isvr Institute of Sound and Vibration Research

AutoROV
An Underwater Flight Vehicle Simulation Program

R.K. Lea

ISVR Technical Memorandum 828

July 1998

University of Southampton

SCIENTIFIC PUBLICATIONS BY THE ISVR

Technical Reports are published to promote timely dissemination of research results by ISVR personnel. This medium permits more detailed presentation than is usually acceptable for scientific journals. Responsibility for both the content and any opinions expressed rests entirely with the author(s).

Technical Memoranda are produced to enable the early or preliminary release of information by ISVR personnel where such release is deemed to the appropriate. Information contained in these memoranda may be incomplete, or form part of a continuing programme; this should be borne in mind when using or quoting from these documents.

Contract Reports are produced to record the results of scientific work carried out for sponsors, under contract. The ISVR treats these reports as confidential to sponsors and does not make them available for general circulation. Individual sponsors may, however, authorize subsequent release of the material.

UNIVERSITY OF SOUTHAMPTON

INSTITUTE OF SOUND AND VIBRATION RESEARCH

FLUID DYNAMICS AND ACOUSTICS GROUP

AutoROV
An Underwater Flight Vehicle Simulation Program

by

R K Lea

ISVR Technical Memorandum No. 828

July 1998

Authorized for issue by
Professor P A Nelson
Group Chairman

Acknowledgements

The author would like to express his thanks to Florida Atlantic University for the donation of the original simulation program. Also appreciated was the use of the Department of Ship Science's Austin Lamont Towing Tank for propeller testing and the use of DERA Haslar's Ocean Basin for vehicle testing.

Abstract

AutoROV is a program written in C that simulates the motion of the tethered underwater flight vehicle, *Subzero II*. This report contains an annotated listing of the entire program as well as descriptions of the physical models used. It is intended for those wishing to use to program as well as those wishing to modify it.

Table of Contents

Part I — Use and Theory of the Simulation

Part II — Module Listings

List of Figures

List of Tables

Nomenclature and Glossary

BAR Blade Area Ratio

ROV Remotely Operated Vehicle

η Global position vector

$\boldsymbol{J}$ Euler angle transformation matrix

$\boldsymbol{v}$ Vehicle velocity vector

β Advance angle used in propeller model

δr Rudder deflection [rad]

δs Sternplane deflection [rad]

λ_r Constant used in finite difference method = $\frac{\Delta t}{2\Delta s}$

ϕ Global roll angle [rad]

φ Tether angle [rad]

θ Global pitch angle [rad]

ρ Water density [kg/m^3]

ψ Global yaw (heading) angle [rad]

ω Propeller speed [rad/s]

B Vehicle buoyancy [N]

C_{dn}, C_{dt} Normal and tangential drag coefficients of tether, repectively

C_Q^*,C_T^* Torque and thrust coefficient respectively, used in propeller model

d Tether diameter [m]

D Propeller diameter [m]

E Young's modulus (tether) [Pa] or back emf (motor) [V]

h Inter-node spacing in finite difference method

I Moment of inertia [kgm^2]

I_a Armature current [A]

J_T Thruster inertia

k_φ Motor constant [Vs]

L_a Armature inductance

m Vehicle mass [kg]

m_a Added mass per unit length of tether [kg]

m_c Mass per unit length of tether [kg]

n Propeller speed [rev/s]

p Roll rate [rad/s]

q Pitch rate [rad/s]

Q Torque [Nm]

r Yaw rate [rad/s]

R_a Armature resistance [Ω]

R_f Effective resistance of FET drive [Ω]

s Distance along tether [m]

t Time [s]

T Tether tension [N]

u Surge velocity [m/s]

v Sway velocity [m/s]

V_a Armature voltage [V]

V_b Brush voltage [V]

w Heave velocity [m/s]

W Vehicle weight [N]

x, y, z Vehicle position relative to Earth [m]

x_B, y_B, z_B Position of centre of buoyancy of vehicle [m]

x_G, y_G, z_G Position of centre of mass of vehicle [m]

Part I

Use and Theory of the Simulation

1

Quick Start

The AutoROV simulation program is designed to simulation the motion of *Subzero II*, a tethered underwater flight vehicle described below. The program is written in C and was developed and run using Borland C++ v4.5. Throughout this manual it is assumed that the reader is conversant with C programming.

1.1 The Vehicle

The *Subzero II* vehicle has a cylindrical hull, is made from perspex, and has removable nose and tail sections (Figure 1). For ease of access, which is important in such a test-bed, the drive and control gear are mounted on a removable tray inside the centre section.

The overall vehicle layout is shown in Figure 2. Propulsion is from a 250W, 16,000 rpm, samarium-cobalt DC motor, powered by a 9.6V Ni-Cad battery pack which gives a maximum speed of 2m/s. The supply is controlled by a set of power MOSFETs in an H-bridge chopper arrangement; this allows forward and reverse action. The chopper is controlled by an 800Hz pulse-width modulated (PWM) drive. The original specification called for a propeller diameter of 10cm; a pitch ratio of 1.0 and a blade area ratio (BAR) of 0.12 is currently used. Such a low BAR is atypical of marine vehicles, and thus the propeller is actually a reshaped

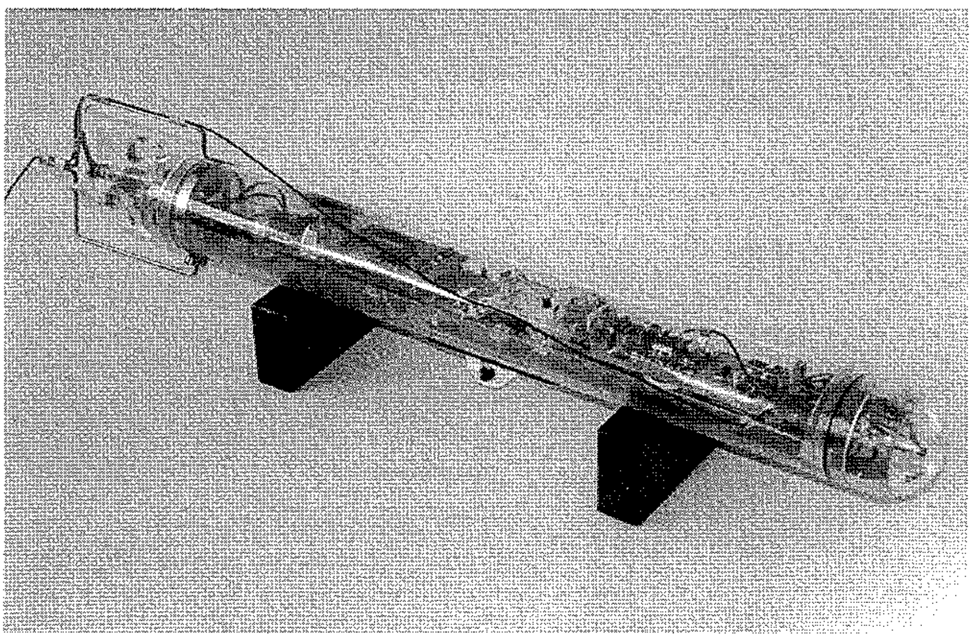


Figure 1: *Subzero II*.

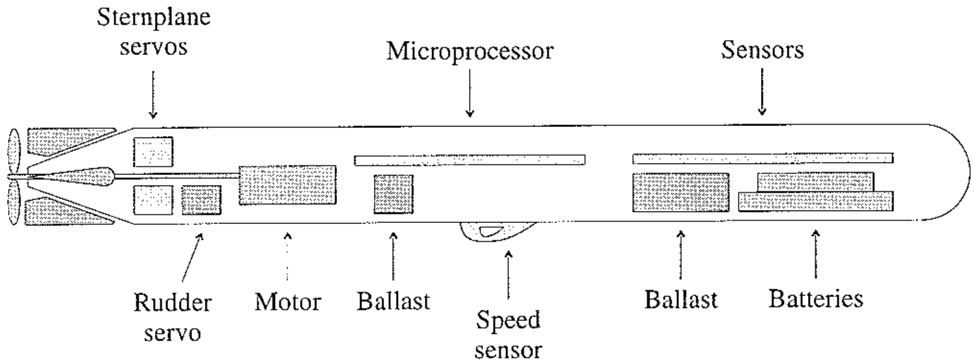


Figure 2: *Subzero II* internal layout.

model aircraft propeller. The motor is geared down by 5:1 for the propeller.

The four control surfaces, a linked rudder and two independent sternplanes, are actuated by model aircraft servos. The roll mode is currently passively stable as the heavy components such as the battery pack and the motor are mounted as low in the vehicle as possible. Although this has proved sufficient so far, the roll mode may be actively controlled by means of the independent sternplanes if necessary.

Vehicle control is achieved using a host PC on the shore which communicates with the ROV over a bi-directional link. To reduce the load on the PC, communications are handled by two Motorola 68HC11 8-bit microcontrollers (MCUs) operating at 2MHz: one onboard the ROV, the other operating within the PC on a custom-built communications card. The vehicle MCU collects sensor data such as propeller speed and depth before transmitting this to the PC. The host uses this information together with pilot demands to control the vehicle. These control signals are transmitted back to the ROV MCU which adjusts the pulse width modulated signals to the motor drive and the control surface actuators. Sensor limitations reduce the ROV-PC communications update rate to 10Hz although the actual data transmission rate is much higher.

The physical data link between the ROV and the PC is either a fibre-optic or wire cable. Currently, the wire link is used — it employs RS-422 differential drivers and receivers in a full-duplex configuration and thus requires two twisted-strand cable pairs (one pair for each direction). Also available is a special-purpose link for vehicle MCU programming and debugging which can be used when the vehicle is out of the water.

The sensors installed in the ROV are three rate gyros, three accelerometers, two pressure sensors, an external speed sensor, an optical shaft encoder and a TCM2 digital compass module. Together they provide data on the vehicle as shown in Table 1. Information on work done on the vehicle can be found in Lea, Allen and Merry [1] and Lea [2].

Table 1: Summary of the *Subzero II* sensor package.

Measurand	State	Sensor	Range	Resolution	LPF	HPF	Update rate	Latency†
Depth	z	Pressure	0-6m 0-3m	2.3cm 1.1cm	–	–	–	<0.1ms
Roll Pitch Heading	ϕ θ ψ	TCM2 digital compass module	±50° ±50° 0-360°	0.3° 0.3° 0.1°	–	–	10Hz	100ms
Speed	u	Pressure	0-1.6m/s	variable††	100Hz	–	–	<0.1ms
		Impeller	>0.7m/s	variable††	–	–	>18Hz‡	–
Surge accel Sway accel Heave accel	$\dot{u}$ $\dot{v}$ $\dot{w}$	Accelerometer	± 4m/s²	3.1cm/s²	50Hz	0.2Hz	–	<0.1ms
Roll rate	p	Rate gyro	±90°/s	0.7°/s	–	–	–	<0.1ms
Pitch rate	q		±90°/s	0.9°/s	–	–	–	<0.1ms
Yaw rate	r		±73°/s	0.57°/s	–	–	–	<0.1ms
Prop speed	n	Optical encoder	±2750rpm	1.2rpm	–	–	10.17Hz	–

LPF = low pass filter; HPF = high pass filter

† Time between when the state is sampled and when valid data is received by MCU. Where no value is given, the figure reported by the sensor is an average across a sample period.

‡ Speed sensor characteristics are speed dependent.

Sensors displayed in light type were not used in tests and so were not calibrated. Any data given is nominal.

1.2 The Program

The program has a number of options that alter the vehicle or simulations model — e.g. whether to model the tether or ignore its dynamics, whether to use a PID or sliding mode autopilot, etc. These options are currently set within a header file and thus the program needs to be recompiled each time these are changed. This, together with a command file of vehicle manoeuvres comprises the inputs to the simulation.

The simulation output is numeric: the data displayed on screen can be set from within the header file; a comprehensive data file is created each time the program is run regardless.

1.2.1 Command File

The command file is used to give a list of manoeuvring commands to the vehicle. These commands are indexed against time and cover the three manoeuvring systems — speed, heading and depth. There are two variants of the command file depending on the type of autopilot selected.

The usual (PID, sliding mode, etc.) autopilots attempt to keep the vehicle going at the commanded speed, on the correct course and at the right depth. Thus, the commands in the file specify the speed u, course (or heading) ψ and depth z. Speed is in metres per second, heading in de-

Table 2: Typical simulation command file (autopilot).

Time sec	ψ_d deg	z_d m	u_d m/s
1	0	1	1.3
5	90	1	1.3
10	90	5	1.3
15	0	5	1.3
20	0	0	0

Where ψ_d, z_d and u_d are the heading, depth and speed demands, respectively.

Table 3: Typical simulation command file (fixed controls).

Time sec	δr_d deg	δs_d rad	m_d
1	0	0	600
5	-10	0	600
10	10	0.175	600
15	0	0.175	600
20	0	0	0

Where δr_d, δs_d and m_d are the rudder, sternplane and motor commands, respectively.

grees and depth in metres. See Table 2, although the header rows are an addition here.

Conversely, there is a direct (or fixed controls) routine that allows motor and fin commands to be specified directly. In this case, the sense of the columns is still the same with the first one specifying the time of the commands, the second one the rudder positions, the third the sternplane positions and the last the motor commands. The angles are in degrees and the motor command is from -2100 to 2100 with 0 being no motion. See Table 3, again the header rows should not be present in the actual file.

By default, the command file is named CMDS.DAT and should be in the same directory as the program. There can be up to ten commands with the final one not being activated but serving to mark the end of the list. Time can be specified to 0.01s; other commands may be specified to **float** precision.

1.2.2 Options File

The options header file is called OPT.H and is shown in Table 4 as well as appearing in the listings in Part II. In order, the options are:

- DISTURBANCE — a water current disturbance can be included in the simulation model. The exact nature of the disturbance is specified elsewhere in the program.

- MTR_TIME_DELAY — due to the communications setup of the vehicle, a time delay is present between issuing commands and having the actuators respond. This is the time delay for the motor/ speed system; units are 0.01s so the delay of 15 is 0.15s.

- RUD_TIME_DELAY and SPL_TIME_DELAY are the delays for the rudder and sternplane systems respectively. Again in units of 0.01s.

- DODGY_FINS — the fins on the vehicle exhibit backlash and gen-

Table 4: OPT.H — simulation options file.

```
#define YS TRUE
#define NO FALSE

#define DISTURBANCE        FALSE
#define MTR_TIME_DELAY     15
#define RUD_TIME_DELAY     70
#define SPL_TIME_DELAY     40
#define DODGY_FINS         FALSE
#define SENSOR_REAL        TRUE
#define TETHER_DYNAMICS    FALSE
#define BENDING            FALSE
#define SL_TETHER_DRAG     FALSE
#define CHANGE_PARAMS      FALSE
#define CONTROL_TYPE       SLIDING_MODE

#define PRINT_COURSE       YS
#define PRINT_RUDDER       YS
#define PRINT_DEPTH        YS
#define PRINT_Z_DOT        NO
#define PRINT_PTCH_D       NO
#define PRINT_PITCH        YS
#define PRINT_Q            NO
#define PRINT_STERNP       YS
#define PRINT_SPEED        YS
#define PRINT_U_DOT        NO
#define PRINT_RPS          NO
#define PRINT_MTRCMD       YS
#define PRINT_STR          NO
```

eral play in the systems. If TRUE, this models this effect.

