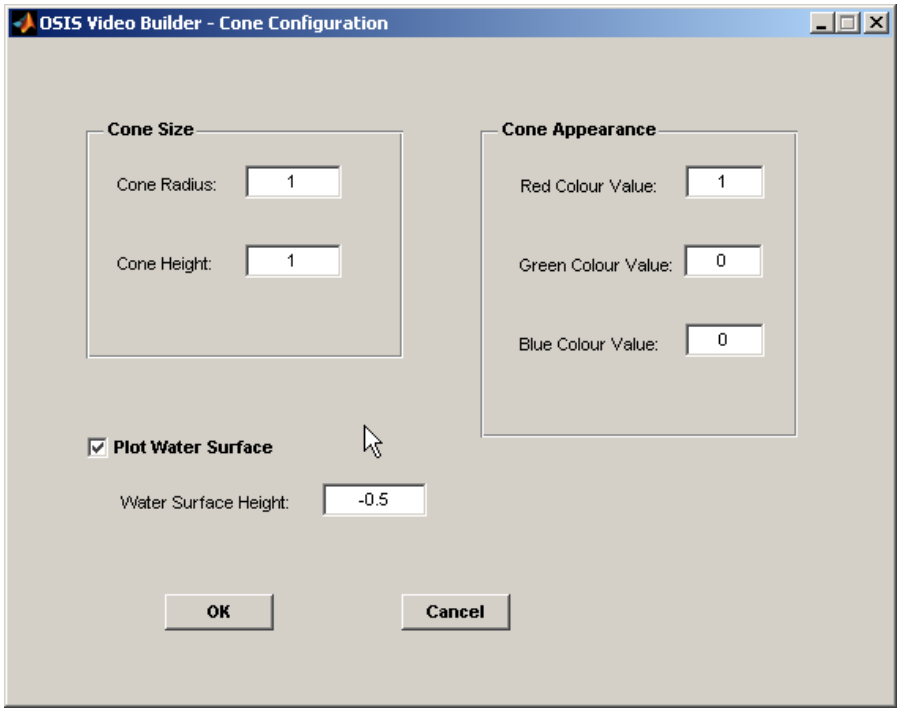

Figure A-41: OSIS Cone Video Configuration window (March 29, 2006)

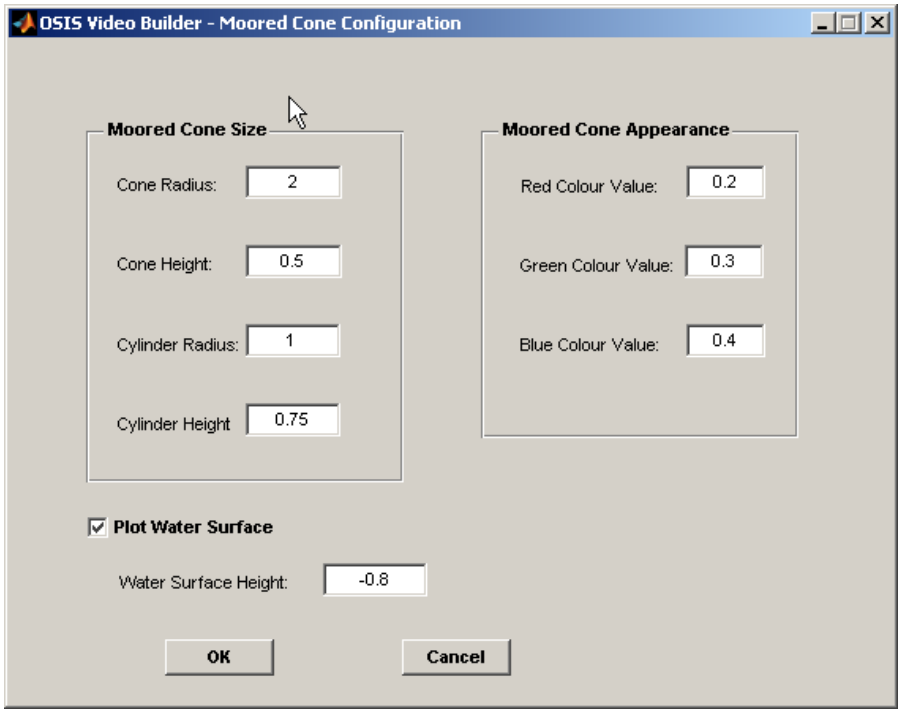

Figure A-42: OSIS Moored Cone Video Configuration window (March 29, 2006)

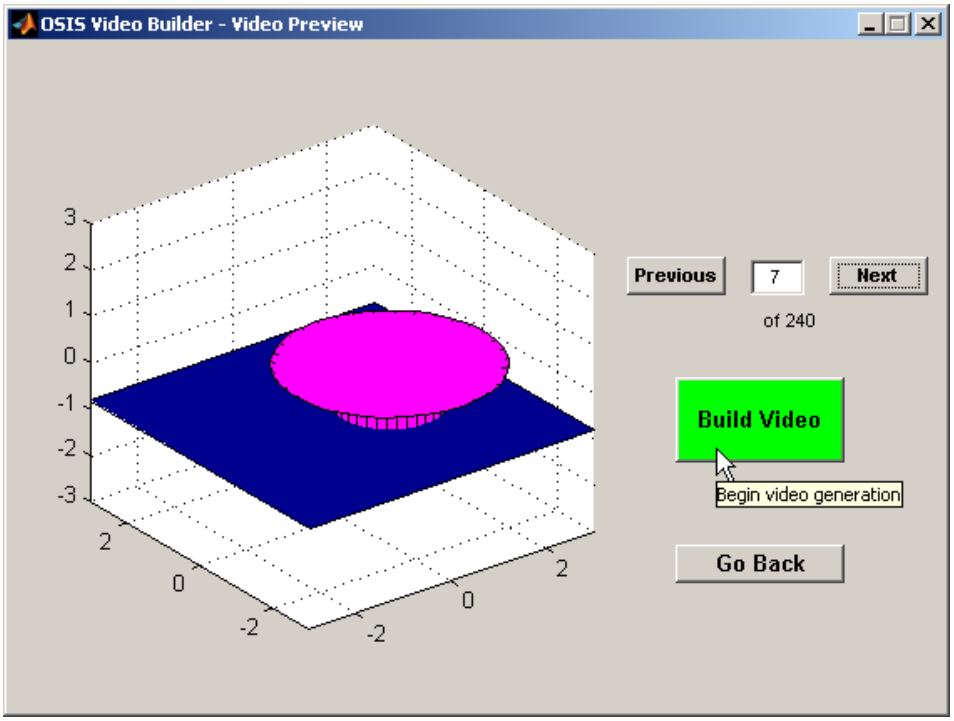

Figure A-43: OSIS Video Preview window (March 29, 2006)

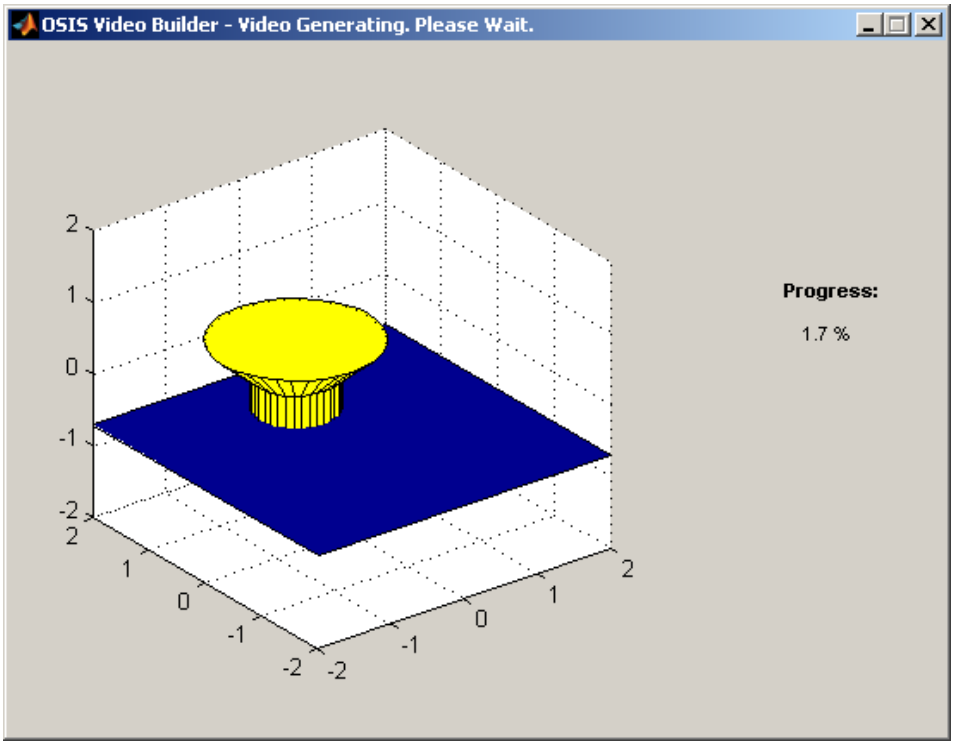

Figure A-44: OSIS Video Generation window (March 29, 2006)