- SENSOR_REAL — the sensors on the vehicle do not report the vehicle's motions exactly as each sensor is affected by noise. If TRUE, this models this effect; if FALSE then the sensor data is equal to the accurate velocities and positions generated by the simulation.

- TETHER_DYNAMICS — if TRUE, then the manoeuvring tether model is used. This models the tether as a 20m long streamer with 20 nodes.

- BENDING — a subset of the tether model, this specifies whether to included bending moments due to the cable in the tether model. If FALSE, then only hydrodynamic forces act on the tether.

- SL_TETHER_DRAG — a simple tether model in which the vehicle is assumed to be travelling in a straight line with more and more cable being pulled into the water as the vehicle travels. Intended for investigation into speed controllers.

- CHANGE_PARAMS — if TRUE, then the vehicle's mass, trim or other characteristics may be changed partway through the simulation run at a point determined elsewhere in the program.

- CONTROL_TYPE — specifies the autopilot that is to be used. Options are PID for a PID autopilot, FUZZY for fuzzy logic, SLIDING_MODE for sliding mode, STR for self-tuning and FIXED for direct motor and fin commands.

- PRINT_XXX — the final block of options determine the data that is printed on-screen when the program is run. In order, the options are for heading, rudder position, depth, rate of change of depth, demanded pitch, pitch angle, pitch rate, sternplane position, speed, acceleration, propeller speed, motor command and self-tuner variables.

1.2.3 Outputs

Three outputs are produced by the simulation — one on screen and two files to disk. The screen output consists of the variables specified in OPT.H and by default occurs every 0.1s.

The first disk file, ROV.OUT contains data on all pertinent vehicle states and commands. Output as text delimited by commas and spaces, it is 41 columns wide and consists of the following data in order:

- Time, speed, acceleration, sway speed, heave speed, roll rate, pitch rate, yaw rate, depth, heading, pitch, roll, speed, N/A, N/A, propeller speed, motor command, rudder command, N/A, N/A, sternplane command, speed command, heading command, depth command, speed, KF sway speed estimate, KF yaw rate estimate, KF heading estimate, sensor yaw rate, sensor heading, KF acceleration estimate, KF speed estimate, sensor acceleration, sensor speed, KF heave speed estimate, KF pitch rate estimate, KF pitch estimate, KF depth estimate, sensor pitch rate, sensor pitch and sensor depth.

When the experimental vehicle is being run, it produces a similar data file. The N/A fields are here to ensure compatibility, but record items such as communications problems that are not relevant to the simulation program. The KF fields give the results from the three Kalman filters that estimate parameters for the speed, heading and depth systems. The sensor fields contain the (possibly) noisy sensor data that was generated by the simulation.

The second file, TETHER.OUT contains the positions of the tether at each 0.1s time interval. Again divided into columns, the first field contains the vehicle's x-position with the next 20 fields being the x-positions of each of the tether nodes. Following those is the y-position of the centre of the vehicle and then the y-positions of each node.

1.2.4 Operation

Essentially, the simulation program is operated from the compiler. Depending on the compiler being used this may involve an integrated development environment, or the header and command files may be edited using an external editor then passed to a command line program for compilation.

2

Simulation Models

This chapter details the various models that are used in the simulation program, namely:

- The coordinate system.
- The six degree-of-freedom vehicle model that uses rigid-body mechanics together with a model of the hydrodynamic forces and moments on the vehicle to simulate its motion through the water.
- The tether model for manoeuvring that models the tether as a fixed-length streamer in the horizontal plane.
- The propeller model that details the thrust and torque developed by the propeller for varying propeller and vehicle speeds.
- The motor model that specifies the motor speed and dynamics given varying commands.
- The control surface actuator models that map the dynamic position of the rudder and sternplane for varying fin positions and commands.

Whilst brief details are given of the various coefficients used in the simulation at the time of writing, little or no justification is given for their specific values. The interested reader is advised to consult Lea [3] where full descriptions may be found.

2.1 The Coordinate System

The AutoROV simulation program the motion of an underwater vehicle in six degrees of freedom (DOF), since six independent coordinates are required to determine the position and orientation of a rigid body in three dimensions. The first three coordinates and their time derivatives represent the translational position and motion of the body while the last three describe the rotational position and motion.

When considering a dynamic model of an underwater vehicle, it is convenient to use two coordinate systems: a global (earth-fixed) coordinate system or frame XYZ and a body-fixed system $X_0Y_0Z_0$ as shown in Figure 3. In terms of vehicle position and motion, the earth-fixed system is the frame of interest whereas the equations describing the vehicle's

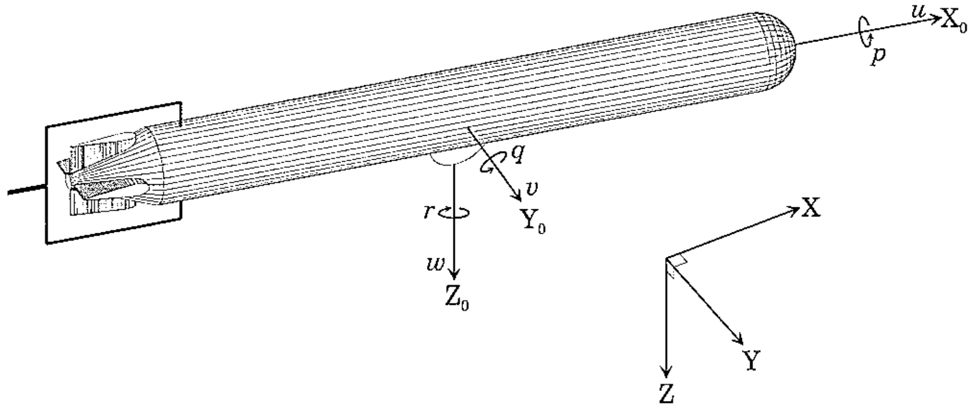


Figure 3: Coordinate systems.

behaviour are more easily developed in the body-fixed system.

The position of the vehicle cannot be expressed in the vehicle-fixed system, as position relative to itself is meaningless. Instead, the velocity of the vehicle is described as a vector $\boldsymbol{v}$:

$$\boldsymbol{v} = [u \quad v \quad w \quad p \quad q \quad r]^T \tag{1}$$

where p, q and r are rotations around the three axes X_0, Y_0 and Z_0 respectively (Figure 3). The six different motions corresponding to movements in the respective coordinates are *surge*, *sway*, *heave*, *roll*, *pitch* and *yaw*.

Given that the three rotational components of the body-fixed frame are defined as rotations around the three axes, a reasonable definition for their counterparts in the global frame are an equivalent set of rotations around axes X, Y and Z. Indeed this representation is used, and the three angles are known as the Euler angles ϕ, θ and ψ, and are also referred to as *roll*, *pitch* and *yaw* (or *heading*).

Thus using Euler angles the position and orientation of the vehicle may be described as a vector η relative to the earth-fixed frame.

$$\eta = [x \quad y \quad z \quad \phi \quad \theta \quad \psi]^T \tag{2}$$

To transform between the two coordinate systems, the Euler angle mapping $\boldsymbol{J}$ is used:

$$\dot{\eta} = \boldsymbol{J}\boldsymbol{v} \tag{3}$$

where

$$J(\eta) = \begin{bmatrix} c\psi\, c\theta & -s\psi\, c\theta + c\psi\, s\theta\, s\phi & s\psi\, s\phi + c\psi\, c\phi\, s\theta & 0 & 0 & 0 \\ s\psi\, c\theta & c\psi c\phi + s\phi s\theta s\psi & -c\psi s\phi + s\theta s\psi c\phi & 0 & 0 & 0 \\ -s\theta & c\theta s\phi & c\theta c\phi & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & s\phi t\theta & c\phi t\theta \\ 0 & 0 & 0 & 0 & c\phi & -s\phi \\ 0 & 0 & 0 & 0 & \frac{s\phi}{c\theta} & \frac{c\phi}{c\theta} \end{bmatrix} \tag{4}$$

and $c\alpha$=cos(α), $s\alpha$=sin(α) and $t\alpha$=tan(α).

Thus we have the body-fixed frame in which the vehicle dynamics are expressed, and the global frame in which the vehicle's behaviour is described. $\boldsymbol{J}$ allows effects in one coordinate system to be mapped to the other. See Lea [3] or Fossen [4] for a more comprehensive treatment of the above.

2.2 Hydrodynamic Vehicle Model

In 1979, Feldman [5] published standard equations of motions for a torpedo-shaped vehicle, and the model used by the simulation is based on those equations. Presented below are the six nonlinear equations of motion that it uses.

Each coefficient (e.g. X'_{qq}) is non-dimensional, hence the expressions involving density and vehicle length. There are 63 coefficients in total — see Table 5. Most coefficients are constant, however the vehicle's drag coefficient X'_{uu} was found to vary with vehicle speed.

Some coefficients were obtained by tank tests, however the remainder were scaled from a similarly-shaped vehicle and thus should not be viewed as definitive for the *Subzero II* vehicle. See Lea [3] for more details.

The equation of motion for surge:

$$\begin{aligned} \left(m - \frac{\rho}{2} l^3 X'_{\dot{u}}\right)\dot{u} + m z_G \dot{q} - m y_G \dot{r} = {} & \frac{\rho}{2} l^4 \left[X'_{qq} q^2 + X'_{rr} r^2 + X'_{rp} rp\right] \\ & + \frac{\rho}{2} l^3 \left[X'_{vr} vr + X'_{wq} wq\right] \\ & + \frac{\rho}{2} l^2 \left[X'_{vv} v^2 + X'_{ww} w^2\right] \\ & + \frac{\rho}{2} l^2 \left[X'_{\delta r \delta r} \delta r^2 + X'_{\delta s \delta s} \delta s^2\right] u^2 \\ & - (W - B)\sin\theta + F_{prop} - \frac{\rho}{2} l^2 X'_{uu} u^2 \\ & + m\left[vr - wq + x_G\left(q^2 + r^2\right) - y_G qp - z_G rp\right] \end{aligned} \tag{5}$$

Table 5: *Subzero II* coefficients.

Rigid Body Data

$\rho = 1000.0\text{kg/m}^3$	$m = 7.0\text{kg}$	$W = 69.0\text{N}$	$B = 69.0\text{N}$
$len = 0.98\text{m}$	$I_{xx} = 0.006\text{kgm}^2$	$I_{yy} = 0.3\text{kgm}^2$	$I_{zz} = 0.3\text{kgm}^2$
$I_{xy} = 0.0\text{kgm}^2$	$I_{yz} = 0.0\text{kgm}^2$	$I_{zx} = 0.009\text{kgm}^2$	$x_G = 0.055\text{m}$
$y_G = 0.0\text{m}$	$z_G = 0.006\text{m}$	$x_B = 0.055\text{m}$	$y_B = 0.0\text{m}$
$z_B = 0.0\text{m}$			

Non-Dimensional Hydrodynamic Data

$X'_{ww} = -1.38\times10^{-2}$	$X'_{qq} = -3.2\times10^{-3}$	$X'_{rr} = -3.2\times10^{-3}$	$X'_{rp} = 0.0$
$X'_{\dot{u}} = -1.77\times10^{4}$	$X'_{wq} = 3.02\times10^{-3}$	$X'_{vr} = -3.02\times10^{-3}$	$X'_{vv} = -1.38\times10^{-2}$
$X'_{\delta s\delta s} = -7.0\times10^{-3}$	$X'_{\delta r\delta r} = -7.0\times10^{-3}$		
$Y'_{\dot{p}} = 0.0$	$Y'_{\dot{r}} = 1.31\times10^{-4}$	$Y'_{pq} = 5.98\times10^{-5}$	$Y'_{\dot{v}} = -1.5\times10^{-2}$
$Y'_{p} = 0.0$	$Y'_{r} = 1.76\times10^{-2}$	$Y'_{wp} = 1.04\times10^{-2}$	$Y'_{v} = -3.68\times10^{-2}$
$Y'_{\delta r} = 2.16\times10^{-2}$	$Y'_{pp} = 0.0$	$Y'_{vv} = 0.0$	$Y'_{uu} = 0.0$
$Z'_{\dot{q}} = -1.31\times10^{-4}$	$Z'_{\dot{w}} = -1.07\times10^{-2}$	$Z'_{q} = -1.76\times10^{-2}$	$Z'_{vp} = -1.04\times10^{-2}$
$Z'_{w} = -3.68\times10^{-2}$	$Z'_{\delta s} = -2.16\times10^{-2}$	$Z'_{uu} = 0.0$	$Z'_{w\cdot} = 0.0$
$Z'_{ww} = 0.0$	$C_d = 1.9$		
$K'_{\dot{p}} = -2.64\times10^{-6}$	$K'_{\dot{r}} = 0.0$	$K'_{qr} = 0.0$	$K'_{\dot{v}} = 0.0$
$K'_{p} = -2.86\times10^{-4}$	$K'_{r} = 0.0$	$K'_{wp} = 0.0$	$K'_{pp} = 0.0$
$K'_{uu} = 0.0$	$K'_{vR} = 0.0$	$K'_{\delta r} = 0.0$	
$M'_{\dot{q}} = -7.71\times10^{-4}$	$M'_{rp} = 7.51\times10^{-4}$	$M'_{\dot{w}} = -1.31\times10^{-4}$	$M'_{q} = -8.37\times10^{-3}$
$M'_{vp} = -3.98\times10^{-5}$	$M'_{w} = -7.34\times10^{-3}$	$M'_{\delta s} = -1.02\times10^{-2}$	$M'_{uu} = 0.0$
$M'_{w\cdot} = 0.0$	$M'_{ww} = 0.0$		
$N'_{\dot{v}} = 1.31\times10^{-4}$	$N'_{\dot{p}} = 0.0$	$N'_{\dot{r}} = -7.71\times10^{-4}$	$N'_{pq} = -7.51\times10^{-4}$
$N'_{p} = 0.0$	$N'_{r} = -8.37\times10^{-3}$	$N'_{uu} = 0.0$	$N'_{v} = 7.34\times10^{-3}$
$N'_{vR} = 0.0$	$N'_{\delta r} = -1.02\times10^{-2}$		

$$
\begin{aligned}
X'_{uu} = &-3.12661\times 0.5 + 1.44963\cos(u) + 2.27372\sin(u) \\
&+ 0.71716\cos(2u) - 1.53679\sin(2u) - 0.73919\cos(3u) + 0.09957\sin(3u) \\
&+ 0.13634\cos(4u) + 0.15798\sin(4u) - 0.00735\cos(5u) - 0.02748\sin(5u)
\end{aligned}
$$

The equation of motion for sway:

$$
\begin{aligned}
&\left(m-\frac{\rho}{2}l^3Y'_{\dot{v}}\right)\dot{v}-\left(mz_G+\frac{\rho}{2}l^4Y'_{\dot{p}}\right)\dot{p}+\left(mx_G-\frac{\rho}{2}l^4Y'_{\dot{r}}\right)\dot{r}= \\
&\qquad \frac{\rho}{2}l^4\left[Y'_{p|p|}p|p|+Y'_{pq}pq\right] \\
&\qquad +\frac{\rho}{2}l^3\left[Y'_{r}ur+Y'_{p}up+Y'_{wp}wp\right] \\
&\qquad +\frac{\rho}{2}l^2\left[Y'_{uu}u^2+Y'_{v}uv+Y'_{vv}v\sqrt{v^2+w^2}\right] \\
&\qquad +\frac{\rho}{2}l^2Y'_{\delta r}u^2\delta r \\
&\qquad +(W-B)\cos\theta\sin\phi \\
&\qquad -\frac{\rho}{2}C_d\int_{x_{tail}}^{x_{nose}}y(x)(v+xr)\sqrt{(w-xq)^2+(v+xr)^2}\,dx \\
&\qquad +m\left[wp-ur+y_G\left(r^2+p^2\right)-z_Gqr-x_Gqp\right]
\end{aligned}
\tag{6}
$$

The equation of motion for heave:

$$
\begin{aligned}
&\left(m-\frac{\rho}{2}l^3Z'_{\dot{w}}\right)\dot{w}+my_G\dot{p}-\left(mx_G+\frac{\rho}{2}l^4Z'_{\dot{q}}\right)\dot{q}=\frac{\rho}{2}l^3\left[Z'_{q}uq+Z'_{vp}vp\right] \\
&\qquad +\frac{\rho}{2}l^2\left[Z'_{uu}u^2+Z'_{w}uw\right] \\
&\qquad +\frac{\rho}{2}l^2\left[Z'_{w*}u|w|+Z'_{ww}\left|w\sqrt{v^2+w^2}\right|\right] \\
&\qquad +\frac{\rho}{2}l^2Z'_{\delta s}u^2\delta s \\
&\qquad +(W-B)\cos\theta\cos\phi \\
&\qquad -\frac{\rho}{2}C_d\int_{x_{tail}}^{x_{nose}}y(x)(w-xq)\sqrt{(w-xq)^2+(v+xr)^2}\,dx \\
&\qquad +m\left[uq-vp+z_G\left(p^2+q^2\right)-x_Grp-y_Grq\right]
\end{aligned}
\tag{7}
$$

The equation of motion for roll:

$$
\begin{aligned}
&-\left(mz_G+\frac{\rho}{2}l^4K'_{\dot{v}}\right)\dot{v}+my_G\dot{w}+\left(I_x-\frac{\rho}{2}l^5K'_{\dot{p}}\right)\dot{p}-I_{xy}\dot{q}-\left(I_{zx}+\frac{\rho}{2}l^5K'_{\dot{r}}\right)\dot{r} \\
&\qquad =\frac{\rho}{2}l^5\left[K'_{qr}qr+K'_{pp}p|p|\right] \\
&\qquad +\frac{\rho}{2}l^4\left[K'_{p}up+K'_{r}ur+K'_{wp}wp\right] \\
&\qquad +\frac{\rho}{2}l^3\left[K'_{uu}u^2+K'_{vR}uv\right] \\
&\qquad +\frac{\rho}{2}l^3K'_{\delta r}u^2\delta r \\
&\qquad +\left(y_GW-y_BB\right)\cos\theta\sin\phi+\frac{\rho}{2}l^3K'_{prop}n^2 \\
&\qquad -\left(I_z-I_y\right)qr+I_{zx}qp-\left(r^2-q^2\right)I_{yz}-I_{xy}pr \\
&\qquad +m\left[y_G(uq-vp)-z_G(wp+ur)\right]
\end{aligned}
\tag{8}
$$

The equation of motion for pitch:

$$
\begin{aligned}
mz_G\dot{u} - \left(mx_G + \frac{\rho}{2}l^4M'_{\dot{w}}\right)\dot{w} - I_{xy}\dot{p} + \left(I_y - \frac{\rho}{2}l^5M'_{\dot{q}}\right)\dot{q} - I_{yz}\dot{r} \\
&= \frac{\rho}{2}l^5M'_{rp}rp + \frac{\rho}{2}l^4M'_q uq \\
&+ \frac{\rho}{2}l^3\left[M'_{uu}u^2 + M'_w uw + M'_{wwR}w\sqrt{v^2+w^2}\right] \\
&+ \frac{\rho}{2}l^3\left[M'_{w*}u|w| + M'_{ww}\left|w\sqrt{v^2+w^2}\right|\right] \\
&+ \frac{\rho}{2}l^3M'_{\delta s}u^2\delta s \\
&+ \frac{\rho}{2}C_d\int_{x_{tail}}^{x_{nose}} y(x)(w-xq)\sqrt{(w-xq)^2+(v+xr)^2}\,xdx \\
&- (x_GW - x_BB)\cos\theta\cos\phi - (z_GW - z_BB)\sin\theta \\
&- (I_x - I_z)rp + I_{xy}qr - (p^2 - r^2)I_{zx} - I_{yz}qp \\
&- m[z_G(wq - vr) + x_G(uq - vp)]
\end{aligned}
\tag{9}
$$

The equation of motion for yaw:

$$
\begin{aligned}
-my_G\dot{u} + \left(mx_G - \frac{\rho}{2}l^4N'_{\dot{v}}\right)\dot{v} - \left(I_{zx} + \frac{\rho}{2}l^5N'_{\dot{p}}\right)\dot{p} - I_{yz}\dot{q} + \left(I_z - \frac{\rho}{2}l^5N'_{\dot{r}}\right)\dot{r} \\
&= \frac{\rho}{2}l^5N'_{pq}pq + \frac{\rho}{2}l^4\left[N'_p up + N'_r ur\right] \\
&+ \frac{\rho}{2}l^3\left[N'_{uu}u^2 + N'_v uv + N'_{vvR}v\sqrt{v^2+w^2}\right] \\
&+ \frac{\rho}{2}l^3N'_{\delta r}u^2\delta r \\
&- \frac{\rho}{2}C_d\int_{x_{tail}}^{x_{nose}} y(x)(v+xr)\sqrt{(w-xq)^2+(v+xr)^2}\,xdx \\
&- (x_GW - x_BB)\cos\theta\sin\phi + (y_GW - y_BB)\sin\theta \\
&- (I_y - I_x)pq + I_{yz}rp - (q^2 - p^2)I_{xy} - I_{zx}rq \\
&+ m[x_G(wp - ur) - y_G(vr - wq)]
\end{aligned}
\tag{10}
$$

2.3 Tether Model

The two-dimensional tether model used in this work is based on that given by Howell in his doctoral dissertation [6]. Howell's model was also two-dimensional, but was for the vertical plane whereas the model here is used in the horizontal plane. Thus gravity effects are ignored, as are the effects due to water currents which are present in Howell's work.

It should be noted that although the equations of motion given by Howell are correct, there are a number of sign errors in the finite-difference method as presented in the dissertation. These have been corrected here. Howell also uses numerical damping (viscosity) which was not found to be needed once the sign errors were corrected.

2.3.1 Equations of Motion

The fundamental equations of motion for an inextensible tether are:

$$
\begin{aligned}
m\left(\frac{\partial u}{\partial t}-\frac{\partial \varphi}{\partial t}v\right) &= \frac{\partial T}{\partial s}-\frac{1}{2}\rho d\pi C_{dt}u^2+EI\frac{\partial \varphi}{\partial s}\frac{\partial^2 \varphi}{\partial s^2} \\
m\left(\frac{\partial v}{\partial t}+\frac{\partial \varphi}{\partial t}u\right)+m_a\frac{\partial v_r}{\partial t} &= T\frac{\partial \varphi}{\partial t}-\frac{1}{2}\rho dC_{dn}v^2-EI\frac{\partial^3 \varphi}{\partial s^3} \\
\frac{\partial u}{\partial s}-\frac{\partial \varphi}{\partial s}v &= 0 \\
\frac{\partial v}{\partial s}+\frac{\partial \varphi}{\partial s}u &= \frac{\partial \varphi}{\partial t}
\end{aligned}
\tag{11}
$$

where u is the tangential velocity of the cable, v is the normal velocity, T is the tension, m is the mass per unit length, m_a is the added mass per unit length, d is the diameter of the cable, φ is the angle of the cable, t is time and s is distance along the cable. See Figure 4.

To begin with, we shall ignore the bending moment terms, i.e. the terms in the first two equations beginning EI...

Table 6 gives details of finite difference methods to find the derivative of a function f at a point x. (The expressions are routine and can be found in a standard book on finite difference methods.) It can be seen that the central-difference method is preferable for the majority of the solution as it requires less computation than the other two methods for the same degree of accuracy — the function need be evaluated at only two points compared to the three needed by the others. The first central difference algorithm available involves an error of h^2, so this degree of accuracy will be used throughout the solution.

Using the forward-difference method for the first node, the central-difference method for the middle nodes and the backward-difference

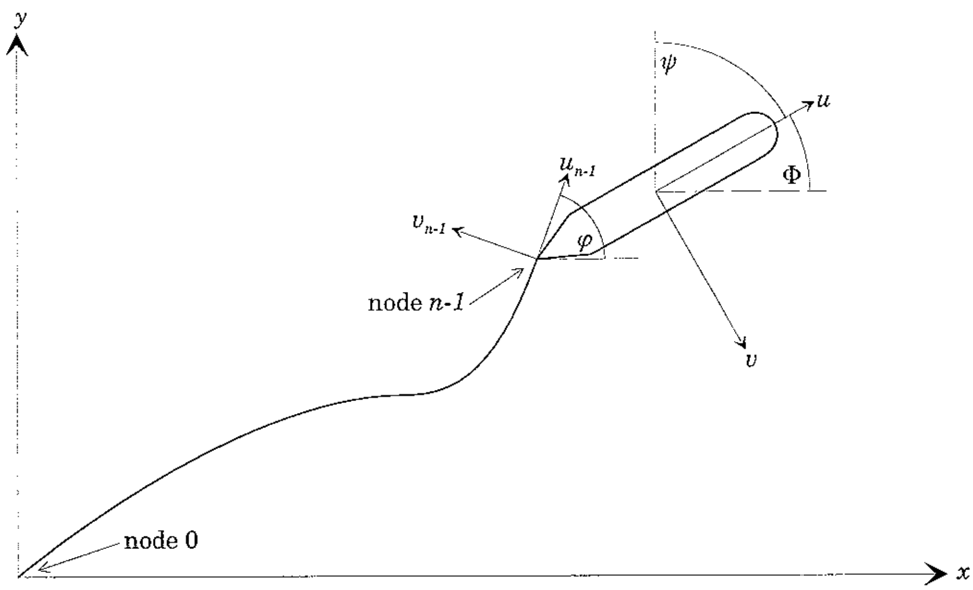


Figure 4: Tether system coordinates and layout.

Table 6: Finite differences to calculate $f'(x)$.

Error	Forwards	Central	Backwards
O(h)	$\frac{f_0 - f_1}{h}$	–	$\frac{-f_{-1} + f_0}{h}$
O(h²)	$\frac{-3f_0 + 4f_1 - f_2}{2h}$	$\frac{-f_{-1} + f_1}{2h}$	$\frac{f_{-2} - 4f_{-1} + 3f_0}{2h}$
O(h³)	$\frac{-11f_0 + 18f_1 - 9f_2 + 2f_3}{6h}$	–	$\frac{-2f_{-3} + 9f_{-2} - 18f_{-1} + 11f_0}{6h}$

h is the inter-node spacing, i.e. $f_0=f(x)$, $f_1=f(x+h)$, $f_{-1}=f(x-h)$, etc. The error column indicates the most significant power of h contributing to the error.

method for the last node, we obtain the following numerical algorithm for updating the node angles (note the convention x_{node}^{time}):

$$\begin{aligned}
\varphi_0^{i+1} &= \varphi_0^i - \lambda_r\left[v_2^i - 4v_1^i + 3v_0^i + u_0^i\left(\varphi_2^i - 4\varphi_1^i + 3\varphi_0^i\right)\right] \\
\varphi_j^{i+1} &= \varphi_j^i + \lambda_r\left[v_{j+1}^i - v_{j-1}^i + u_j^i\left(\varphi_{j+1}^i - \varphi_{j-1}^i\right)\right] \\
\varphi_{n-1}^{i+1} &= \varphi_{n-1}^i + \lambda_r\left[v_{n-3}^i - 4v_{n-2}^i + 3v_{n-1}^i + u_{n-1}^i\left(\varphi_{n-3}^i - 4\varphi_{n-2}^i + 3\varphi_{n-1}^i\right)\right]
\end{aligned} \tag{12}$$

To update the tangential velocities:

$$\begin{aligned}
u_0^{i+1} &= u_0^i + v_0^i\left(\varphi_0^{i+1} - \varphi_0^i\right) + \frac{\lambda_r}{m}\left(-T_2^i + 4T_1^i - 3T_0^i\right) - \frac{\Delta t}{2m}\rho d\pi C_{dt} u_0^i\left|u_0^i\right| \\
u_j^{i+1} &= u_j^i + v_j^i\left(\varphi_j^{i+1} - \varphi_j^i\right) + \frac{\lambda_r}{m}\left(T_{j+1}^i - T_{j-1}^i\right) - \frac{\Delta t}{2m}\rho d\pi C_{dt} u_j^i\left|u_j^i\right| \\
u_{n-1}^{i+1} &= u\cos\left(\varphi_{n-1} - \Phi\right) - (v - 0.5r)\sin\left(\varphi_{n-1} - \Phi\right)
\end{aligned} \tag{13}$$

Note the correction for u_{n-1} in that the tether is connected to the rear of the vehicle, but u and v for the vehicle are specified at the vehicle's centre and thus an allowance must be made for yaw rate.

To update the normal velocities:

$$\begin{aligned}
v_0^{i+1} &= v_0^i - \frac{m}{m_1}u_0^i\left(\varphi_0^{i+1} - \varphi_0^i\right) - \frac{\lambda_r}{m_1}T_0^i\left(\varphi_2^i - 4\varphi_1^i + 3\varphi_0^i\right) - \frac{\Delta t}{2m_1}\rho d\pi C_{dn} v_0^i\left|v_0^i\right| \\
v_j^{i+1} &= v_j^i - \frac{m}{m_1}v_j^i\left(\varphi_j^{i+1} - \varphi_j^i\right) + \frac{\lambda_r}{m_1}T_j^i\left(\varphi_{j+1}^i - \varphi_{j-1}^i\right) - \frac{\Delta t}{2m_1}\rho d\pi C_{dn} v_j^i\left|v_j^i\right| \\
v_{n-1}^{i+1} &= -u\sin\left(\varphi_{n-1} - \Phi\right) - (v - 0.5r)\cos\left(\varphi_{n-1} - \Phi\right)
\end{aligned} \tag{14}$$

And for keeping the cable in equilibrium:

$$\begin{aligned}
T_0^i &= 0 \\
u_{j+1}^{i+1} - u_{j-1}^{i+1} &= v_j^{i+1}\left(\varphi_{j+1}^{i+1} - \varphi_{j-1}^{i+1}\right) \\
u_{n-3}^{i+1} - 4u_{n-2}^{i+1} + 3u_{n-1}^{i+1} &= v_{n-1}^{i+1}\left(\varphi_{n-3}^{i+1} - 4\varphi_{n-2}^{i+1} + 3\varphi_{n-1}^{i+1}\right)
\end{aligned} \tag{15}$$

where

$$\begin{aligned}
\lambda_r &= \frac{\Delta t}{2\Delta s} \\
m_1 &= m + m_a = m + \frac{\pi}{4}\rho d^2
\end{aligned} \tag{16}$$

Δt is the time interval, Δs is the spatial interval (i.e. the distance between nodes on the tether) and m_a is the added mass.