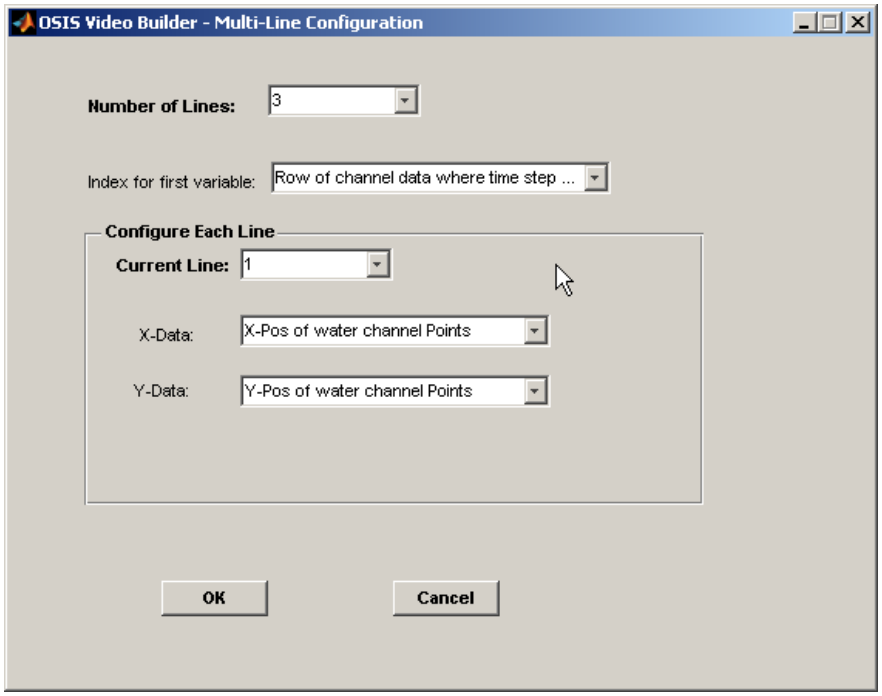

Figure A-45: OSIS Multi-Line Video Configuration window (April 7, 2006)

Appendix B

Matlab Programming and GUI Fundamentals

TABLE OF CONTENTS

MATLAB PROGRAMMING AND GUI FUNDAMENTALS

B.1.0 INTRODUCTION

The Ocean-Structure Interactions Simulator (OSIS) interface was designed completely in Matlab, a programming language developed by The MathWorks. Matlab version 7 (Release 14) is used, and this version or later is required to properly modify and use OSIS.

Matlab is a high-level programming language, unlike low-level languages such as C++ or Java. It is short for "Matrix Laboratory", whereby all variables are stored in some form of matrix. Many functions are already included in this language, and there are many aspects that are automated and thus do not need to be considered by the programmer.

This document serves as an introduction of how to code in Matlab, focusing on capabilities that were and will continue to be integral to the development of OSIS. This document is not meant to be a complete reference guide; Matlab comes with extensive documentation, also available on-line via www.themathworks.com. It is strongly recommended to go through the introductory components of the help files as well as consult other Matlab references when necessary.

Section B.2.0 outlines the basics of Matlab and how it works as a programming language. An overview of graphical user interfaces (GUIs) and how they are coded is given in Section B.3.0. Section B.4.0 describes plotting and exporting plot information from Matlab, and Section B.5.0 provides guidelines for reading from and writing to text files. The process of compiling Fortran code to be used by Matlab is discussed in Section B.6.0. Finally, miscellaneous topics are briefly noted in Section B.7.0.

B.2.0 THE MATLAB PROGRAMMING LANGUAGE

B.2.1 Introductory Notes

Matlab has a working directory that it uses, while running, to access scripts, saved data, and other files. When Matlab is opened change the working directory to a folder that can be written to.

Semi-colons are not necessary at the end of every line of code, but they prevent the result from being displayed in the command window. They reduce a lot of clutter when running programs, or could be omitted in specific locations to look for errors.

The percent symbol, '%', is used for comment lines. Typing '%{' on a single line begins a comment section, and typing '%}' on a single line closes the comment section.

If a command is too long to reasonably write on one line, use '…' at the end of the line and continue with the command on the following line.

B.2.2 Creating Variables

All data in Matlab is stored in some form of matrix that can be read from or written to. There is in fact only one true data type: the Matlab array. Any continuous collection of alphanumeric characters can be a variable name, unless they are a reserved name (ex: 'if', 'for', 'return', 'plot', etc.). Creating a single variable is very straightforward; type

```
x = 4;
```

in the command prompt and the variable 'x' is created storing a value of 4. The semi-colon at the end prevents the result from being immediately read back to the user. To create a 2x2 matrix of data, simply type

```
x = [1, 2; 3, 4];
```

The matrix 'x' has a first row of [1,2] and a second row of [3,4]. When entering a matrix, semi-colons separate rows. The commas separate values but are not necessary. It is sufficient to leave blank spaces in between the values of a row.

To append to matrices, include the variable name as an input into the matrix, then add the desired values. In the case of the previous 'x', type

```
x = [x; 5 6];
```

to add the row vector [5,6] as the third row in 'x'. Note that whatever is appended must have the correct number of values to match what is already in the 'x' matrix.

Strings (i.e. words or sentences) are built into the Matlab language, and can be created as easily as variables. Type

```
y = 'good morning';
```

to create a string with the words "good morning." Enclose a string with single quotation marks.

One powerful option is to combine strings and numbers under a single variable name in matrix form. This kind of matrix is referred to as a cell array. One cannot normally include strings as entries in a regular matrix unless they are all of identical length, but cell arrays make this possible. In a sense, cell arrays are individual matrices that are stored together like matrices. For example, type

```
z = {[1] 'hello' [45]; 'good' [3] 'morning'};
```

to create a 2x3 matrix 'x' with three strings and three numeric values. All of the values are considered to be arrays. Notice that each entry resembles the form used if it were created externally. One can easily change [1] to [1 2; 3 4], nesting a matrix inside a cell of the cell array.

Multi-line strings can be made using matrices, cell arrays and the same appending method for numbers. For example, if the 'greeting' variable contains the word 'good', and 'morning' is to be added as a second line, type

```
greeting = [greeting; {'morning'}];
```

and one word will be used on each line. Numbers can be added as well, either as strings (ex: '2') or as actual numbers since both types can be combined when using cell arrays, although only actual numbers can be used for calculations without performing a conversion.

An alternative method for storing multiple forms of data together is via structures. Structures are similar to cell arrays, but instead of placing data together in a numerical fashion (like in a matrix) they are stored with field names. Type

```
food.value1 = 2;
food.friday = 'hello';
food.monitor = [3; 5; 6];
```

to create the structure 'food' with three fields. What is stored in these fields is independent of the data in other fields, just like with cell arrays.

Additionally, structures can be nested in cell arrays. This technique is very powerful when cell arrays are required for passing data but the programmer does not want to keep track of where each item is located.

B.2.3 Reading Variables

Surprisingly, reading data in Matlab can sometimes be more complex than creating it, especially when dealing with nested structures and cell arrays. To view a variable or matrix as soon as it is created, omit the semi-colon at the end of the line. To view a variable or matrix after it has been created, type the name of the variable itself. If the variable is a nested structure or cell array, it is likely not all of the exact data will appear on-screen unless explicitly referenced.

Indexing is required to access a particular element in a matrix, structure, or cell array. An element can itself be a matrix, structure, or cell array, so further indexing may be required to extract a desired value. Indexing can also be used to write to particular elements of a matrix, structure, or cell array, giving immediate access to portion of created data.

Consider the matrix x = [5, 6; 7, 8]. To read the 7, you can type

```
x(2,1)
```

or

```
x(3)
```

since 7 is both the first value in the second row as well as the third value overall. Reading multiple rows or columns at once requires the use of the colon, ':'. To read the 7 and 8 together, type

```
x(2,1:2)
```

or

```
x(2,:)
```

and the matrix [7 8] will be available. The colon can be used by itself to denote an entire dimension, in this case the second row, or in between two numbers, in this case indexing all elements from the first to the second in the second row. This concept easily extends to three or more dimensions.

For a structure, simply typing the structure name followed by a period and then the field name will display the data in the respective field. In the case of x.monitor = [3; 5; 6], typing

```
x.monitor
```

will give the entire field, while

```
x.monitor(2:3)
```

will give a column vector [5;6].

Cell arrays are similarly indexed using curly brackets. When data is nested extra brackets are required. First, the case of a matrix in a cell will be considered.

Consider the cell array x = {[4] [34.7] [19 25]; 'telephone' [13; 1] [6 8; 4 11]}. What if a user wanted to write over the 8? In order to do this, curly brackets must be used to reference the cell, and then round brackets must be used to reference the matrix. So, type

```
x{2, 3}(1,2)
```

or

```
x{6}(2)
```

to access the desired element. Note that column/row indexing is used more commonly than total element indexing.

For the case of a structure nested in a cell array, it will be simplest for now to consider a single cell containing a structure with single-argument fields. First, the structure will be defined:

```
x.red = false;
x.blue = 2;
x.green = 'hello';
```

The 'x' structure contains three fields, which will now be placed into cell array 'y':

```
y{1} = x;

y{1}.green
```

The string 'hello' will appear on-screen. Elements in structures in cell arrays are indexed first by the cell array using curly brackets and then using the field name.

B.2.4 Control Statements

Matlab contains most expected control statements, including 'if', 'else', 'for', 'while', and 'switch'. All of these are used at some point within the code of OSIS. Unlike in other languages, no curly brackets are needed to contain these statements; rather, the keyword 'end' signifies when the control ceases.