2.3.2 Solution Method

Here, the algorithm for solution of the tether equations will be presented.

The values of φ^{i+1} can be found directly from φ^i, $\boldsymbol{u}^i$ and $\boldsymbol{v}^i$ using (12). However, the other equations are coupled and thus cannot be resolved explicitly. To solve them, we begin by rewriting the finite difference equations (13)-(15) in matrix form as

$$\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix}^{i+1} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_{\substack{connection\\point}} \end{bmatrix}^{i} + \frac{1}{2}\begin{bmatrix} \varphi_0^{i+1}-\varphi_0^i & 0 & 0 & 0 & 0 \\ 0 & \varphi_1^{i+1}-\varphi_1^i & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \varphi_{n-2}^{i+1}-\varphi_{n-2}^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}^{i}$$
$$+\frac{\lambda_r}{m}\begin{bmatrix} -3 & 4 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & \ddots & 0 & \ddots & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 0 \\ T_1 \\ \vdots \\ T_{n-2} \\ T_{n-1} \end{bmatrix}^{i} - \frac{\Delta t}{2m}\rho D\pi Cd_t\begin{bmatrix} u_0^i\left|u_0^i\right| \\ u_1^i\left|u_1^i\right| \\ \vdots \\ u_{n-2}^i\left|u_{n-2}^i\right| \\ 0 \end{bmatrix} \tag{17}$$

(14) in matrix form:

$$\begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}^{i+1} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{\substack{connection\\point}} \end{bmatrix}^{i} - \frac{m}{2m_1}\begin{bmatrix} \varphi_0^{i+1}-\varphi_0^i & 0 & 0 & 0 & 0 \\ 0 & \varphi_1^{i+1}-\varphi_1^i & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \varphi_{n-2}^{i+1}-\varphi_{n-2}^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix}^{i}$$
$$+\frac{\lambda_r}{m_1}\begin{bmatrix} -\varphi_2^i+4\varphi_1^i-3\varphi_0^i & 0 & 0 & 0 & 0 \\ 0 & \varphi_2^i-\varphi_0^i & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \varphi_{n-1}^i-\varphi_{n-3}^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 0 \\ T_1 \\ \vdots \\ T_{n-2} \\ T_{n-1} \end{bmatrix}^{i}$$
$$-\frac{\Delta t}{2m_1}\rho DCd_n\begin{bmatrix} v_0^i\left|v_0^i\right| \\ v_1^i\left|v_1^i\right| \\ \vdots \\ v_{n-2}^i\left|v_{n-2}^i\right| \\ 0 \end{bmatrix} \tag{18}$$

and (15):

$$
\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & \ddots & 0 & \ddots & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & -4 & 3 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix}^{i+1} =
$$

$$
\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \varphi_2^{i+1} - \varphi_0^{i+1} & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \varphi_{n-1}^{i+1} - \varphi_{n-3}^{i+1} & 0 \\ 0 & 0 & 0 & 0 & \varphi_{n-3}^{i+1} - 4\varphi_{n-2}^{i+1} + 3\varphi_{n-1}^{i+1} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}^{i+1} \tag{19}
$$

The unknowns to be solved for are the tangential velocities $\boldsymbol{u}^{i+1}$, the normal velocities $\boldsymbol{v}^{i+1}$ and the tensions at each node $\boldsymbol{t}^{i+1}$. Thus the above three equations can be rewritten as

$$\boldsymbol{u}^{i+1} = \boldsymbol{a} + \boldsymbol{B}\boldsymbol{t}^{i+1} \tag{20}$$

$$\boldsymbol{v}^{i+1} = \boldsymbol{c} + \boldsymbol{D}\boldsymbol{t}^{i+1} \tag{21}$$

$$\boldsymbol{E}\boldsymbol{u}^{i+1} = \boldsymbol{G}\boldsymbol{v}^{i+1} \tag{22}$$

where

$$
\boldsymbol{a} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_{\substack{connection \\ point}} \end{bmatrix}^i + \frac{1}{2} \begin{bmatrix} \varphi_0^{i+1} - \varphi_0^i & 0 & 0 & 0 & 0 \\ 0 & \varphi_1^{i+1} - \varphi_1^i & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \varphi_{n-2}^{i+1} - \varphi_{n-2}^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}^i
$$

$$
- \frac{\Delta t}{2m} \rho D \pi C d_t \begin{bmatrix} u_0^i \left| u_0^i \right| \\ u_1^i \left| u_1^i \right| \\ \vdots \\ u_{n-2}^i \left| u_{n-2}^i \right| \\ 0 \end{bmatrix} \tag{23}
$$

$$
\boldsymbol{B} = \frac{\lambda_r}{m} \begin{bmatrix} -3 & 4 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & \ddots & 0 & \ddots & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{24}
$$

$$
\boldsymbol{c} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-2} \\ v_{\substack{connection \\ point}} \end{bmatrix}^i - \frac{m}{2m_1} \begin{bmatrix} \varphi_0^{i+1} - \varphi_0^i & 0 & 0 & 0 & 0 \\ 0 & \varphi_1^{i+1} - \varphi_1^i & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \varphi_{n-2}^{i+1} - \varphi_{n-2}^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_{n-1} \end{bmatrix}^i \tag{25}
$$

$$
- \frac{\Delta t}{2m_1} \rho D C d_n \begin{bmatrix} v_0^i |v_0^i| \\ v_1^i |v_1^i| \\ \vdots \\ v_{n-2}^i |v_{n-2}^i| \\ 0 \end{bmatrix}
$$

and

$$
\boldsymbol{D} = \begin{bmatrix} -\varphi_2^i + 4\varphi_1^i - 3\varphi_0^i & 0 & 0 & 0 & 0 \\ 0 & \varphi_2^i - \varphi_0^i & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \varphi_{n-1}^i - \varphi_{n-3}^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{26}
$$

The composition of $\boldsymbol{E}$ and $\boldsymbol{G}$ should be obvious if (19) and (22) are compared.

Thus by substituting (20) and (21) into (22) we obtain

$$
\begin{aligned} \boldsymbol{E}(\boldsymbol{a} + \boldsymbol{B}\boldsymbol{t}) &= \boldsymbol{G}(\boldsymbol{c} + \boldsymbol{D}\boldsymbol{t}) \\ \therefore \boldsymbol{E}\boldsymbol{a} + \boldsymbol{E}\boldsymbol{B}\boldsymbol{t} &= \boldsymbol{G}\boldsymbol{c} + \boldsymbol{G}\boldsymbol{D}\boldsymbol{t} \end{aligned} \tag{27}
$$

The i+1 superscripts indicating the next time step have been dropped. This can be rearranged to give a solution to t:

$$
\begin{aligned} (\boldsymbol{EB} - \boldsymbol{GD})\boldsymbol{t} &= (\boldsymbol{Gc} - \boldsymbol{Ea}) \\ \Rightarrow \boldsymbol{t} &= (\boldsymbol{EB} - \boldsymbol{GD})^{-1}(\boldsymbol{Gc} - \boldsymbol{Ea}) \end{aligned} \tag{28}
$$

Once a value for $\boldsymbol{t}^{i+1}$ has been found this can be substituted into (20) and (21) to find $\boldsymbol{u}^{i+1}$ and $\boldsymbol{v}^{i+1}$.

2.3.3 Tether Model with Bending Moments

To include the effects due to bending moments in the tether model, the terms neglected in (11) need to be included. These are

$$
EI \frac{\partial \varphi}{\partial s} \frac{\partial^2 \varphi}{\partial s^2} = \begin{cases} EI \dfrac{\lambda_r}{m(\Delta s)^2} \left(-\varphi_2^i + 4\varphi_1^i - 3\varphi_0^i\right)\left(-\varphi_3^i + 4\varphi_2^i - 5\varphi_1^i + 3\varphi_0^i\right) & j = 0 \\ EI \dfrac{\lambda_r}{m(\Delta s)^2} \left(\varphi_{j+1}^i - \varphi_{j-1}^i\right)\left(\varphi_{j+1}^i - 2\varphi_j^i + \varphi_{j-1}^i\right) & 1 \le j \le n-2 \end{cases} \tag{29}
$$

Table 7: Finite differences to calculate $f'(x)$.

	Forwards	Central	Backwards
$\frac{\partial f}{\partial x}$	$\frac{-3f_0+4f_1-f_2}{2h}$	$\frac{-f_{-1}+f_1}{2h}$	$\frac{f_{-2}-4f_{-1}+3f_0}{2h}$
$\frac{\partial^2 f}{\partial x^2}$	$\frac{2f_0-5f_1+4f_2-f_3}{h^2}$	$\frac{f_{-1}-2f_0+f_1}{h^2}$	$\frac{f_{-3}-4f_{-2}+5f_{-1}-2f_0}{h^2}$
$\frac{\partial^3 f}{\partial x^3}$	$\frac{-5f_0+18f_1-24f_2+14f_3-3f_4}{2h^3}$	$\frac{-f_{-2}+2f_{-1}-2f_1+f_2}{2h^3}$	$\frac{3f_{-4}+4f_{-3}+24f_{-2}-18f_{-1}+5f_0}{2h^3}$

and

$$-EI\frac{\partial^3\varphi}{\partial s^3}=\begin{cases}\frac{-EI\lambda_r}{m_1(\Delta s)^2}\left(-3\varphi_4^i+14\varphi_3^i-24\varphi_2^i+18\varphi_1^i-5\varphi_0^i\right) & j=0\\ \frac{-EI\lambda_r}{m_1(\Delta s)^2}\left(\varphi_{j+2}^i-2\varphi_{j+1}^i+2\varphi_{j-1}^i-\varphi_{j-2}^i\right) & 1\le j\le n-3\\ \frac{-EI\lambda_r}{m_1(\Delta s)^2}\left(3\varphi_{n-6}^i-14\varphi_{n-5}^i+24\varphi_{n-4}^i-18\varphi_{n-3}^i+5\varphi_{n-2}^i\right) & j=n-2\end{cases} \tag{30}$$

(The finite difference expansions are those given in Table 7.) The above terms can be added directly to vectors $\boldsymbol{a}$ and $\boldsymbol{c}$ in (23) and (25) respectively and thus do not require a change in the solution method.

2.3.4 Effect on the Vehicle

The forces on the vehicle at the connection point due to tension can be broken down into surge and sway components on the vehicle:

$$\begin{aligned}F_{surge}&=-T_{n-1}\cos(\Phi-\varphi_{n-1})=-T_{n-1}\cos\left(\frac{\pi}{2}-\varphi_{n-1}-\psi\right)\\ F_{sway}&=-T_{n-1}\sin(\Phi-\varphi_{n-1})=-T_{n-1}\sin\left(\frac{\pi}{2}-\varphi_{n-1}-\psi\right)\end{aligned} \tag{31}$$

Similarly, the forces due to bending can be resolved into

$$\begin{aligned}F_{surge}&=EI\frac{\partial^2\varphi_{n-1}}{\partial s^2}\sin(\varphi_{n-1}-\Phi)\\ F_{sway}&=-EI\frac{\partial^2\varphi_{n-1}}{\partial s^2}\cos(\varphi_{n-1}-\Phi)\end{aligned} \tag{32}$$

where

$$\frac{\partial^2\varphi_{n-1}}{\partial s^2}=\frac{-\varphi_{n-4}^i+4\varphi_{n-3}^i-5\varphi_{n-2}^i+2\varphi_{n-1}^i}{(\Delta s)^2} \tag{33}$$

If the connection point is not mounted at the centre of mass of the vehicle, then the forces at the connection point will result in moments. In the case of *Subzero II* the tether is connected at the centre of the tail and thus, only the effects on yaw were considered. This was taken to be a moment equal to the sway force acting over the length from the tail to the centre of mass.

2.4 Propeller Model

The thrust produced by a propeller is in general a function of its speed of rotation n, and the velocity of the water flowing in to it. (This is normally affected by interference and other mounting considerations, but as the *Subzero II* propeller was tested on the vehicle, the inflow velocity will be taken as the vehicle's velocity u.)

In other words, a given propeller speed will produce differing amounts of thrust depending on whether the vehicle it is mounted on is stationary or moving forwards or backwards. To aid in modelling, a *quadrant* representation is used. The propeller can be rotating forwards or backwards, and the vehicle can be moving forwards or backwards through the water; the combination of these gives rises to four quadrants. In the first quadrant, both propeller and vehicle are moving forwards. In the second, the vehicle is moving forwards but the propeller is rotating backwards. In the third, the propeller is still rotating backwards and the vehicle is now moving backwards. In the fourth, the vehicle is still moving backwards but the propeller is moving forwards.

The advance angle β is defined as [7]:

$$\tan\beta = \frac{u}{0.7\pi nD} \tag{34}$$

and the thrust and torque developed by the propeller is given by

$$\begin{aligned} T &= C_T^* \frac{1}{2}\rho\left[u^2 + (0.7\pi nD)^2\right]\frac{\pi}{4}D^2 \\ Q &= C_Q^* \frac{1}{2}\rho\left[u^2 + (0.7\pi nD)^2\right]\frac{\pi}{4}D^3 \end{aligned} \tag{35}$$

where C_T^* is the thrust coefficient and C_Q^* the torque coefficient. Both these coefficients are found by tank tests. In the case of *Subzero II*, only thrust tests could be done thus the torque coefficient had to be extrapolated from data for other propellers.

The thrust and torque coefficients vary with the advance angle and it is standard to represent them by means of a 20 term Fourier series [7] such that

$$C(\beta) = \frac{a_0}{2} + \sum_{n=1}^{20}\left[a_n \cos(n\beta) + b_n \sin(n\beta)\right] \tag{36}$$

The data for this may be found in the listing of module INTS_ROV.C; Figure 5 shows the thrust coefficient.