Only the 'switch' will be demonstrated here, as it is perhaps the least common of the given control types. Consider a variable named 'doNow' that could contain one of a few different strings. The 'switch' control checks the string against the given cases to determine what commands to follow. The possible cases and results are given:

```
switch doNow
     case 'continue'
          % Do nothing here. The percent symbol is for comments
     case 'pause'
          pause % The program waits for a button to be pressed
     case 'calculate'
          x = x + 3; % Three is added to the value of x
     case 'quit'
          close % Running code is closed
end
```

B.2.5 M-Files and Functions

None of the actual code used to run OSIS is entered into Matlab's command window. Rather, routines have been developed on external files that Matlab can read and execute. These files are called m-files, and have a '.m' extension. Matlab has an m-file editor which opens when it loads an m-file or when a user clicks to begin a new m-file. Running the code contained in an m-file is as straightforward as pressing F5 from within the editor, or typing the name of the file (without the extension) in the Matlab command window. Ensure a given file is in the Matlab working directory for it to open. If the file is not in the directory but F5 is pressed while editing, a prompt will appear to change the directory.

The m-file editor makes it easy to view and edit m-files. Comment sections are coloured in green and keywords are coloured in blue. Strings are coloured purple, but are coloured red if a quotation mark is missing.

Usually, an m-file contains a function or collection of functions that are executed when called by other functions. For example, typing

```
function answer = sample(x, y)
```

at the top of an m-file creates a function named 'sample' that requires two inputs, 'x' and 'y', and outputs a variable called 'answer'. These arguments can be matrices, structures, cell arrays, or simple variables, so long as they are used consistently, i.e. don't pass 'sample' an 'x' that is a string if it is used as a numerical matrix. The 'answer' output must be used somewhere in the function or an error will occur. If a function is created later in the file it is considered a sub-function and can only be called by functions within the same file. There is no need to worry about what to type in order to end a function; defining another function automatically terminates the code for the previous function.

For very large m-files it is important to be able to access particular sub-functions as quickly as possible. There is a "Show Functions" button in the editor toolbar, with a picture of a blue f, to go to any sub-function contained in the current m-file.

An m-file without a function at the top is considered a script, since there are no arguments to pass in or out. Scripts are called using the filename without the '.m' extension. Since Matlab interface code creates m-files with leading functions by default, scripts are not used.

B.2.6 Handles, Getting, and Setting

Any object created in Matlab will have a series of properties available for reading from or writing to. By far the most important property is the handle, which can be considered as the proper individual name of the object. No two objects existing at the same time will have the same handle. Hence, the handle is used to reference a specific object when its properties are being accessed.

The handle types most relevant to OSIS are those for figures, axes, graphical user interface (GUI) objects, menus, and toolbars. The handles for many of these objects are stored in a particular format that is discussed in greater detail in Section B.3.0. In general, the handle of an object can be stored at the moment it is created. For example, typing

```
h = plot(x,y);
```

will create a figure plotting x vs. y (assuming these are data with values that the programmer has created) and will also store the handle of the figure to the variable 'h'. Now, using the "get" and "set" commands, we can change or retrieve information about the figure. The formats for using get and set are the same, except set requires additional arguments for property values. Typing

```
width = get(h, 'LineWidth');

set(h, 'Title', 'First Test', 'Color', 'blue')
```

will return the line width of the plot to store in the variable 'width', create a title for the figure, and will colour the data plot blue. There is no limit on the number of property-value pairs that can be included in a call to the "set" function; in this case two pairs were used. Note that the "get" function requires a semi-colon to suppress displaying the result in the command window, while the "set" function does not. To view a complete listing of all properties available for an object, type

```
get(h)
```

and the list will appear in the command window. The help files also contain listings of all properties available for particular objects. Similarly, type

```
set(h)
```

and a list of user-settable properties will appear without the read-only ones. This works even though "set" does not normally output results in the command window.

B.3.0 GUIS IN MATLAB

Matlab is a versatile development environment for creating Graphical User Interfaces (GUIs). A GUI is a set of windows and controls that a user can see and interact with, avoiding the need to enter memorized commands in a command window. With one of the objectives of OSIS being to market it for commercial use, GUIs are important to make the models working in the background accessible to the user without the user needing to know exactly how they work.

The Graphical User Interface Development Environment (GUIDE) uses the advantages of GUIs to build GUIs. While it is possible to create a GUI via lines of code in functions, GUIDE is Matlab's interface for GUI design. The layout of the GUI, consisting of panels, buttons, displays, and other objects, can be visually placed and edited. Much of the code for running the GUI is automatically generated by GUIDE, so the programmer can focus their time on how the controls are set up and how they interact. This is preferable to using trial and error to determine an appropriate pixel offset for a collection of buttons to be lined up properly, for example.

When a user adds objects to a window in GUIDE, Matlab generates a '.fig' file which saves the layout and properties of the window. The code for how the GUI is used is saved to an '.m' file that the user edits to give commands to the functions of the controls. For example, if pushing a particular button is supposed to run another m-file, the call to that file must be included in the button's "Callback" function. More details on "Callback" functions are to follow.

To open GUIDE, type 'guide' in the Matlab command window or press the GUIDE button in the Matlab toolbar. GUIDE has a few template GUIs, but they are not recommended for use because they will not compile into external executables. This constrains a GUI's marketability. Instead, open a blank template when starting a new GUI.

B.3.1 GUI Objects

GUIDE has many pre-defined objects for use that are familiar to computer users. Those used in OSIS, and thus those that will be described here, are edit boxes, popup menus, sliders, push buttons, radio buttons, check boxes, static text, panels, button groups, and axes. To place any of these into a GUI, click its respective button in the object toolbar and drag the object onto an available region of the GUI. These objects can be copied, pasted, and resized as expected. They also share a number of properties that are necessary to consider. The properties window for an object is opened from the right-click menu of the object, clicking on "Property Inspector". The Matlab help files list all of the properties for each control, including descriptions and defaults. Note that these properties can be manipulated via the set and get commands, described in Section B.2.6.

The most commonly used properties for GUI objects are 'String' and 'Tag'. The 'String' entry is what a user will see when they view the object, such as the label for a push button or the description of a checkbox. If more text is needed for a control than what the single-line String can provide, static text can be added where needed. The 'Tag' entry is what will be used to

distinguish a particular object from another. There will always be a default name provided for a tag, such as 'pushbutton1', but it is strongly recommended to use tag names that describe the object more specifically, such as 'ok_button'. Tags must be one continuous collection of characters, so underscores are often used to distinguish multiple words.

The 'TooltipString' property is another that should not be overlooked when developing a GUI for others to use. Any string entered will appear if a user holds the mouse cursor over the respective control. This is useful for providing extra information that may not be obvious from the control label (if there is one) or nearby static text.

Without any code added by the programmer, a GUI object would not normally serve a practical purpose. The functions written for GUI objects are called "Callbacks", and are called when the control is changed (ex: button is pressed, box is checked, slider is moved, etc.). To see a list of callbacks available for a particular object, right-click on the object and select "View Callbacks". Any button that is a control will have a callback entitled "Callback", along with a "DeleteFcn", "CreateFcn", etc. All of the callback types, if the programmer modifies them, are defined in the m-file for the GUI, keeping them in a single file for quick access. Matlab creates default names for the functions, and are always TAG_CALLBACKTYPE. So if the OK button had a callback function to be called when it is pressed, it would be named ok_button_Callback if the tag for the button was the one named in the previous paragraph. The button's function called when it is created would be named ok_button_CreateFcn.

Primarily, only the explicit "Callback" function is used when programming GUIs, so the following sections describe the code typical for Callbacks of the different objects. For the rest of this section, "callback" denotes the actual "Callback" function, and not the category of available functions.

Note that in general, 'hObject' is the variable used to store the handle of the object whose callback is being accessed. Handles are introduced in Section B.2.6.

B.3.1.1 Edit boxes

Edit boxes can be used to display data or allow changes to be made in a program. The callback is called when a user presses enter while in the edit box or when a user changes the focus of the cursor away from the edit box. By default the entered values are considered as a string, but a quick conversion can read the value as a number. Typing

```
value = get(hObject, 'String');
```

or

```
value = str2double(get(hObject, 'String'));
```

in the callback will retrieve what is entered in the edit box as a string or a double number, respectively. The function "str2num" is available as an alternative to "str2double".

B.3.1.2 Popup menus

The listing seen in a popup menu is created as a single, multi-line string. This string can be created in the Property Inspector by clicking on the button next to the 'String' property. If any changes to the popup menu options need to be made from elsewhere in the GUI, a string can be created and stored via

```
set(h, 'String', newString)
```

where 'h' is the handle of the popup menu and 'newString' is the string to be entered. See Section B.2.2 to read about creating multi-line strings.

With regards to the callback, a check must be made to see which option has been selected from the popup menu. Even though a string is used to define what is written on each line, the value changed when the menu is used is the number of the line that was selected. Lines are numbered from 1 onwards. Therefore, depending on the context in which the popup menu is used, a check can be made to decide what course of action to take. Type

```
value = get(hObject, 'Value');

switch value
     case 1
          % Do this if the first entry is chosen
     case 2
          % Do this if the second entry is chosen
     case 3
          % Do this if the third entry is chosen
end
```

and a comparison will be made based on which of the three popup menu entries were selected. In some cases a 'switch' comparison may not be necessary since 'value' will be an integer that can be used directly in making some other function call or setting another variable.

B.3.1.3 Sliders

Sliders allow for a setting to be changed while letting the user know how close they are to the minimum or maximum acceptable values. They can be used in conjunction with edit boxes so that both change a variable and using one will update the display on the other. For now the slider by itself will be considered. From the "Property Inspector" the programmer can adjust the minimum and maximum possible values ('Min' and 'Max'). The 'SliderStep' property is a two-value vector to specify the percent change when clicking on the arrow button or clicking on the trough. In the callback, all that is needed is to retrieve the value. Type

```
x = get(hObject, 'Value');
```

to store the slider value to the 'x' variable.

When using a slider in conjunction with an edit box, the callback for the slider will need to include a set function call using the edit box handle to change the edit box's 'String' property, and the callback for the edit box will need to include a set function call using the slider handles to change the slider's 'Value' property. Note that using a numeric value for a string will automatically be converted into string format.

B.3.1.4 Checkboxes

Check boxes have two possible states: checked and unchecked. The callback for a checkbox should compare the 'Value' property to the 'Max' value. The following example shows this:

```
if (get(hObject, 'Value') == get(hObject, 'Max'))
     % Box is checked so run this code
else
     % Box is unchecked so run this code
end
```

Typically, the callback for a checkbox would take a Boolean variable and make it 'true' or 'false' (without the quotation marks). Note the extra brackets around the "get" functions, which are needed for the logical '==' (is equal?) statement to be used in context with the 'if'.

B.3.1.5 Static text and panels

These objects usually do not need to be coded at all, since they would not normally change during the course of a running program. The tags for these objects can hence be left as their defaults. Static text is useful for giving permanent instructions, such as the setting to be entered in an edit box. Panels can visually combine similar controls, such as the buttons to change different categories of settings or the displays of various simulation variables. The string property of both objects defines what a user will see.

B.3.1.6 Radio buttons via button groups

Button groups appear identical to panels, except that as GUI objects they control any toggle buttons or radio buttons that are placed onto them. They are especially useful for radio buttons. Any radio button or toggle button placed on a button panel will never have its own callback called; instead, the "SelectionChangeFcn" of the actual panel must be coded. This type of callback is called when a selection is made via a radio or toggle button on the respective button panel. The 'SelectedObject' property stores the handle of the selected button, so the tag can be checked to know which of the buttons was selected.

```
selection = get(hObject, 'SelectedObject'); % Get handle
```

```
switch get(selection, 'Tag')  % Use handle to find tag
     case 'radiobutton1'
          % Do this when the first radio button is selected
     case 'radiobutton2'
          % Do this when the second radio button is selected
end
```

This process simplifies the number of callbacks to be coded, since only one is needed per relevant collection of radio buttons. Note that only one radio button on a button panel can be selected at a time.

B.3.1.7 Push and toggle buttons

Conveniently, push buttons are either not pressed or they are, therefore the callback, which is only called when the button is pressed, does not need to do any kind of check to see the state the button is in. The code written for push buttons will only be executed when the button is pressed.

For toggle buttons, a check must be made on the state of the button, which is stored in the 'Value' property. Typing

```
state = get(hObject, 'Value')
```

will store the value in the 'state' variable. An example of the check is as follows:

```
% Value is max if the button is pressed in
if state == get(hObject, 'Max')
     % this code will run if the button is toggled on
else
     % this code will run if the button is not pressed in
end
```

This minor but extra addition to code for toggle buttons make it preferable to use push buttons more. An exception could be for a pause button or similar control, which will pause the GUI when pressed and resume when the button is pressed again.

B.3.1.8 Axes

Axes define a region of a GUI specified for plotting graphs or pictures. In terms of programming, it is usually not necessary to define callbacks since other functions will work with the axes directly. Therefore, only the 'Tag' property is important to define.

Once a program grows into multiple GUIs, as is the case with OSIS, it is essential that if any graphs are being created they be done cleanly. That is, a user would not want to change what window they are looking at while running a plotting simulation and have the plots created on

their newly opened window. By default, a plot command will graph on whatever window was the most recently opened or accessed. To avoid this situation, consider the following code:

```
hAxes = findobj('tag', 'primary_axes');

while graphNow
     axes(hAxes);
     plot(xpts,ypts);
end
```

The "findobj" function can be used to find a handle based on a particular property value, in this case 'tag'. The handle is stored to the 'hAxes' variable. Within the graphing while loop, the "axes" function is used to set the current graphing axes to 'hAxes' before plotting. This ensures that no matter where a user is the plotting will occur in the correct figure. The axes could also be defined as the first argument to "plot".

B.3.2 Other GUIDE Tools

In addition to creating GUI objects, GUIDE has a few other practical tools that were useful in developing the GUIs of OSIS.

B.3.2.1 Align objects

From an aesthetic perspective, it is important that GUI objects have a consistent alignment. This can include vertical spacing in a column of buttons or having a series of static text displays aligned together from the left. To align a group of objects, hold shift and then left click on all the objects to be aligned together. Then click "Align Objects" on the toolbar or under the "Tools" menu. Options are available for alignment and distribution in both the horizontal and vertical directions with a user-defined spacing if required. Objects contained in different panels cannot be selected together, but their respective panels can.

B.3.2.2 Grid and rulers

To help with the placement of objects without needing to align them are the grid and rulers. They can be changed via "Grid and Rulers" under the "Tools" menu. Using the grid will divide the GUI display into a series of squares of a specified pixel-length. Rulers appear, if chose, along the top and left side of the GUI and measure in pixels. If the "Snap to Grid" checkbox is checked, any object that the programmer tries to place into the GUI will snap to the nearest gridline when it is close.

Note that the grid and rulers will not appear on the GUI when it is running.

B.3.2.3 Menu editor

It is very common for programs to have buttons available in their GUIs with similar options available from a menu bar. GUIDE has a Menu Editor for creating menus and menu items that can then be programmed similarly to regular object callbacks.

To open the Menu Editor, click on the "Menu Editor" button in the GUIDE toolbar or from the "Tools" menu. Click on the "New Menu" button, which will create a menu that can be given a name and a tag. A descriptive tag is not entirely necessary for the menus themselves. While a menu is selected, click on "New Menu Item" to add items to the menu, or click on "New Menu" to add a submenu with its own items. Menu items receive their own names and tags. Click on "View" to access their callbacks, or type a simple callback in the edit box provided.

As an example of using a menu item in conjunction with a push button, consider a GUI that needs to prompt to save data before closing. There may be a close push button and a close item under the "File" menu. The callback for each could call a single function, perhaps entitled "closeRequest", which will make the prompt for saving and then close the GUI. Thus multiple ways are provided to safely close, and if any changes need to be made they only need to be performed once. The code's efficiency has been increased and its maintenance decreased.

B.3.3 Passing Data Within a GUI

Typically, when data is presented to a GUI through the interface (Section B.3.1) or as input arguments to the GUI itself (Section B.3.4.2), it is stored to one or more variables within the respective function. If this data is not stored on a higher level, it will disappear once the callback or opening function has completed. Automatic output only occurs via the GUI output function (Section B.3.4.3), but even then the variables need a way to get there. Passing data within a GUI is single-handedly the most important aspect of properly coding a GUI. A user will not appreciate entering data into an edit box if the data is not actually used for anything.

This topic is presented before a GUI's primary callbacks because the theory behind it is essential to understand how those primary callbacks can allow access to variables for other callbacks.

B.3.3.1 The handles structure

By default, every GUI made with GUIDE will automatically contain a handles structure. It will typically contain a single noticeable field, but there are many more fields present. The programmer is free to append as many fields to this structure as necessary.

A GUI's handles structure contains all of the information about the GUI itself, including a field for every object contained in the GUI. Thus, it is literally a structure of handles (Section B.2.6), so to get or set any property of an object in the GUI, use handles.TAG_OF_OBJECT as the handle to the object.

The handles structure is automatically created in the opening function of every GUI made with GUIDE with the code

```
handles = guidata(gcf);
```

which will store the GUI information ('gcf' stands for "get current figure"; when the opening function runs the current figure must be the respective GUI). The next default line of code is

```
handles.output = hObject;
```

adding the 'output' field to the structure to be 'hObject'. As with GUI objects, 'hObject' here refers to the handle of the object whose callback in running, in this case the GUI itself. If the programmer is appending to the handles structure, all additions must be made after the handles structure is initialized and before it is saved with the line

```
guidata(hObject, handles);
```

If this line is not used, or fields are appended to 'handles' after it, changes will not be saved and the data will not be accessible to other functions.

B.3.3.2 Callbacks accessing data

As discussed in Section B.3.0 many GUI objects, including edit boxes and buttons, have a "Callback" function to implement code when the object is used (i.e. value entered, button pressed, selection made, etc.). Often, a variable will have its value adjusted. In order for the change to be noticed by other GUI components, the change must be registered with the handles structure and then saved.

The 'handles' structure is an input, along with the 'hObject' handle, for every callback function, so it is straightforward to access. Consider an edit box that expects a velocity that needs to be stored in order to output it when the window is closed. A velocity above 100 km/h is not accepted. The entire callback code is as follows:

```
value = str2double(get(hObject,'String'));

if value <= 100
     handles.velocity = value;
     % Update handles structure
     guidata(gcbo, handles);
else
     % Display old value in edit box
     set(hObject, 'String', handles.velocity)
     % Display warning to user
     msgbox('Velocity cannot be above 100 km/h');
end
```

The ‘velocity’ field of the ‘handles’ structure now contains the value entered in the edit box, if it is below or equal to 100. If the velocity is above 100, the previous value is displayed in the edit box and a message is displayed to the user to explain why the value was invalid. The ‘handles’ structure only needs to be saved when changes are made to it, not when its fields are being read.

It is important to note that the “guidata” function is called with ‘gcbo’ as the first argument, and not ‘hObject’. ‘gcbo’ refers to the handle of the figure that called the respective function; here the figure to call the function would be the GUI. It is acceptable to use ‘hObject’ for saving the ‘handles’ structure within an object callback.