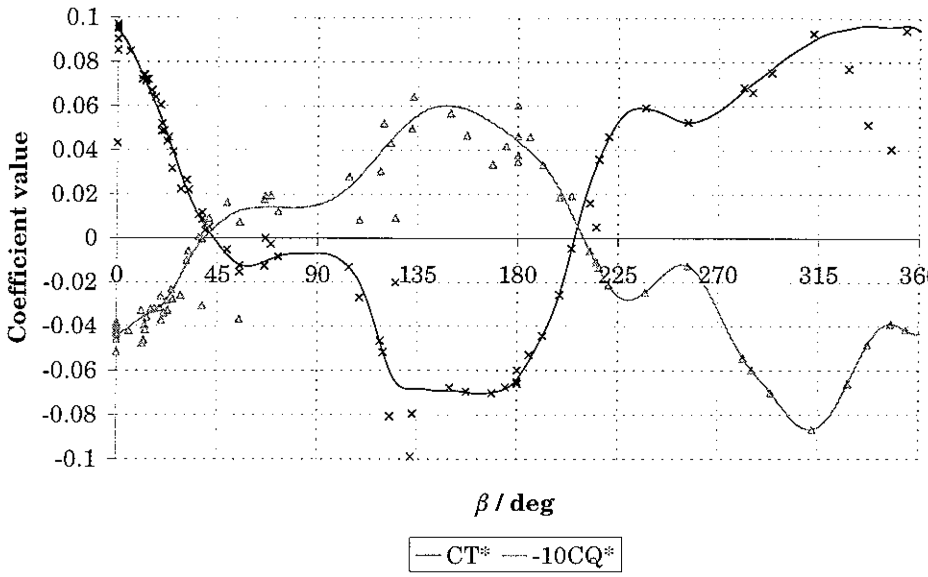


Figure 5: Four quadrant thrust coefficient.

2.5 Motor Model

As stated earlier, the motor in the vehicle is a permanent magnet DC motor and thus a standard DC motor model was used (see [8], for example). As the gearbox is an integral part of the motor, the 'shaft speed' was taken to be the propeller shaft speed, rather than the motor shaft speed.

The motor is controlled by a PWM drive generated by the ROV's micro-controller, and so the motor command entered at the PC is a signed number representing the cycle on-time in clock units. As the MCU runs at 2MHz and the PWM cycle runs at 800Hz, a command $m_d = 2500$ corresponds to a duty cycle of 1. In practice, MCU processing overheads means that the motor command ranges from -2100 to 2100 with 0 being no movement.

2.5.1 Motor Equations

The steady-state equations are (with allowances for the resistance of the drive FETs):

$$\begin{aligned} \omega &= 2\pi \cdot n \\ V_s &= \left(R_a + R_f\right)I_a + V_b + E \\ E &= k_\phi \omega \\ Q &= k_\phi I_a \end{aligned} \tag{37}$$

and the dynamic equations are:

$$\begin{aligned} L_a \frac{dI_a}{dt} &= V_s - V_b - \left(R_a + R_f\right)I_a - k_\phi \omega \\ J_T \frac{d\omega}{dt} &= k_\phi I_a - Q_{prop} \end{aligned} \tag{38}$$

Table 8: DC motor constants.

Constant	Value
V_b	0.019V
R_a	0.223Ω
L_a	74μH
J_T	165×10^{-6}kgm^2
k_φ	$0.0374-6.17\times10^{-5}\omega+5.12\times10^{-7}\omega^2$ $-2.27\times10^{-9}\omega^3+7.72\times10^{-12}\omega^4$
R_a+R_f	1.8Ω

where n is the shaft speed in revolutions per second and ω the shaft speed in radians per second, V_s is the voltage supplied to the motor, R_a is the armature resistance of the motor, R_f is the effective resistance of the FET drive circuitry, I_a is the armature current, V_b is the voltage drop across the brushes, E is the back emf generated by the motor, k_ϕ is the motor constant (in volt seconds), Q is the torque generated, L_a is the armature inductance, J_T is the thruster system inertia and Q_{prop} is the torque required to turn the propeller. The values for the constants are given in Table 8. Although k_ϕ is nominally a constant, it was found from tests to vary with motor speed and was thus modelled as a function of ω.

2.5.2 Simulation Model

The effects of mechanical inertia on the motor were modelled by using the second expression in (38). Given the propeller and vehicle speed, the torque required by the propeller, Q_{prop}, was found using (35). When producing the system model, a torque loss equal to $0.001n$ was used to represent frictional losses in the motor; a current dead zone (see later) equivalent to 1% of V_s was found to be appropriate to model stiction losses. Therefore, given the armature current I_a, the new motor shaft speed could be found.

As the motor was operated via a PWM drive with frequency 800Hz, an accurate simulation model of the drive would require the simulation to be run with a time-step less than a tenth of the PWM period, i.e. 1/8KHz=0.000125s. This level of temporal detail was not required in the rest of the simulation (the actual time-step used was 0.01s) and so the current from the PWM drive was modelled externally and found to be proportional to the mark-space ratio of the PWM drive.

The algorithm used was:

$$\begin{aligned}
V_s &= 9.6\text{V}\\
E &= k_\phi \omega\\
I_a &= \frac{m_d}{2500}\cdot\frac{\text{dead_zone}(V_s, V_b) - E}{R_a + R_f}\\
\dot{n} &= \frac{\text{dead_zone}\left(k_\varphi I_a, \, {}^{0.01V_s}\!/\!{}_{R_a+R_f} + 0.001|n|\right) - Q_{prop}}{2\pi\cdot J_T}
\end{aligned} \tag{39}$$

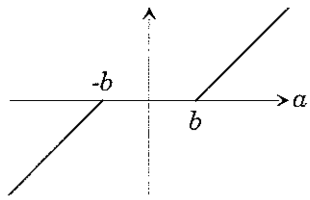


Figure 6: Deadzone definition.

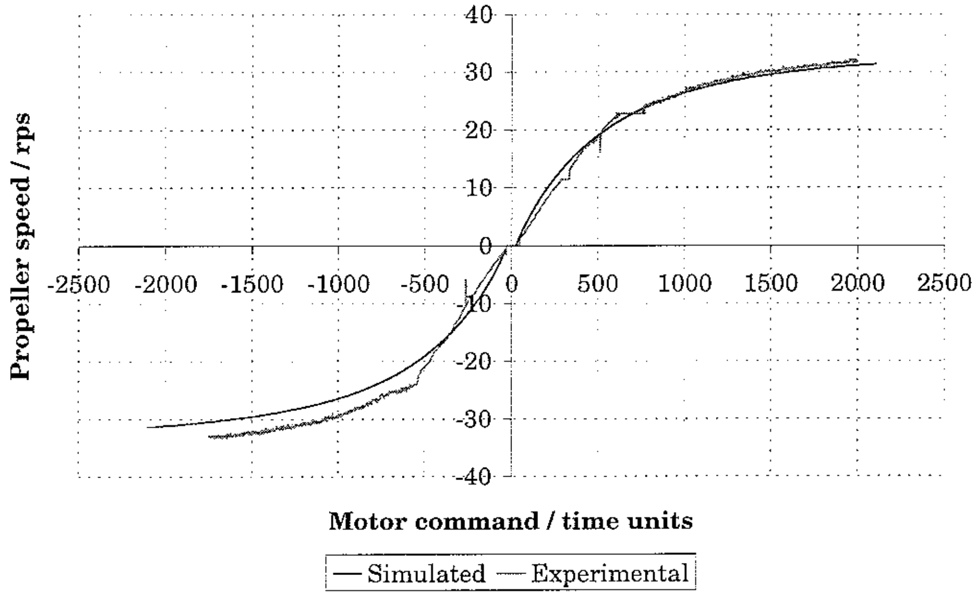


Figure 7: Propeller shaft speed in air against motor command.

where Q_{prop} is the torque required by the propeller and

$$\text{dead\_zone}(a,b) = \begin{cases} 0 & \text{if} |a| < b \\ a - b & \text{if } a > 0 \\ \mathrm{a + b} & \text{if a} < 0 \end{cases} \tag{40}$$

(b is always positive — see Figure 6.) Figure 7 shows the simulation model compared to an experimental run of the propeller in air — Q_{prop} was taken as being zero. It can be seen that the model is a good match, especially in the forwards direction.

2.6 Control Surface Actuator Model

Both rudder and sternplanes are controlled by PWM drives. Play and backlash in the servo systems meant that the control surfaces were not guaranteed to reach their commanded positions exactly, so the final steady-state positions were taken to be 90% of the demanded position. Figure 8 shows the actuator deisng conventions.

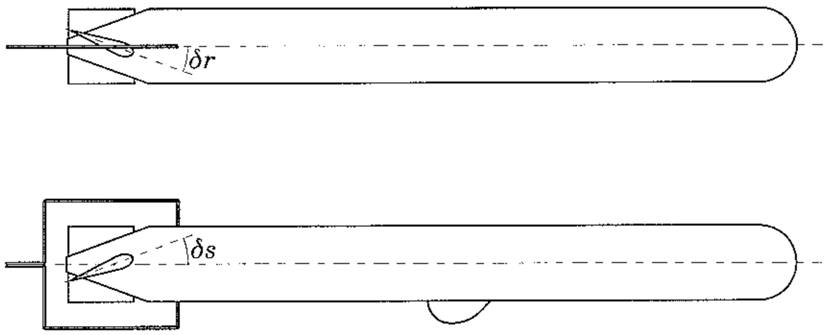


Figure 8: Control surface conventions.

2.6.1 Rudder

The electromechanical response of the rudder system was approximated as a first order lag with time constant 0.13s. As a transfer function,

$$\frac{\delta r}{\delta r_d} = 0.9\frac{7.69}{s + 7.69} \tag{41}$$

Tustin's mapping was then used to discretize this for a step size of 0.01s to give:

$$\delta r_k = 0.926\delta r_{k-1} + 0.067\delta r_{d_{k-1}} \tag{42}$$

2.6.2 Sternplane

Similarly, the sternplane system was modelled as a first order lag, with time constant 0.087s. Thus,

$$\frac{\delta s}{\delta s_d} = 0.9\frac{11.53}{s + 11.53} \tag{43}$$

with the discrete version being:

$$\delta s_k = 0.891378\delta s_{k-1} + 0.0981\delta s_{d_{k-1}} \tag{44}$$

2.7 Model Validation

It is normal practice with AUV controllers to partly decouple the six degrees of freedom into three subsystems — speed, heading and depth. Given this, model validation tests were carried out on the three subsystems to test the accuracy of the modelling between motor command and speed, rudder command and heading, and sternplane command and pitch/depth.

2.7.1 Speed

The straight line speed response of the vehicle with the tether model is shown in Figure 9. Agreement is good.

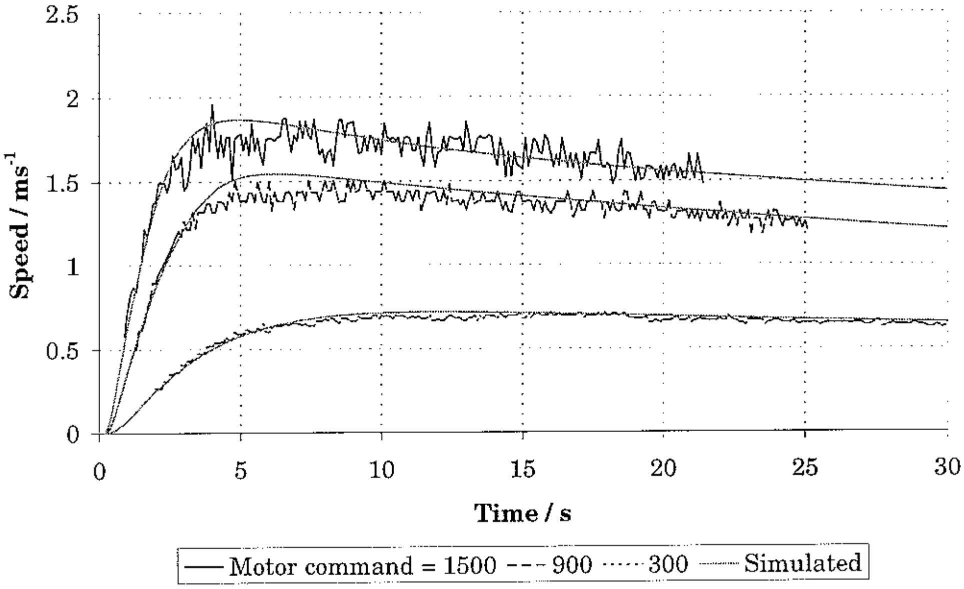


Figure 9: Straight line run — experimental versus simulation with tether model.

2.7.2 Heading

Figure 10 shows the experimental and simulation responses to the rudder command shown. Although the responses may not appear similar, it can be seen that both have the same overall shape with rises and peaks at the same times. However the experimental behaviour is less deterministic than might be expected — for example, before $t = 10$s, the vehicle swings to the left, although the rudder that is being applied is either zero, or acting to turn the vehicle to the right.

2.7.3 Depth

The definitive depth-model validation test is naturally between sternplane angle and depth. However, the sternplane position affects the

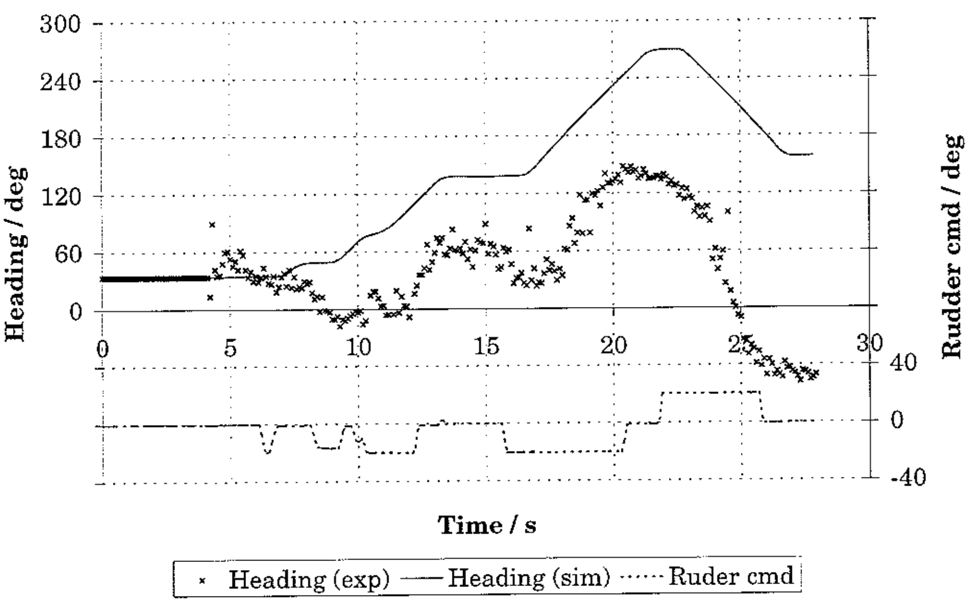


Figure 10: Open-loop model validation — heading.