B.3.3.3 Other functions accessing data

A GUI may contain functions that are not explicit callback functions for its objects. For example, there may be a function to change the value of several variables and display the changes on-screen. This type of function can be written as a sub-function in the same m-file as the GUI code, or written in a separate m-file (but not as a sub-function of that separate m-file or it cannot be accessed). The only requirements are that the function is passed the handles structure as an input argument and that the function saves any changes made to the structure. Consider the following call, which may be contained within multiple callbacks of the GUI:

```
changeDisplay(handles)
```

The “changeDisplay” function is called with the handles structure as the sole input. It can be called whenever the user changes the operation to be performed, or the numbers used to operate on. Here is the complete implementation of the function:

```
function changeDisplay(handles)
% A function to change the display of information in GUI X
% Input: handles - the handles structure of GUI X

switch handles.operation
     case ‘add’
          handles.answer = handles.left + handles.right;
     case ‘subtract’
          handles.answer = handles.left - handles.right;
     case ‘multiply’
          handles.answer = handles.left * handles.right;
     case ‘divide’
          handles.answer = handles.left / handles.right;
end

set(handles.answer_edit, ‘String’, handles.answer)

% Update handles structure
```

```
guidata(gcbo, handles);
```

Since this function is not an explicit callback, the handles structure must be saved via 'gcbo' and not 'hObject'. Here, this is no 'hObject' at all, but 'gcbo' will reference the GUI that called the function. While it may seem easier to change the answer directly when the operation is changed, it would not normally change when the left or right number is altered. The callbacks for all of those edit boxes can call "changeDisplay" to ensure updating is uniform. Note that the tag for the answer display box is entitled 'handles.answer_edit', while the answer itself is stored in 'handles.answer'.

An alternative to using 'gcbo' is to pass 'hObject' to the function, along with 'handles', and output the 'handles' structure to the calling function so that changes can be saved as would be done within a normal callback.

B.3.4 Programming a GUI's Primary Callbacks

A GUI will have its own callbacks, which are arguably more important than the callbacks of its objects. The "OpeningFcn" (Opening Function) can define any settings and create data based on inputs before the GUI is made visible to the user. The "OutputFcn" (Output Function) determines what forms of data are passed on as output from the GUI as it closes.

B.3.4.1 Initial function

At the beginning of every m-file created by GUIDE is a function whose name is that of the GUI it controls. This function contains initialization code for the GUI and should not be edited. Any practical code that must be performed before the GUI is available to the user can be implemented in the "OpeningFcn".

B.3.4.2 Opening function

The function where a GUI's inputs are manipulated and initial settings are defined is called "GUINAME_OpeningFcn". Any settings defined from GUIDE using the "Property Inspector" can be over-written, including here before a user can see what a property may have been. If the GUI was called with a number of input arguments, they are accessible here via the 'varargin' cell array. Varargin stands for "variable arguments in", although usually if one is using 'varargin' they are expecting a specific number of argument variables. To check the number of input variables, the following code can be used:

```
numOfVar = size(varargin);
numOfVar = numOfVar(2);
```

The "size" function creates a 1x2 row vector measuring the size of varargin. The first value will either be 0 if varargin has no values or 1 if varargin has any values in it. The second entry from

size will be the number of values in the 'varargin' cell array. Note that because of indexing it is necessary to use 'numOfVar' as a holder before having direct access to the actual number of inputs.

To access varargin's actual contents use normal cell indexing. It is important to know that the index values of 'varargin' correspond to the order in which the arguments were passed. It is also common to keep the names of variables consistent so that they are easier to keep track of. Consider an example where another GUI calls a GUI from a button callback using this line of code:

```
plotterGUI(xValues, yValues, title, colour, display)
```

In this example, 'xValues' and 'yValues' define points for a line, 'title' is a string for a graph title, 'colour' will be the colour of the plotted line, and 'display' is a Boolean to determine whether or not the plot is displayed. In the opening function of "plotterGUI", use the following code to extract the settings:

```
handles.xValues     = varargin{1};
handles.yValues     = varargin{2};
handles.title       = varargin{3};
handles.colour      = varargin{4};
handles.display     = varargin{5};
```

The variables are extracted using the order in which they were entered by referencing the 'varargin' cell array. Again, the same names are used for consistency throughout the different GUIs. PlotterGUI's opening function now has complete access to the data passed to it with identical nomenclature. Also, all of the data has been added to the handles structure so that other callbacks within plotterGUI can access it; this code must appear after the handles structure is created but before it is saved.

There is a disadvantage to this method of using 'varargin'. The programmer must continuously keep track of how many variables are being passed, and ensure that the references to varargin are numbered properly. If GUIs are being developed that pass large amounts of data to each other, such as for OSIS, then this process can quickly become cumbersome. The solution to this problem is to only use a single element of the 'varargin' cell array. The first element can hold a structure, which can be filled before the call to the GUI is made. Using the previous example again:

```
data.xValues   = xValues;
data.yValues   = yValues;
data.title     = title;
data.colour    = colour;
data.display   = display;
plotterGUI(data)
```

Now, in the opening function of "plotterGUI", the following code can be used:

```
handles.xValues     = varargin{1}.xValues;
handles.yValues     = varargin{1}.yValues;
handles.title       = varargin{1}.title;
handles.colour      = varargin{1}.colour;
handles.display     = varargin{1}.display;
```

Overall, a bit more code is needed, but this method is far easier to keep track of. This method is especially useful if the programmer prefers that all of the data is stored as fields of a structure; only one reference to 'varargin' in the opening function is necessary to pass all the data into a new structure.

Once the data has been extracted, but also before the 'handles' structure is saved, the GUI objects can be manipulated via the "set" and "get" functions. Other commands can also be run. Consider that the current example GUI has a set of axes with a tag of 'main_axes'. The following code can be added:

```
% Ensure plotting occurs in right place
axes(handles.main_axes);
if handles.display % handles.display is either true or false
     plot(handles.xValues,handles.yValues);
     set(handles.main_axes, 'Color', handles.colour, ...
          'Title', handles.title)
end
```

The colour and title could have been defined directly in the call to the "plot" function, but for the purpose of the example "set" was used. Similarly, use "set" to predefine any displays before the GUI becomes visible to the user. Note that, for a set of radio buttons in particular, each possibility for the 'Value' property (0 or 1) should be explicitly defined for each button with an if/else control.

B.3.4.3 Output function

The final function to be executed that has access to GUI data is "GUINAME_OutputFcn". When a GUI closes, the output function prepares the output arguments to pass via the 'varargout' cell array. Varargout stands for "variable arguments out". It works identically to 'varargin', described in the previous section, except that it must be written to explicitly, instead of read from. Using the single-cell method that was discussed with varargin, fields of the first 'varargout' cell can be given all of the output data and returned to a single structure.

Consider a call to a GUI, entitled "basicStatistics", that returns the mean, median, and mode of a series of numbers that a user enters via the interface. We will assume no inputs are necessary, so the varargin cell array is not used. Here is the implementation using the single-cell method:

Here is the call to the GUI with extraction of the returned structure:

```
responses = basicStatistics;

mean      = responses.mean;
median    = responses.median;
mode      = responses.mode;
```

Here is the code in the output function of "basicStatistics":

```
varargout{1}.mode   = handles.mode;
varargout{1}.median = handles.median;
varargout{1}.mean   = handles.mean;
```

Alternatively, using the multiple-cell method, here is how the call to the GUI is different:

```
[mean median mode] = basicStatistics;
```

Note how the output arguments are collected using a single-row matrix. The number of variables in the matrix must equal the number of cells in 'varargout'. Here is the code in the output function of "basicStatistics" using the multiple-cell method:

```
varargout{1} = handles.mean;
varargout{2} = handles.median;
varargout{3} = handles.mode;
```

Note as with varargin that the orders of variables must match when using multiple cells.

The 'handles' structure does not need to be saved within the output function, since it is about to be released.

B.4.0 VISUALIZING DATA WITH PLOTS AND VIDEOS

Matlab offers a broad range of options for graphing data contained in the current workspace. A number of functions exist that can be used to perform certain types of plotting, depending on the form of data that a user has available. No plots are displayed in the command window of Matlab; they are always drawn in a figure window. If no figure window exists, a default one will be created. A programmer can also specify the exact figure or axes in which the plot will be created.

A collection of plots can be put together to make a movie, either for viewing within Matlab or externally as an AVI (Audio Video Interleave) file. The development of OSIS has favoured creating AVI video files since they can be viewed on any computer.

The most basic plot functions in Matlab are “plot” and “plot3”. They are for two- and three-dimensional line plots, respectively. Within the call, certain properties can be specified, such as the line type, colour, point size and shape, and others. A complete list of properties is provided within the official Matlab help documentation.

B.4.1 Specifying Axes

When working with a complex GUI that may have multiple windows and figures open at once, it is essential that any plotting be performed correctly. If not, graphs may not always appear where they need to be.

B.4.1.1 The current axes

If a figure in the GUI has been created specifically for plotting, it must be opened before any plotting on it is attempted. The handle to either the figure itself or axes in the figure should be stored to a variable. Typically, a region of the figure will have already been designated for plotting, so the handle to the axes is most important. Specifying just a figure for plotting will use the whole figure for the plot, regardless of any other objects placed in the figure.

There are a number of ways to store the axes handle. The following method is ideal for a function that does not have direct access to the handles of the plotting GUI:

```
myFigure % Open the plotting figure
drawnow % Make sure figure is drawn
hFigure = gcf;
hAxes   = findobj(hFigure, 'type', 'axes');
```

In this case, the figure ‘myFigure’ was not already open. After it is called, the “drawnow” command forces it to appear and be the current figure. If it were already open, another “findobj” would be needed to get its handle since “gcf” (get current figure) could refer to a different figure. The call to “gcf” returns the handle of the current figure; it is guaranteed to work only in certain

circumstances, such as immediately after a figure has been opened or while a GUI's opening function is executing.

Note that in this case "findobj" searched based on 'type' within 'hFigure', the handle to 'myFigure'. This works only because 'myFigure' is assumed to have a single plotting axes. A different search criteria would be required if the figure contained multiple sets of axes. While it is possible to search based on 'tag', please note that the 'tag' of a set of axes can be cleared if the "cla" (clear axes) or "hold off" (allow new plots to over-write previous ones) commands are used in between plots.

B.4.1.2 Using "axes"

Just before any plot is about to be performed, a call to the "axes" function ensures that the plot will be drawn where it is expected to. The "axes" function can be passed the handle to the axes.

```
axes(hAxes)
plot(x, y);
```

In this case, the set of axes represented by 'hAxes' will be used to plot the data defined by 'x' and 'y'.

B.4.2 The Patch Function

The "patch" function is a low-level Matlab plotting function, and it provides an amount of freedom and versatility over higher-level functions. Its primary use is for graphing two- and three- dimensional objects that can be defined by a set of points in space and how the points are connected to create each face.

There are several methods for using "patch", however one method in particular is more intuitive to understand. This method does not use direct arguments to the function, but instead passes in the arguments to specific function properties. The call appears as follows:

```
patch('Vertices', vertices, 'Faces', faces, 'FaceColor', fColor,
    'EdgeColor', eColor);
```

In this call, 'vertices' is an n x 2 or n x 3 matrix where each row defines the x-y or x-y-z co-ordinates for a vertex of the object, 'faces' is a matrix where each row defines the vertices that come together to create a face, and 'fColor' and 'eColor' define the face and edge colouring, respectively. Note that 'fColor' and 'eColor' are 3-element vectors, a Matlab pre-defined color name, or 'none'. Other colouring options are available using 'FaceVertexCData' for non-solid colouring, and details of those options are available in the Matlab documentation.

Consider the following definitions of the inputs used for "patch":

```
vertices = [0 0; 1 0; 0 1; 1 1; 1 2;];
faces = [1 2 4 3; 3 4 5 NaN];
fColor = [0 1 0];
eColor = 'blue';
```

The 'vertices' matrix defines 5 points: 4 points to make a square in along the first quadrant, and an additional point above the square. The 'faces' matrix specifies that the first four points in the 'vertices' matrix will create a face (i.e. fill in the square) and that the third, fourth, and fifth points will create a face (i.e. create a triangle above the square). The order in which the vertices are listed in a row of 'faces' is the order in which they will be connected. The 'NaN' (Not a number) is required to fill up the second row of 'faces' because the dimensions must be consistent. Here, 'fColor' is a three-element vector, defining the RGB values for the faces. RGB values can range between zero and one, and the common resulting colours are given in this table:

R	G	B	Colour
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Hence, the faces will be coloured a solid green. Likewise, 'eColor' specifies that the edges of the faces will be coloured a solid blue. Shorthand names for the common colours are given by the first letter of the colour (ex: 'r' for red). The exception is black, which uses 'k'.

B.4.3 Proper Object Rendering

When graphing multiple objects simultaneously it is essential that they be rendered properly. If a line is passing through a sphere, one would expect to notice where the line enters the sphere and expect to not view the line any more until it has finished passing through. Unfortunately, rendering glitches can occur between multiple objects, including portions of one object appearing to have the colour that was meant to define another object. It is recommended to re-define the renderer used by Matlab in the opening function of any plotting GUI, or defined from the current workspace when the plotting figures will be created as needed. The default renderer changes depending on the type of plotting being performed, but it has been found that "zbuffer" is most effective for avoiding glitches between multiple objects. The call:

```
set(hFigure, 'Renderer', 'zbuffer')
```

will set the renderer of the figure defined by 'hFigure' to "zbuffer".

B.4.4 Building AVI Video Files

An excellent way to export a collection of plots is to create an AVI video file so that it can be viewed on any personal computer with the correct compression-decompression (codec) format. Frames are sequentially captured using functions designed to compile a video.

B.4.4.1 Creating the AVI object

Before any frames can be assembled together, the video file must be created as an AVI object using the "avifile" function. The call includes the video filename (must come first), frames-per-second rate, compression method, and quality level. For example,

```
aviobj = avifile('my Video', 'fps', 25, 'compression',
    'Cinepak', 'quality', 90);
```

creates the video object using the name 'my Video.avi' (the '.avi' extension is appended automatically if it not there), a frames-per-second rate of 25, Cinepak compression, and a quality level of 90. The quality level must be a number between 0 and 100 inclusive. A 'videoname' property is also available to give the video a name independently of the filename used.

Cinepak is the most effective codec bundled with Matlab; the other codecs are terrible for generating videos from object plots. There are other free codecs available for use if they are installed on the user's computer. The Xvid codec is recommended for its consistent quality and smaller file sizes, and is available from www.xvid.org.

If a video with the chosen filename already exists, it will not be overwritten. The computing resources will still be taken to compile the frames, but the video itself will not be saved. A workaround to this is to perform the following check before making the call to "avifile":

```
if exist('my Video.avi', 'file')
     delete('my Video.avi')
end
```

Now the previous video file will be deleted before the new one is created, effectively overwriting the file.

The returned argument, stored to 'aviobj', is the new video object itself. Note that since no frames have been added to it yet, it is an empty video.

B.4.4.2 Capturing a frame

It is most likely that a video's frames will be generated within some form of loop, such as a 'for'-loop or 'while'-loop. Within the loop, the plotted frame must be captured and then added to the video object.

First, one must be sure that the generated graph will actually be visible on-screen; AVI video files cannot be made up of frames that are not each displayed on the monitor. Using the "drawnow" command after the plotting calls and before attempting to capture the frame guarantees that the plot is made visible. The "getframe" function is used to perform the capture itself. The call to "getframe" can vary depending on what needs to be captured. If it is not passed any arguments, the frame will come from the current axes (only the axes; no labels). The function can be given a handle to a figure or axes to specify from where to get the frame. The most versatile alternative is to pass the function a handle to a figure as well as a four-element vector specifying the region of the figure to use. This allows specification of a region of the figure so that a set of axes and its labels will be included in the frame. The call

```
frame = getframe(hFigure, [5 10 50 100]);
```

will capture a frame from the figure specified by 'hFigure' and store it to 'frame'. The frame area will begin five pixel units from the left and ten pixel units from the bottom of the figure. The frame will have a width of 50 pixel units and a height of 100 pixel units.

Once the frame has been obtained, the "addframe" function stores the frame to the AVI video as follows:

```
aviobj = addframe(aviobj, frame);
```

The frame stored to 'frame' is now part of the video object defined by 'aviobj'. This process is required for every frame in the video. There is no limit to how many frames can be added to a single video object.

B.4.4.3 The final AVI object

When all of the frames have been added to the video, the "close" function is needed for the file to be accessible. The call

```
aviobj = close(aviobj);
```

finishes the process of building the AVI video file. If the call to "close" is not made, Matlab must be shut down for the file to be moved, deleted, or opened.

The finished file can be viewed by any media player that supports AVI files, and on any computer that has the codec that was used to create the video.

B.4.5 Creating Image Files

It may seem unintuitive to describe the procedure for exporting image files after explaining the building of AVI video files, however the process is somewhat more complex. Less actual code is

required for single images, though it is easier to grasp after having learned to compile frames to a video.

Matlab supports many image file types, and a complete listing is provided in the official Matlab documentation. Formats include '.png', '.bmp', '.jpg', and others. In order to create an image of a particular format, the proper extension must be appended to the filename; otherwise, the code itself is identical.

Image files, like AVI video files, require the use of "getframe" to capture a region of a figure or axes on-screen. The returned frame is actually a structure with fields that include 'cdata' and 'colormap'. These fields are needed to properly generate the image with the "imwrite" function. Consider the following code:

```
hFigure = gcf;
filename = 'myPicture.png';
frame = getframe(hFigure);
if isempty(frame)
     imwrite(frame.cdata, filename);
else
     imwrite(frame.cdata, frame.colormap, filename);
end
```

If the contents of 'frame' did not actually have any data (which would be a very rare occurrence considering that if there is no figure before a call to "gcf", then one is created), only the 'cdata' field is saved to the image. Otherwise, both 'cdata' and 'colormap' are used, and 'myPicture.png' becomes a valid image file that can be viewed by any program that can open image files.

B.5.0 READING AND WRITING TEXT FILES

Matlab offers various methods for saving data. The "save" function is used to store any Matlab variables to a special file, called a MAT-file. MAT-files can then be loaded at a later point in time to access the variables again. It is possible to store all of the data in the current workspace or specify particular variables to save. The MAT-file format is ideal for work that will be conducted completely in Matlab since it is a native Matlab file type. Unfortunately, using MAT-files limits the readability of the data to Matlab itself, so a broader method has been used for the development of OSIS. For more information on the "save" function, please refer to the official Matlab documentation.

Creating text files to store data provides a broader range of options than MAT-files, including being able to read and edit them without executing Matlab itself. It also allows data to be read by other programming languages.

There are different functions available for reading and writing text files. The two described here offer a lot of flexibility. The "fprintf" function is used to write lines of text, while "textscan" is used to read them. Both of these functions make use of a common file handle to refer to the file in question, along with functions that open and close the file.