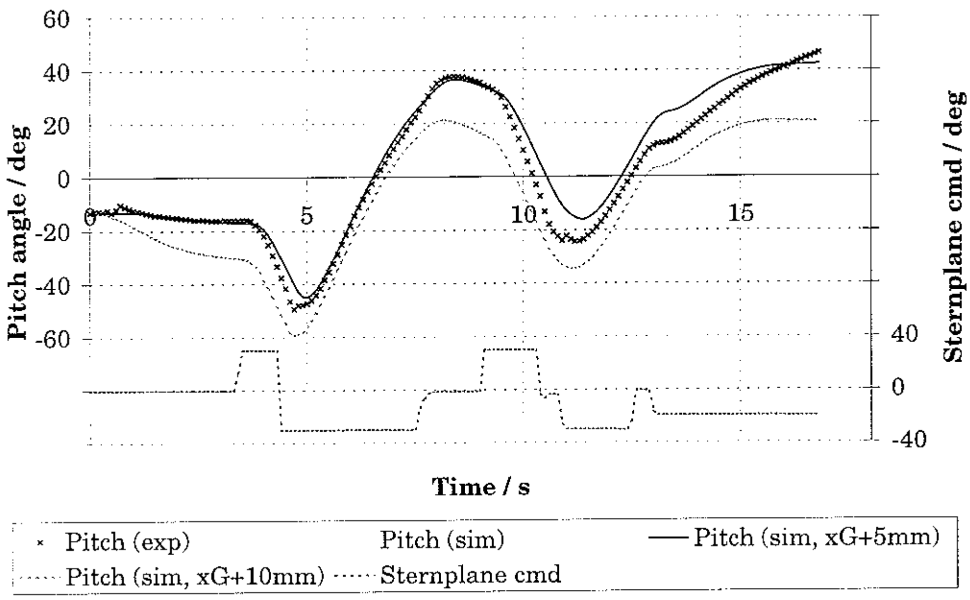


Figure 11: Open-loop model validation — pitch.

pitch rate, the time integral of which gives pitch, and the time integral of pitch with speed affects depth. Thus, for example, although a simulated pitch response that is within ±1° of the experimental response for dynamic manoeuvres might be considered accurate, a 1° pitch error over time results in a significant depth error. Thus from the depth response alone one might erroneously conclude that the model was not accurate.

Therefore, the pitch response of the vehicle is considered first. It is well predicted by the simulation, as shown in Figure 11. Agreement here is good between the experimental and simulation results although it was necessary to adjust the position of the vehicle's centre of mass in the simulation — the best agreement is when the centre of mass is 5mm forwards of its nominal position. (This suggests that the vehicle was not

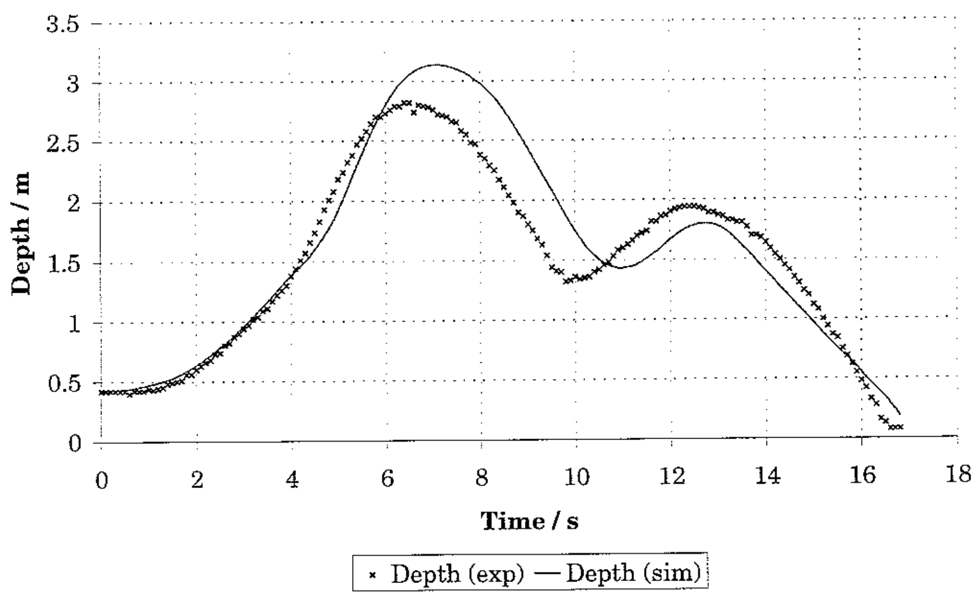


Figure 12: Open-loop model validation — depth.

trimmed exactly level in the experimental test.) Nevertheless, the sternplane-pitch dynamics of the vehicle are accurately modelled by the simulation, although, unsurprisingly, the open-loop response is affected by the vehicle's trim.

The depth response from the same test is shown in Figure 12 for the case with the adjusted centre of mass position. The agreement is reasonable.

3

Flow and Functions

Intended for those wishing to modify the program, this chapter describes the overall flow of the simulation before listing and describing each function in the program.

3.1 Simulation Flow

3.1.1 Abstracted Algorithm

The overall solution method is shown in Figure 13. This shows the fun-

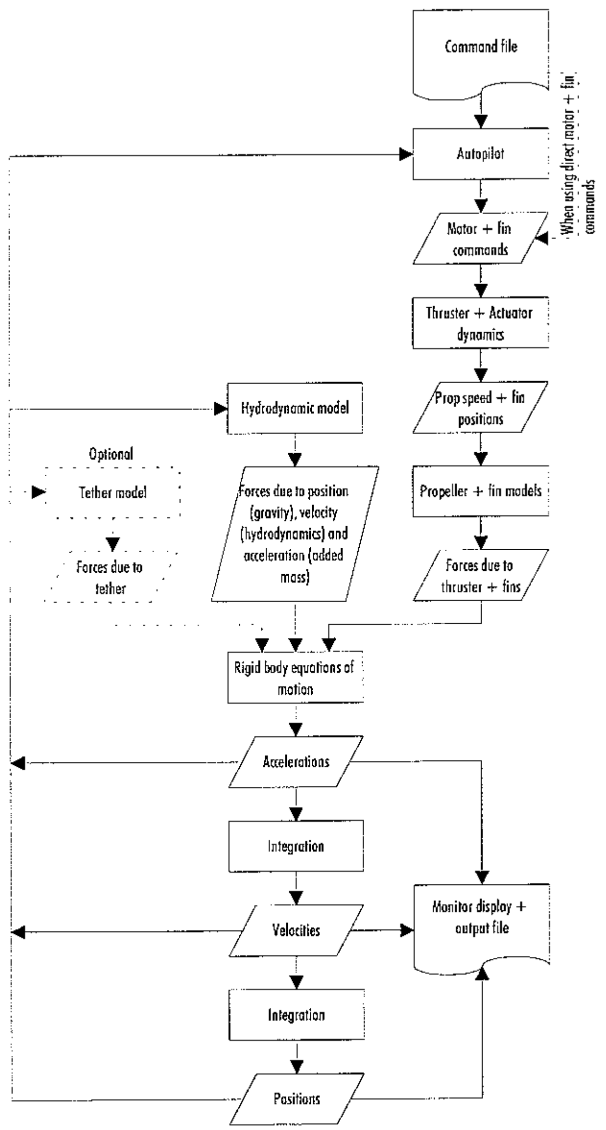


Figure 13: Flow diagram of simulation program.

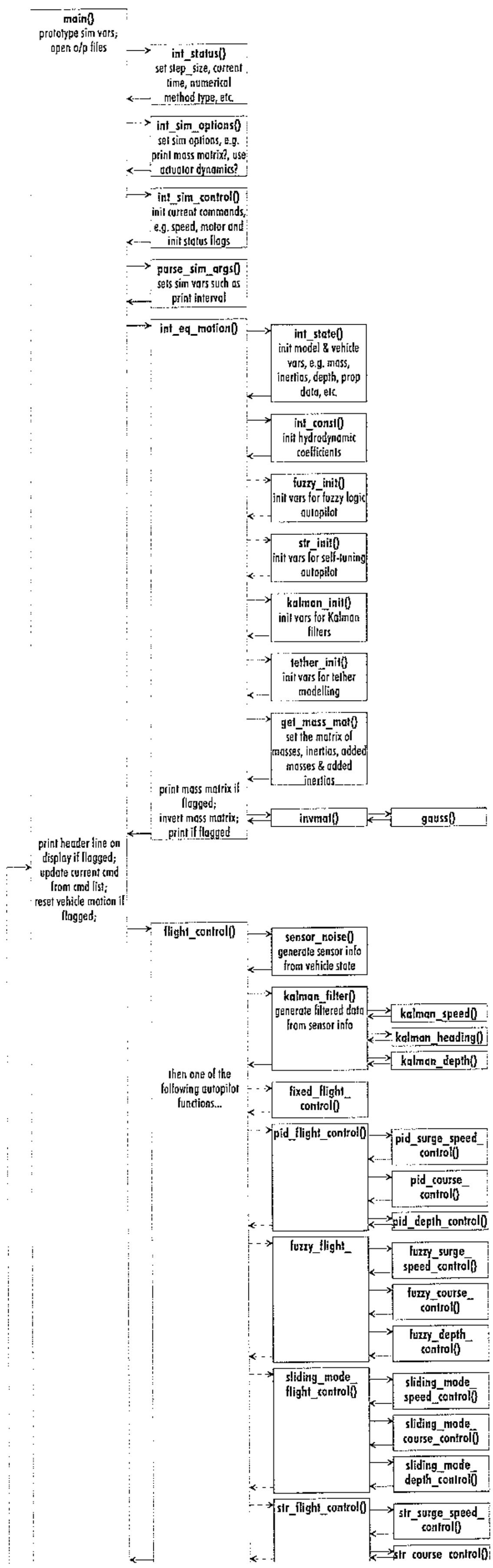
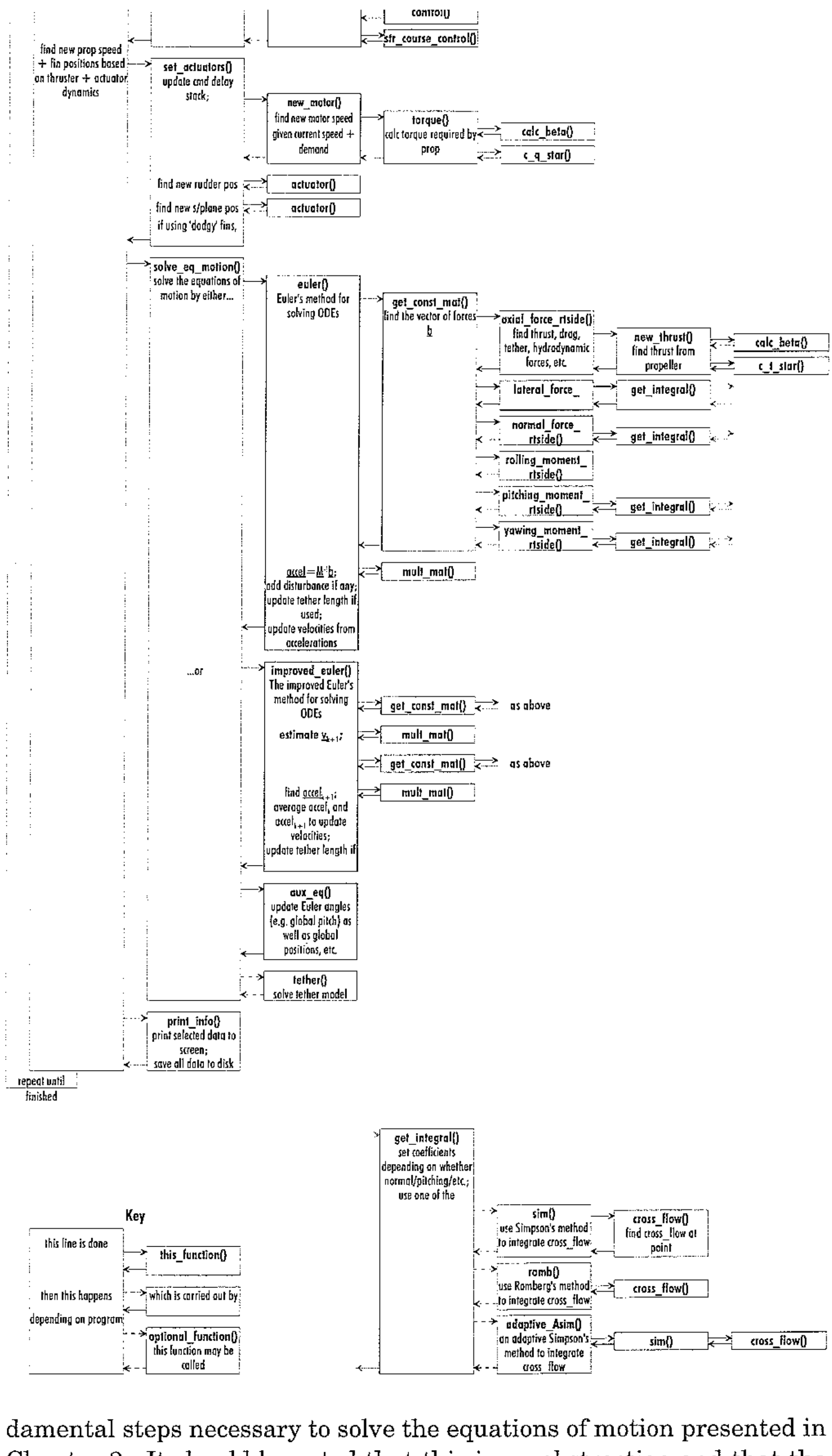


Figure 14: Simulation run-time flowchart.



damental steps necessary to solve the equations of motion presented in Chapter 2. It should be noted that this is an abstraction and that the steps presented may actually occur in a slightly different order. Certainly, the size of the steps is in no way related to the effort needed to carry them out.

3.1.2 Actual Program Flow

Figure 14 shows the actual flow through the functions of the simulation when the simulation is run.