B.5.1 Finding a File to Use

Whether a file is going to be opened or created, there are functions available to provide the standard windows for opening and saving files. These functions include warnings of over-writing previous files, and return the name and directory path of the chosen file.

The "uiputfile" function displays the dialogue window for saving a file. It can be passed a specific extension or set of extensions to use, a title for the window, as well as a default filename, in this order. The call

```
[filename pathname] = uiputfile('.abc', 'Save data as', ...
    'myData');
```

limits the user to creating a file with the '.abc' extension. In order for the chosen directory to be used properly, the "strcat" function must be called to concatenate the file and path names together before creating the file.

The "uigetfile" function displays the dialogue window for opening a file. It can be passed a specific extension or set of extensions to use, a title for the window, as well as a default filename, in this order. The call

```
[filename pathname] = uigetfile({'.abc'; '.xyz'}, ...
    'Open file', 'myData');
```

limits the user to opening a file with the '.abc' or '.xyz' extension. In order for the chosen directory to be used properly, the "strcat" function must be called to concatenate the file and path names together before opening the file.

Please note that "uiputfile" and "uigetfile" do not actually open or create any files; they only provide the string to use as the name for opening the file. They are useful for any function that requires a filename string.

B.5.2 Creating and Using the File Handle

When a file is opened, the handle to the file is returned and is used for all calls to read from or write to the file. It is very important to close the file once all tasks using the file are complete.

The "fopen" function is used to open the file. A format must be specified declaring how the file will be used, or it will be opened for reading only by default. Typically, the returned handle is stored to the variable 'fid'. The call

```
fid = fopen(filename, 'wt');
```

opens the file whose name and directory are specified by the 'filename' string. The mode 'wt' is for creating a new file or re-writing the file if it already exists. In this mode, any previous contents in the file are discarded before writing.

It is essential to perform a check on the file handle, 'fid', before any attempt is made to reference it. The standard check is as follows:

```
if fid == -1
     msgbox('File could not be opened or created', 'Warning',...
     'warn', 'modal');
     return
end
```

The handle 'fid' will have a value of –1 if the "fopen" call failed to properly open the file. The "return" command exits whatever function or script the code is contained in, preventing any attempt to use the file.

When use of the file has been completed, the call

```
fclose(fid);
```

closes the file.

B.5.3 Writing to a File

As mentioned, the "fprintf" function is ideal for writing lines of text. Calls to this function require a properly available file object handle, the format of the text to be written, and then the values to be used. The format is specified as a string declaring the types of data to write and in what order. The '%' symbol is used to denote each data type within the format string.

Consider a simple example to write a single value:

```
fprintf(fid, '%d', x);
```

The format string '%d' specifies that a decimal notation number will be written. If the variable 'x' is not a numerical value, an error will occur. The data type itself can have a particular format. Consider:

```
fprintf(fid, '%2.3d', x);
```

The '2' before the decimal point specifies a minimum of two digits to be printed, while the '3' after the decimal point specifies the number of digits of 'x' to print after the decimal point. Using '0' after the decimal point will guarantee a printed integer value.

Consider writing a single string:

```
fprintf(fid, '%s', 'Greetings people');
```

Here, the format string '%s' specifies that a string will be written. In this case, an actual string was passed, but it is also possible to pass a variable that holds a valid string. Please refer to the official Matlab documentation for other data types with "fprintf".

Now consider an example where "fprint" is directed to write multiple values with a particular spacing:

```
fprint(fid, '%s\t%d %d\n', 'Two numbers are:', myVector);
```

This case prints a string and two values. A tab is placed in between the string and first decimal value using '\t', the blank space in between the two decimal values means that a blank space will appear between the values in the file, and '\n' denotes that a new-line character is inserted, meaning that the next call to "fprintf" will be entering text on the following line. The data arguments are given in the order they are requested by the format string, with commas separating them. It is expected that 'myVector' will contain two numerical values in its first two elements or an error will occur.

Using "fprintf" multiple times will begin writing where the previous call left off, so it is important to use new-line characters to avoid having an entire text file made up of a single line.

B.5.4 Reading a Created File

As mentioned, the "textscan" function is ideal for reading lines of text. Calls to this function require a properly available file object handle and the format of the text to be read. The format is specified as a string declaring the types of data to read and in what order. The '%' symbol is used to denote each data type within the format string. The notation for data types used by "textscan" is different from that of "fprintf".

Before designing code to read a file, it is important to know what the file will look like, so the code for creating the file should always be completed first.

Reading in data is more complicated than writing the data for three reasons. The first is that all data read by "textscan" is returned to a cell array that must be referenced properly to extract the actual data. The second reason is that the data format must be well understood before it can be properly read. The third reason is that it is more complicated to trouble-shoot; a problem in writing a file may be easily identified by opening the created file, while actual debugging would be needed for resolving problems in reading data.

Consider a line of text at the beginning of a file with three integers, each separated by blank space. Fortunately, "textscan" does not need to know how much space exists in between each integer, or even if they are on the same line (unless we specify to scan the first line only). The call to the function can be as follows:

```
holder = textscan(fid, '%d %d %d', 1);
myVector = [holder{1}(1) holder{2}(1) holder{3}(1)];
```

The '%d' specifies an integer (not a decimal number as with "fprintf"), and the spaces in between each '%d' are only used to make the code easier to read. The 1 defines that the scan will only be performed once. The three integers are stored to the 'holder' cell array, whose first element of each of the three cells must be referenced to extract into a simple, three-element vector. Use '%f32' for reading numbers as single precision and '%n' for reading numbers as double precision. Please refer to the official Matlab documentation for a complete listing of data formats with "textscan".

Like "fprintf", '%s' denotes a string. By default, a single string is considered complete with the first blank space after a character. Changing the delimiter property will force a string to be read until a new-line character is encountered. However, if no string is located where expected, and the new-line character is being used as the delimiter, "textscan" will continue scanning until it finds a character to begin a string, even if it is only supposed to scan once. This creates the problem of either using a single-word string, or guaranteeing that a string will always be present. Consider this call to read until the new-line character:

```
holder = textscan(fid, '%s', 1, 'delimiter', '\n');
myString = holder{1};
```

The string in this case could be a single word or a lengthy one-line sentence; in any case, it will be stored to the first cell of 'holder'. Please note that in some cases for a string cell to be properly extracted it must be referenced twice. This would appear as follows:

```
holder = textscan(fid, '%s', 1, 'delimiter', '\n');
holder = holder{1};
myString = holder{1};
```

Using "textscan" multiple times will begin reading where the previous call left off. Except for the case of strings with spaces, it is unnecessary to consider the new-line characters, as they will be skipped while scanning.

The real advantage of "textscan" is that it can selectively ignore contents in a file, which becomes especially useful when a file contains comments and labels that are to be ignored. Consider a line in a file that reads "Length (mm) = 15". It is already expected that the length is going to be read, but the magnitude and units are unknown. The only expectation is that the unit will be contained within a pair of brackets, and that the value will follow the equals sign, in that order. The line may also read "The length (mm) is found to be = 15". The following code can extract and reference the data properly:

```
holder = textscan(fid, '%*[^(] %*c %[^)] %*[^=] %*c %n', 1);
length = holder{2};
units = holder{1};
```

The code does seem to be very complex initially, though each symbol performs a specific task. The first entry to scan, '%*[^(]', is to scan up to but not including the first appearance of the left bracket, and discard whatever was scanned. The square brackets denote that a list of characters to scan will be given, the exponential symbol, '^', requests the scan to be performed until a character in the list is found, and the star symbol, '*', is used to discard what was scanned. The second entry, '%*c', reads and discards the next character, in this case known to be the left bracket. The third entry, '%[^)]', will scan all characters up to but not including the next right bracket, thus storing the units to a string.

The fourth entry, '%*[^=]', scans up to but not including the equals sign and discards the contents. The fifth entry, '%*c', reads and discards the next character, in this case known to be the equals sign. Finally, '%n' reads in the length as a double precision number. Note that even though five scans were performed, three were skipped so the 'holder' cell array only has two cells. The first cell contains the string of units and the second cell contains the length value. The delimiter property did not need to be re-defined in this case because it is explicitly specified how to scan the characters in the line.

The process of scanning up to a particular character and ignoring everything before it allows any number of lines of text to appear beforehand, as long as the special character is not present within the text.

The importance of knowing the contents of data before making a call to “textscan” has already been noted. A case may arise where a line is known to contain a row of numbers, but it is unknown how many numbers there actually are. The process for scanning this data is to first read it as a string with the new-line as a delimiter, convert the string to a character array, and then convert to a numeric array:

```
holder = textscan(fid, ‘%s’, 1, ‘delimiter’, ‘\n’);
holder = char(holder{1});
myRow = str2num(holder);
```

The “char” function does the string-to-character-array conversion, while the “str2num” function does the character-to-numeric conversion. An alternative to using “char” is to reference “holder” twice before using “str2num”. Using either method will make ‘myRow’ a row-vector of numeric data.

B.6.0 CREATING AND USING FORTRAN MEX-FILES

Sometimes, it can be necessary to integrate a program into Matlab that was originally designed in a different programming language. Matlab has native support for working with code generated in C and Fortran. If an appropriate compiler exists, Matlab can compile a MEX-file (with a '.dll' extension) from C or Fortran code. A MEX-file can be given arguments and return variables just as an m-file can, although there are a few distinct differences. MEX-files are not as easily editable, although they use the same workspace as that of the function that calls it (m-files have their own workspace and variables). This section will focus on creating MEX-files from Fortran code.

The information in this section regarding MEX-files, how they are compiled, and the functions available in coding them, was mostly gathered from the Matlab External Interfaces documentation. The most comprehensive form of this documentation can be found at the following address: http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf.

B.6.1 Fortran to Matlab Overview

An important consideration in compiling Fortran code into a Matlab MEX-file is the data types involved. The Matlab language only has a single object type: the Matlab array. All variables are stored as arrays. In Fortran this object type is referred to as an mxArray, and all arrays must be of double-precision numbers or strings.

A series of routines exists to manipulate Matlab arrays using Fortran code. All the routines begin with 'mx', short for 'mxArray'. A separate series of routines begin with 'mex' and perform specific operations in the Matlab environment, such as printing a message in the command window.

MEX-functions are unique in terms of workspace use. It uses the workspace of the calling function and not its own, and its variables are persistent, but it automatically releases any data from memory when it has completed. However, it is possible to manipulate variables in the workspace via special 'mex' subroutines without having the variables declared explicitly as arguments to the MEX-function. Other 'mex' subroutines create Matlab data types in the workspace that remain after the MEX-function has executed. Additional routines can read from and write to Matlab MAT-files. In summary, powerful tools are available.

It is also possible to call Matlab functions from within Fortran code via the "mexCallMatlab" subroutine, whose arguments include a string containing the name of the Matlab function/operator/m-file to be called.

Debugging in Matlab on a Windows machine is possible, but the Microsoft compiler is required. This compiler must be opened, and Matlab must be opened from within it. The MEX-file source code can then be opened and debugged (not the MEX-file itself).

B.6.2 Modifying Source Code

The primary advantage to creating a MEX-file is to avoid having to translate every individual line of code in a program to Matlab code. It is necessary, however, to create a gateway subroutine in the source language so that it can interact with Matlab properly.

A Fortran subroutine entitled "mexFunction" must be created to act as a gateway to the Fortran computational routines. When the source code is being compiled, Matlab looks to this function for very specific directions regarding how the arguments are used and data is assigned. The "mexFunction" can be located in a separate file, in the same file as the routine it is supposed to call, or it can be written to replace an interface program.

The following naming and declarations must be used to begin any mexFunction:

```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer nlhs, nrhs, plhs(*), prhs(*)
```

where: nlhs – number of left-hand side arguments
nrhs – number of right-hand side arguments
plhs – a length nlhs array of pointers to the left-hand side outputs
prhs – a length nrhs array of pointers to the right-hand side inputs

The values of nlhs and nrhs are determined when a call to the compiled MEX-file is made. For example, having compiled the Fortran routine 'myRoutine' it can be called as such from within Matlab:

```
z = myRoutine(x,y);
```

The value of nlhs would be 1 and the value of nrhs would be 2. The mexFunction should include code that checks the values of nlhs and nrhs to make sure they are what are expected for the MEX-file, and the routine 'mexErrMsgTxt' can display a message to explain how many arguments are required. For the example above, prhs would contain pointers to the arguments x and y, which must be converted to Fortran pointers using mxGet routines in order to pass them correctly to the computational routines.

It is the responsibility of the programmer to ensure that the pointers of plhs are assigned to data that Matlab can read, and that this data is created before the subroutine finishes using the 'mxCreate' series of routines. The mxCreate routines create mxArray structures, so in order to pass the created pointers to computational routines they must first be converted to Fortran pointers using the mxGet routines. The following example creates an m x n matrix to serve as output, where 'm' and 'n' are values previously defined in the mexFunction, and the only input argument to the MEX-file is a matrix:

```
plhs(1) = mxCreateDoubleMatrix(m, n, 0)
yp = mxGetPr(plhs(1))
xp = mxGetPr(prhs(1))
```

```
call convertMatrix(%val(yp), %val(xp))
```

'mxCreateDoubleMatrix' builds an m x n matrix of double-precision values. 'mxGetPr' converts the pointers to the real components of the plhs and prhs arrays to Fortran pointers. The %val construct is used to pass the value and not the reference of the pointer to the convertMatrix routine. 'xp' and 'yp' had to have been declared at the beginning of the routine in the proper format (i.e. as integers)

Note that the Compaq Visual Fortran compiler supports the %val construct; if it were not supported it would be necessary to use 'mxCopy' routines to copy the input pointer to a local array, and then copy the necessary local array back to the output pointer after the computational routine had been called. The same code without %val would appear as follows:

```
plhs(1) = mxCreateDoubleMatrix(m, n, 0)
yp = mxGetPr(plhs(1))
xp = mxGetPr(prhs(1))

call mxCopyPtrToReal8(xp, xpr, m*n)

call convertMatrix(ypr, xpr)

call mxCopyReal8ToPtr(ypr, yp, m*n)
```

Additionally, it would be required to declare 'xpr' and 'ypr' as arrays of size m*n at the beginning of the mexFunction.

The full listing of Fortran 'mx' and 'mex' routines is located at http://www.mathworks.com/access/helpdesk/help/techdoc/apiref/. The documentation that is bundled with Matlab only includes descriptions for the C versions of the functions.

B.6.3 Fortran Code Notes

If the data to be given to a Fortran program is not numeric in nature, it has been found that the data can be difficult to properly convert to Fortran. Rather, it is recommended to create a text file in Matlab that the Fortran program can scan and read. The process of reading input files using Fortran code is beyond the scope of this work.

It has been found that Fortran code that writes to an output file directly can crash Matlab when the compiled MEX-file is executed, possibly due to output format lines contained at the end of the file. It is recommended to return any data directly to the Matlab workspace or a MAT-file and write to an output file if desired from within Matlab.

B.6.4 Compiling Into a MEX-File

The process of actually compiling the Fortran code into a MEX-file is fairly straightforward, and is sufficiently described in the Matlab external interfaces documentation, found here: http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf.

B.7.0 MISCELLANEOUS TOPICS

Matlab contains far more capabilities than those described in this guide, however there are a few extra topics that merit discussion. This section does not describe how to perform any particular task, but offers general advice and notes to consider.

B.7.1 Debugging

Matlab is very powerful as a high-level language in that many problems that exist with lower-level languages do not need to be taken into account. However, when errors do occur it is advantageous to perform proper debugging. An error message may point out a particular line of code where a problem occurred, but the original cause of the error may have been earlier in the code.

The debugger can be accessed when the M-file editor is opened. A portion of the toolbar is dedicated to debugging options, and there is a “Debug” menu along the menu bar.

Inserting a breakpoint into a piece of code will halt the execution of the code before the line marked with the breakpoint. Setting the breakpoint can be performed by left clicking on a horizontal line along the left side of the code. The horizontal lines only appear to the left of executing commands and not comment lines or function headers. A valid breakpoint appears as a red circle over the horizontal line.

When the code runs and is stopped by the breakpoint, the programmer can use the Matlab command window to execute any commands before continuing, including checking the value of anything in the current workspace. The debugging toolbar buttons can be used to execute the code one line at a time, so variables can be continuously checked for their proper values. The code can be executed as normal by pressing the “Run” button, and execution can be broken by pressing “Exit Debug Mode”.

If changes are made to the code while debugging, they will not be taken into account and the red breakpoint circle will turn grey. Debug mode must be exited in order to save changes. To remove a breakpoint, click on the corresponding circle.

An alternative to true debugging can be to use commands that will display a result in the command window during execution, either by not using a semi-colon at the end of a command line or by writing a particular variable on its own, without the semi-colon, within the code. This method is good if there is a loop where a variable is expected to have a specific value. Outputting the value to the command window from within the loop is an effective quick check before attempting to debug all iterations of the loop.

B.7.2 Spreadsheets

The default Excel spreadsheet-building function, "xlswrite", may at first seem inadequate for creating spreadsheets with column headings, since the only data input is for the matrix of information to store. The workaround for this is to create a cell array. The first row of the cell array can be filled with all of the column headings, and then the remaining rows can be filled with the numeric values required. This cell array can be passed to "xlswrite", which will properly create the Excel spreadsheet with proper heading alignment.

B.7.3 Multi-Instancing

With the presence today of multi-threaded and multi-core processors, it would seem ideal to run different sets of code at once. The ability to do so would greatly enhance the integration component of OSIS. Unfortunately, the current nature of Matlab prevents it from executing two or more pieces of code at once; the Matlab engine can only perform one task at a time.

There is an alternative method for taking advantage of a multi-threaded or multi-core computer, and that is by running multiple instances of Matlab simultaneously. It is possible to have more than one copy of Matlab opened at once, depending on the user's license. Each copy runs independently of the others, and has its own workspace. Two copies can be executing the same file at the same time and not interfere with one another. The notable exception is with video generation; since Matlab needs the plot to be displayed on-screen in order to capture it, making two videos at the same time can result in a blend of the two. This is especially true if the plotting windows appear in the same section of the screen. Using one copy of Matlab to perform calculations and another to make a video is viable, so long as one does not depend on the other for data.

On a similar note, there does not appear to be a command in Matlab to open another copy of Matlab.

B.7.4 Using Single or Double Numbers

When a real number is created in Matlab, it is stored by default as a double precision number. Double precision numbers use eight bytes of space for storage. Single precision number use only four bytes, but at the cost of reduced precision. Any number can be converted to single precision by passing it to the "single" function. For many purposes, the accuracy of single precision numbers is more than sufficient, and they are actually much faster for computations. When large amounts of data are being stored in the Matlab workspace, considerable time and space can be saved by using single precision values. Please note that functions do exist that require double precision values; they will return errors if passed single precision numbers.