3.2 The Functions

The program's functions are listed below in the following format:

function_name() **module_name.c**
Inputs: **int** input to function, **double** another input to function
Outputs: **float** output from function
This function is described here.

main() **sub.c**
Inputs: none
Outputs: none
The main function of the simulation program. Calls functions to initialize variables before entering the main simulation loop of running an autopilot, doing the actuator dynamics then solving the equations of motion before printing the results.

parse_sim_args() **sim_int.c**
Inputs: STATUS, SIM_OPT_FLAGS, SIM_COMMAND_LIST
Outputs: none
Originally it parsed options off the command line; now it just has things like the time interval between printing results hardcoded in. Also calls int_sim_cmds() to read the command file.

int_sim_options() **sim_int.c**
Inputs: SIM_OPT_FLAGS
Outputs: none
Sets simulation options such as whether to print the mass matrix out and whether to use the model of actuator dynamics, etc.

int_sim_control() **sim_int.c**
Inputs: SIM_CONTROL
Outputs: none
Initializes the simulation's commands (speed, depth, motor command, etc.) and sets some simulation flags.

int_sim_cmds() **sim_int.c**
Inputs: char *file, SIM_COMMAND_LIST
Outputs: **int** -1 if cannot read command list, 0 otherwise
Reads in the file containing the list of commands.

set_actuators() **rov_syst.c**

Inputs: STATE, SIM_CONTROL, STATUS, **int** actuator_flag

Outputs: none

The overall function to generate the new motor speed and fin positions for the current time step. Also updates the command stack (for time delays).

actuator() **rov_syst.c**

Inputs: **double** des_angle, **double** angle, **double** time_step, **char** rudder

Outputs: **double** new fin position

Calculates new fin (rudder and sternplane) positions based on commanded position, current position and the fin's slew rate. Uses different slew rate for the rudder.

new_motor() **rov_syst.c**

Inputs: **double** des_speed, **double** time_step, STATE

Outputs: none

Calculates the new motor speed, given the motor command and current motor speed.

new_thrust() **rov_syst.c**

Inputs: STATE

Outputs: **double** propeller thrust

Calculates the thrust produced by the propeller.

c_t_star() **rov_syst.c**

Inputs: STATE

Outputs: **double** thrust coefficient C_T^*

Produces a value for the thrust coefficient C_T^* given the information in STATE (β and the Fourier series terms).

torque() **rov_syst.c**

Inputs: STATE

Outputs: **double** propeller torque

Returns the torque required by the propeller, given its rotational speed.

c_q_star() **rov_syst.c**

Inputs: **double** beta1, STATE

Outputs: **double** torque coefficient C_Q^*

Returns a value for the torque coefficient C_Q^* from β and the Fourier data.

dead_zone() **rov_syst.c**

Inputs: **double** input, **double** zone_size

Outputs: **double** output value

Returns the value of the input given an opposing force (or whatever) which would result in a dead-zone.

sign() **rov_syst.c**

Inputs: **double** input

Outputs: **int** output value

Returns -1 if the input is negative, +1 if positive and 0 if zero.

calc_beta **rov_syst.c**

Inputs: **double** n, **double** u, **double** point_7_pi_d

Outputs: **double** beta

Calculates β given the propeller speed, vehicle speed and propeller size: $\beta = \tan^{-1}\left(u / 0.7\pi Dn \right)$

int_const() **intc_rov.c**

Inputs: **double** density, **double** length, HYDRO_COEFF

Outputs: none

Sets the hydrodynamic coefficients for the vehicle

int_state() **ints_rov.c**

Inputs: STATE, HULL_SHAPE

Outputs: none

Sets the model and vehicle parameters, e.g. mass, inertias, hull shape, propeller data, etc.

int_eq_motion() **rov_int.c**

Inputs: STATE, HYDRO_COEFF, HULL_SHAPE, STATUS, STR_VARIABLES, **double** A[MATRIX_SIZE][MATRIX_SIZE], **int** debug_flag

Outputs: none

The overall function that initializes the data for the equations of motion (e.g. hydrodynamics coefficients, tether coefficients, etc.).

int_status() **rov_int.c**

Inputs: STATUS

Outputs: none

Initializes certain simulation parameters, e.g. time step, initial time value, etc.

solve_eq_motion **sub_solv.c**

Inputs: STATE, HYDRO_COEFF, HULL_SHAPE, STATUS, **double**A[MATRIX_SIZE][MATRIX_SIZE], **double** b[MATRIX_SIZE]

Outputs: none

The overall function that selects a method to solve the equations of motion; also updates the current time.

euler() **sub_solv.c**

Inputs: STATE, HYDRO_COEFF, HULL_SHAPE, STATUS,**double** A[MATRIX_SIZE][MATRIX_SIZE], **double** b[MATRIX_SIZE], **double** time_step, **int** CFD_method

Outputs: none

Finds a solution to the equations of motion by using Euler's method of solving ODEs: $v_{k+1} = v_k + \Delta t.\dot{v}_k$.

improved_euler() **sub_solv.c**

Inputs: STATE, HYDRO_COEFF, HULL_SHAPE, STATUS, **double** A[MATRIX_SIZE][MATRIX_SIZE], **double** b[MATRIX_SIZE], **double** time_step, **int** CFD_method

Outputs: none

Finds a solution to the equations of motion by using the Improved Euler's method of solving ODEs: $v_{k+1} = v_k + \frac{1}{2}\Delta t(\dot{v}_k + \dot{v}_{k+1})$.

aux_eq() **sub_solv.c**

Inputs: STATE, **double** time_step

Outputs: none

Updates the Euler angles and other global variables given the vehicle velocities for the next time step.

gauss() **sub_math.c**

Inputs: **int** n, **int** m, **double** a[MATRIX_SIZE][2*MATRIX_SIZE]

Outputs: **int** 0 if successful, -1 otherwise

Inverts a matrix by using Gaussian elimination.

invmat() **sub_math.c**

Inputs: **int** n, **double** a[MATRIX_SIZE][MATRIX_SIZE]

Outputs: **int** 0 if successful, -1 otherwise

Prepares a matrix to be inverted using Gaussian elimination in the gauss() function. On return, the matrix a is inverted.

mult_mat() **sub_math.c**

Inputs: **double** a[MATRIX_SIZE][MATRIX_SIZE], **double** b[MATRIX_SIZE], **double** c[MATRIX_SIZE]

Outputs: none

Multiplies a matrix and vector; used to find the vehicle's accelerations by multiplying the inverse of the mass matrix with the vector of hydrodynamic forces.

sim() **sub_math.c**

Inputs: **double** left, **double** right, **int** num, FUNC_DATA, HULL_SHAPE

Outputs: **double** integral value

Integrates the cross-flow function using Simpson's method.

romb() **sub_math.c**

Inputs: **double** left, **double** right, **int** num, FUNC_DATA, HULL_SHAPE

Outputs: **double** integral value

Integrates the cross-flow function using Romberg's method.

adaptive_Asim() **sub_math.c**

Inputs: **double** left, **double** right, **double** tol, FUNC_DATA, HULL_SHAPE

Outputs: **double** integral value

Integrates the cross-flow function using an adaptive Simpson's method.

cross_flow() **sub_misc.c**

Inputs: **double** sum_x, FUNC_DATA , HULL_SHAPE

Outputs: **double** cross_flow value

Returns the value of the cross-flow at a given point along the vehicle' s length.

get_integral() **sub_misc.c**

Inputs: STATE, HULL_SHAPE, **int** eq_type, **int** CFD_method

Outputs: **double** integral value

Sets up the data for finding the cross-flow integral and calls the particular integration routine.

get_const_mat() **sub_misc.c**

Inputs: **double** const_mat[MATRIX_SIZE], STATE, HYDRO_COEFF, HULL_SHAPE, STATUS, **int** CFD_method

Outputs: none

Find the vector of hydrodynamics forces on the vehicle. (The function title is misleading given that it's not constant nor is it a matrix.)

get_mass_mat() **sub_misc.c**

Inputs: **double** A[MATRIX_SIZE] [MATRIX_SIZE], STATE, HYDRO_COEFF

Outputs: none

Sets the matrix of masses, inertias, added masses and added inertias.

axial_force_rtside() **sub_rtsi.c**

Inputs: STATE, AXIAL_COEFF, HULL_SHAPE, STATUS

Outputs: **double** value for force

Calculates the hydrodynamic forces acting axially on the vehicle.

lateral_force_rtside() **sub_rtsi.c**

Inputs: STATE, LATERAL_COEFF, HULL_SHAPE, STATUS, **int** CFD_method

Outputs: **double** value for force

Calculates the hydrodynamic forces acting laterally (i.e. mostly sway) on the vehicle.

normal_force_rtside() **sub_rtsi.c**

Inputs: STATE, NORMAL_COEFF , HULL_SHAPE, STATUS, **int** CFD_method

Outputs: **double** value for force

Calculates the hydrodynamic forces acting normally (i.e. mostly heave) on the vehicle.

rolling_moment_rtside() **sub_rtsi.c**

Inputs: STATE, ROLL_COEFF, HULL_SHAPE, STATUS

Outputs: **double** value for moment

Calculates the hydrodynamic moments acting around the vehicle's main axis.

pitching_moment_rtside() **sub_rtsi.c**

Inputs: STATE, PITCH_COEFF , HULL_SHAPE, STATUS, **int** CFD_method

Outputs: **double** value for moment

Calculates the hydrodynamic moments acting to pitch the vehicle.

yawing_moment_rtside() **sub_rtsi.c**

Inputs: STATE, YAW_COEFF, HULL_SHAPE, STATUS, **int** CFD_method

Outputs: **double** value for moment

Calculates the hydrodynamic moments acting to yaw the vehicle.

print_mass_mat() **sub_prin.c**

Inputs: **double** A[MATRIX_SIZE][MATRIX_SIZE]

Outputs: none

Prints the mass matrix and its inverse for debugging purposes.

print_const_mat() **sub_prin.c**

Inputs: **double** b[MATRIX_SIZE]

Outputs: none

Prints the vector of hydrodynamics forces.

print_info() **sub_prin.c**

Inputs: STATE, SIM_CONTROL, STR_VARIABLES, **int** method, **long int** count, **double** step

Outputs: none

Prints a header giving details of the command file, the integration method, etc. as well selected data (see OPT.H) on the vehicle's states (velocities, positions, etc.).

flight_control() **rov_ctrl.c**
Inputs: SIM_CONTROL, STATE, STATUS, STR_VARIABLES
Outputs: none
General autopilot functions. Gets sensor data before calling the appropriate autopilot.

sensor_noise **rov_ctrl.c**
Inputs: STATE
Outputs: none
Generates (possibly noise corrupted) sensor data from the vehicle's states.

pid_flight_control() **rov_pid.c**
Inputs: SIM_CONTROL, STATE
Outputs: none
General PID/classical autopilot.

pid_surge_speed_control() **rov_pid.c**
Inputs: SIM_CONTROL, STATE
Outputs: none
PID/classical autopilot for speed control.

pid_course_control() **rov_pid.c**
Inputs: SIM_CONTROL, STATE
Outputs: none
PID autopilot for heading control.

pid_depth_control() **rov_pid.c**
Inputs: SIM_CONTROL, STATE
Outputs: none
PID autopilot for depth control.

fuzzy_init() **rov_fuzz.c**
Inputs: none
Outputs: none
Initializes the variables for the fuzzy logic autopilots.

fuzzy_flight_control() **rov_fuzz.c**
Inputs: SIM_CONTROL, STATE, STATUS
Outputs: none
General fuzzy logic autopilot.

fuzzy_surge_speed_control() **rov_fuzz.c**
Inputs: SIM_CONTROL
Outputs: none
Fuzzy logic autopilot for speed control.

fuzzy_heading_control() **rov_fuzz.c**
Inputs: SIM_CONTROL, STATE, STATUS
Outputs: none
Fuzzy logic autopilot for heading control.

fuzzy_depth_control() **rov_fuzz.c**
Inputs: SIM_CONTROL, STATE, STATUS
Outputs: none
Fuzzy logic autopilot for depth control.

fuzzy() **rov_fuzz.c**
Inputs: **double** a, **double** b, **double** c, **double** d, **double** variable
Outputs: **double** set membership value
Returns the membership of the input variable in the fuzzy set defined by a trapezoidal membership function.

sliding_mode_flight_control() **rov_smc.c**
Inputs: SIM_CONTROL, STATE
Outputs: none
General sliding mode autopilot.

sliding_mode_speed_control() **rov_smc.c**
Inputs: SIM_CONTROL, STATE
Outputs: none
Sliding mode autopilot for speed control.

sliding_mode_heading_control() **rov_smc.c**
Inputs: SIM_CONTROL
Outputs: none
Sliding mode autopilot for heading control.

sliding_mode_depth_control() **rov_smc.c**
Inputs: SIM_CONTROL
Outputs: none
Sliding mode autopilot for depth control.

sgn() **rov_smc.c**
Inputs: **double** variable
Outputs: **int** sign of variable
The same as sign().

sat() **rov_smc.c**
Inputs: **double** variable
Outputs: **double** output value
Saturation function. Returns -1 if input is less than -1, +1 if greater than +1, otherwise the input value. (I.e. it saturates at ±1.)

fixed_flight_control() **sub_fix.c**
Inputs: SIM_CONTROL
Outputs: none
'Autopilot' allowing the motor and fin commands to be specified directly.

str_flight_control() **rov_str.c**
Inputs: SIM_CONTROL, STATE, STATUS, STR_VARIABLES
Outputs: none
General self-tuning autopilot.

str_surge_speed_control() **rov_str.c**
Inputs: SIM_CONTROL, STR_VARIABLES
Outputs: none
Self-tuning autopilot for speed control.

str_course_control() **rov_str.c**
Inputs: SIM_CONTROL, STATE, STATUS, STR_VARIABLES
Outputs: none
Self-tuning autopilot for heading control.

str_init() **rov_str.c**
Inputs: STR_VARIABLES
Outputs: none
Initializes variables needed by self-tuning autopilots.

kalman_init() **kalman.c**
Inputs: STATE
Outputs: none
Initializes variables needed by Kalman filters.

kalman_filter() **kalman.c**
Inputs: none
Outputs: none
Calls the three Kalman filters for speed, heading and depth.

kalman_speed() **kalman.c**
Inputs: none
Outputs: none
Kalman filter for speed subsystem.

kalman_heading() **kalman.c**
Inputs: none
Outputs: none
Kalman filter for heading subsystem.

kalman_depth() **kalman.c**
Inputs: none
Outputs: none
Kalman filter for depth subsystem.

mult_3_mat() **kalman.c**
Inputs: **double** a[3][3], **double** b[3][3], **double** c[3][3]
Outputs: none
Multiplies two 3×3 matrices: $\boldsymbol{C} = \boldsymbol{AB}$.

inv_3_mat() **kalman.c**
Inputs: **double** matrix_to_invert[3][3]
Outputs: **int** 0 if successful, -1 otherwise
Inverts a 3×3 matrix.

mult_2_mat() **kalman.c**
Inputs: **double** a[2][2], **double** b[2][2], **double** c[2][2]
Outputs: none
Multiplies two 2×2 matrices: $\boldsymbol{C} = \boldsymbol{AB}$.

inv_2_mat() **kalman.c**
Inputs: **double** matrix_to_invert[2][2]
Inputs: **int** 0 if successful, -1 otherwise
Inverts a 2×2 matrix.

mult_4_mat() **kalman.c**
Inputs: **double** a[4][4], **double** b[4][4], **double** c[4][4]
Outputs: none
Multiplies two 4×4 matrices: $\boldsymbol{C} = \boldsymbol{AB}$.

inv_4_mat() **kalman.c**
Inputs: **double** matrix_to_invert[4][4]
Outputs: none
Inverts a 4×4 matrix.

mult_5_mat() **kalman.c**
Inputs: **double** a[5][5], **double** b[5][5], **double** c[5][5]
Outputs: none
Multiplies two 5×5 matrices: $\boldsymbol{C} = \boldsymbol{AB}$.

inv_5_mat() **kalman.c**
Inputs: **double** matrix_to_invert[5][5]
Outputs: **int** 0 if successful, -1 otherwise
Inverts a 5×5 matrix.

tether_init() **tether.c**
Inputs: STATE
Outputs: none
Initializes variables for the tether model.

tether() **tether.c**

Inputs: STATE
Inputs: **double** time_step
Outputs: none

Calculates new tether positions given the vehicle's current motion.

mult_t_mat() **tether.c**

Inputs: **double** a[TETHER_POINTS][TETHER_POINTS], **double** b[TETHER_POINTS][TETHER_POINTS], **double** c[TETHER_POINTS][TETHER_POINTS], **int** skip
Outputs: none

Multiplies two TETHER_POINTS×TETHER_POINTS size matrices: $\boldsymbol{C} = \boldsymbol{AB}$. Can miss the first 'skip' rows and columns out.

mult_t_vect() **tether.c**

Inputs: **double** a[TETHER_POINTS][TETHER_POINTS], **double** b[TETHER_POINTS], **double** c[TETHER_POINTS], **int** skip
Outputs: none

Multiplies a TETHER_POINTS×TETHER_POINTS size matrix with a TETHER_POINTS size vector: $\boldsymbol{c} = \boldsymbol{Ab}$. Can miss the first 'skip' rows out.

inv_t_mat() **tether.c**

Inputs: **int** n, **double** a[TETHER_POINTS][TETHER_POINTS]
Outputs: **int** 0 if successful, -1 otherwise

Inverts a TETHER_POINTS size matrix.

t_gauss() **tether.c**

Inputs: **int** n, **int** m, **double** a[TETHER_POINTS][2*TETHER_POINTS]
Outputs: **int** 0 if successful, -1 otherwise

Does the inversion of a TETHER_POINTS matrix by Gaussian elimination.

gaussj() **new_math.c**

Inputs: **float** **a, **int** n, **float** **b, **int** m
Outputs: none

Inverts any size matrix using the Gauss-Jordan method.

nrerror() **new_math.c**

Inputs: char error_text[]
Outputs: none

Error return function for the NEW_MATH.C routines.

****ivector()*** **new_math.c**

Inputs: **int** nl, **int** nh
Outputs: **int** pointer to vector

Sets up a vector of **ints**.

free_ivector() **new_math.c**

Inputs: **int** *v, **int** nl, **int** nh

Outputs: none

Releases space when a vector of **ints** set with *ivector() is no longer needed.

*****matrix()*** **new_math.c**

Inputs: **int** nrl, **int** nrh, **int** ncl, **int** nch

Outputs: **float** pointer to matrix

Sets up a matrix of **floats**.

free_matrix() **new_math.c**

Inputs: **float** **m, **int** nrl, **int** nrh, **int** ncl, **int** nch

Outputs: none

Releases space when a matrix of **floats** set with **matrix() is no longer needed.

*****submatrix()*** **new_math.c**

Inputs: **float** **a, **int** oldrl, **int** oldrh, **int** oldcl, **int** oldch, **int** newrl, **int** newcl

Outputs: **float** pointer to submatrix

Extracts a submatrix from a matrix.

free_submatrix() **new_math.c**

Inputs: **float** **b, **int** nrl, **int** nrh, **int** ncl, **int** nch

Outputs: none

Releases space when a submatrix is no longer needed.

big_gauss **new_math.c**

Inputs: **int** n, **int** m, **double** a[13][2*13]

Outputs: **int** 0 if successful, -1 otherwise

Inverts a large matrix by Gaussian elimination.

inv_big_mat() **new_math.c**

Inputs: **int** n, **double** a[13][13]

Outputs: **int** 0 if successful, -1 otherwise

Does the inversion of a large matrix by Gaussian elimination.

4

References

[1] R. Lea, R. Allen, and S. Merry, "A low-cost remotely operated underwater vehicle," *Journal of Measurement + Control*, vol. 29, pp. 201-204, Sept. 1996.

[2] R.K. Lea, "Control of a tethered underwater flight vehicle in the horizontal plane," in *Proceedings of the 10th International Symposium on Unmanned Untethered Submersible Technology*, pp. 261-271, 1997.

[3] R.K. Lea, *Control of a Tethered Underwater Flight Vehicle*, Ph.D. thesis, ISVR, University of Southampton, 1998.

[4] T.I. Fossen, *Guidance and Control of Ocean Vehicles*, Wiley, New York, 1995.

[5] J. Feldman, "Revised standard submarine equations of motion," Tech. Rep. SPD-0393-09, David W Taylor Naval Ship Research and Development Center, 1979.

[6] C.T. Howell, *Investigation of the Dynamics of Low-Tension Cables*, Ph.D. thesis, Woods Hole Oceanographic Institution + Massachusetts Institute of Technology, June 1992, WHOI-92-30.

[7] W.P.A. van Lammeren, J.D. van Manen, and M.W.C. Oosterveld, "The Wageningen B-screw series," *Transactions of the Society of Naval Architects and Marine Engineers*, pp. 269-317, 1969.

[8] R.J. Smith, *Circuits, Devices and Systems*, Wiley, New York, 1984.

Part II

Module Listings

In the following sections, the simulation program is listed via its component modules. The code itself is listed on the right, with all in-program comments included and in italics. On the left pages are additional notes, with the vertical bars indicating the scope of each particular comment.

The usual C conventions apply in the program itself. That is, all functions are lowercase and definitions are uppercase. Each level of looping or program control is indented one tab stop; where a program statement runs across several lines, each subsequent line is indented two tap stops on from the first.

1

OPT.H

Various options for simulation, model and so on

Options to customise the output as displayed on-screen (the output to file is fixed and includes all state variables)

```c
/* Roy Lea 23/6/97 */
/* File: opt.h Version: 1.1 */
/* Options for the simulation */

#define YS TRUE
#define NO FALSE

#define DISTURBANCE     FALSE
#define MTR_TIME_DELAY  15      // Motor time delay in units of step_time (0.01s)
#define RUD_TIME_DELAY  75      // Rudder time delay in units of step_time (0.01s)
#define SPL_TIME_DELAY  23      // Sternplane time delay in units of step_time (0.01s)
#define DODGY_FINS      FALSE
#define SENSOR_REAL     TRUE
#define TETHER_DYNAMICS FALSE
#define BENDING         FALSE
#define SL_TETHER_DRAG  FALSE   // Straight line tether drag
#define CHANGE_PARAMS   FALSE
#define CONTROL_TYPE    FUZZY
       /* PID, FUZZY, [ADAPTIVE_FUZZY], SLIDING_MODE, FIXED, STR are acceptable */

#define PRINT_COURSE    YS
#define PRINT_RUDDER    YS
#define PRINT_DEPTH     YS
#define PRINT_Z_DOT     NO
#define PRINT_PTCH_D    NO
#define PRINT_PITCH     YS
#define PRINT_Q         NO
#define PRINT_STERNP    YS
#define PRINT_SPEED     YS
#define PRINT_U_DOT     NO
#define PRINT_RPS       NO
#define PRINT_MTRCMD    YS
#define PRINT_STR       NO
```

2

SIM.H

Basic definitions

Maximum number of input command...
...and maximum time delay for actuators

Simulation control structure. Contains general simulation flags as well as variables to hold the current commanded speed, depth, etc. as well as current actuator and motor commands.

Various simulation option flags

```c
/* John Kloske  10-31-91    Rev: 11-25-91 */
/* Andy Shein */
/* Roy Lea 23/6/97 */
/* File: sim.h Version: 2.1  */
/* structures used to control the flow and options of sub simulation */

#ifndef SIM_H   /* include safety */
#define SIM_H

#ifndef TRUE
#define FALSE 0
#define TRUE 1
#endif

#ifndef RUN
#define INITILIZE 0   /* where do i use this */
#define RUN 1
#endif

#ifndef MANUAL
#define AUTO 0      /* for manual and pitch flags */
#define MANUAL 1
#endif

#define MAX_CMDS 10
#define MAX_TIME_DELAY 201

typedef struct {
   int reset;    /* boolean used to reset the dynamics */
   int pause;    /* boolean used to pause thr dynamics */
   int exit;     /* boolean used to exit the main simulation loop */
   double deltar;    /* commanded rudder position used to set the state [rad] */
   double deltas;    /* commanded stern plane position used to set the state */
   double deltab;    /* commanded bow plane  position used to set the state */
   double deltaa;    /* commanded differential stern plane */
   double RPS;       /* commanded main prop RPS used to set motor controller */
   double v_arm;     /* commanded motor armature voltage */
   double course;    /* commanded course in [deg] */
   double depth;     /* commanded depth in [ft] */
   double speed;     /* commanded speed in [kts] */
   double pitch;     /* commanded pitch angle in [deg] */
   int manual_flag;  /* manual overide flag, state set by deltar, s, a, a and */
                          /* RPS, insted of using closed loop controller */
   int pitch_flag;   /* flag used to cut out depth error to pitch controller */
                          /* and use SIM_CONTROL pitch as command */
   double cmd_stack[MAX_TIME_DELAY][3]; // command stack for time delay use
   } SIM_CONTROL;

typedef struct {
   int socket_flag;   /* communication through sockets */
   int header_flag;   /* print a header to stdout every print time */
   int debug_flag;    /* print mass matrix and other stuff */
   int actuator_flag; /* dont use dynamics if true */
   int graphics_flag; /* use graphics display on IRIS */
   int full_screen_flag;    /* make the graphics window fill the screen */
   int command_file_flag;   /* read in alist of cammands from a file */

   } SIM_OPT_FLAGS;
```

Similar to SIM_CONTROL defined above, this structure defines the command list that stores autopilot and actuator demands indexed against the time they should occur

The actual command list structure

```
typedef struct {
   double time;        /* time to exicute command */
   double deltar;      /* commanded rudder position used to set the state [rad] */
   double deltas;      /* commanded stern plane position used to set the state */
   double deltab;      /* commanded bow plane  position used to set the state */
   double deltaa;      /* commanded differential stern plane */
   double RPS;         /* commanded main prop RPS used to set motor controller */
   double v_arm;       /* commanded motor armature voltage */
   double course;      /* commanded course in [deg] */
   double depth;       /* commanded depth in [m] */
   double speed;       /* commanded speed in [m/s] */
   double pitch;       /* commanded pitch angle in [deg] */
   int manual_flag;  /* manual overide flag, state set by deltar, s, a, a and */
            /* RPS, insted of using closed loop controller */
   int pitch_flag;    /* flag used to cut out depth error to pitch controller */
            /* and use SIM_CONTROL pitch as command */
   } SIM_COMMAND;

typedef struct {
   SIM_COMMAND cmd_lst[MAX_CMDS];
   int max_index;
   int current_index;

   } SIM_COMMAND_LIST;

#endif   /* end of include safety */
```

3

SUB.H

Give numbers to the control types that are used in OPT.H

Give numbers to the vehicle types simulated (not used)

Various definitions to allow names to be used in the program for the various equations, solution methods, etc.

Definitions used in the cross-flow integral

```
/* John Kloske  4/21/91    Rev: 11-25-91 */
/* Andy Shein */
/* Roy Lea 23/6/97 */
/* File: sub.h Version: 2.1  */
/*  Structure of all non-linear coefficients/Variables.      */
/*    Notation from: "David W. Taylor Naval Ship Research    */
/*                    and Development Center", Feldman, J., */
/*                    June 1979.  (DTNSRDC/SPD-039-09)       */

/* Make it safe to include this file more than once */
#ifndef SUB_H
#define SUB_H

#define SQR(x)  ((x)*(x))
#define MAG(x,y)  (sqrt(SQR(x) + SQR(y)))

#ifndef TRUE
#define TRUE  1
#define FALSE 0
#endif

#ifndef ERROR
#define ERROR -1
#endif

#define PID                 1  /* Control types */
#define FUZZY            2
#define ADAPTIVE_FUZZY 3
#define SLIDING_MODE        4
#define FIXED            5
#define STR                 6

#define FAU  1  /* Vehicle types */
#define ROV  2

#define AXIAL      1       /* First equation listed in DTNSRDC Eqs */
#define LATERAL    2       /* 2 nd eq */
#define NORMAL     3       /* 3 rd eq */
#define ROLL       4       /* 4 th eq */
#define PITCH      5       /* 5 th eq */
#define YAW        6       /* 6 th and last eq listed! */

#define EULER        1     /* to solve ODE's by Euler's method */
#define IMP_EULER    2     /* to solve ODE's by improved Euler's method */
#define RUNGE_KUTTA  3     /* to solve ODE's by Runge-Kutta method */
#define ADAMS        4     /* to solve ODE's by Adams-Moulton method */

#define SIMPSON      1      /* integration by simpson's method */
#define ASIM         2      /* integration by adaptive simpson's method */
#define ROMB         3      /* integration by Romberg method */

#define MATRIX_SIZE 7       /* coeff matrix A[6x6] not using zero position   */
#define NUM_EQ 6

#define NO_STATIONS 300     /* total number of stations from st # 0 */
#define MAX_POINTS  21      /* Max number of points allowed for integration */
            /* Number of data points must be ODD  sim()     */

#define MAX_ASIM_ITTER 1000 /* maximum neber of itterations in adaptive */
```

Conversion factors between degrees and radians (program uses radians internally, but commands for heading are specified in degrees)

Conversion factors between American and SI units

Number of tether points (nodes) used in the model

Variables for the self-tuning speed autopilot

Variables for the self-tuning heading autopilot

Variables for the self-tuning depth autopilot

```
               /* simpson's method */

/* Mathamatical constatnts */

#define RELERR      0.01        /* relative error for Adams() adaptive_romberg()*/
#define PRECISION   0.5e-10     /* Machine tolerence */
#define VEL_REL_ERR 0.01        /* minimum velocity that is != 0.0 */
#define MAX_SIM_TIME 1.0e6      /* max time used to shut up the compiler about */
                 /* not being able to exit the main loop */

#ifndef PI
#define PI          3.1415927  /* just the value of Pi */
#endif

#define PIBY2       PI/ 2.0     /* Pi / 2 used in Rt side Roll eq */
#define TWO_PI      2.0*PI      /* 2* Pi used in solveode.c aux_eq() */

#ifndef TODEG
#define TODEG       180.0/PI    /* convert from degrees to radians */
#define TORAD       PI/180.0    /* convert from radians to degrees */
#define TOKTS      0.5925    /* convert ft/sec to knots */
#endif

#ifndef FTtoM
#define FTtoM            0.3048
#define SLUGtoKG         14.5959
#define SLUGFT2toKGM2    1.356
#define LBFtoN           4.44822
#define HPtoKW           0.7457
#define KNOTtoMS         0.51444
#define SLUGFT3toKGM3    515.449
#define TORQUEchange     1.3558  /* Assuming the original units were lbf.ft */
#endif

#ifndef TETHER_POINTS
#define TETHER_POINTS   21
#endif

typedef struct {
   double theta[2], theta_kminus1[2];
   double phi_kminus1[2];
   double k[2];
   double p[2][2], p_kminus1[2][2];
   double epsilon, lambda, last_command, time_of_last_command, adapt_flag;
   double k_i, k_p;
   } STR_SPEED;

typedef struct {
   double theta[12], theta_kminus1[12];
   double phi_kminus1[12];
   double k[12];
   double p[12][12], p_kminus1[12][12];
   double controller_data[12];
   double epsilon, lambda, last_command, time_of_last_command, adapt_flag;
   int times_round;
   } STR_HEADING;

typedef struct {
   double theta[8], theta_kminus1[8];
   double phi_kminus1[8];
   double k[8];
```

The self-tuner variables packaged into one overall structure.
Plus some additional variables for the autopilot control laws.

Thruster model variables

Vehicle model and state variables

```
   double p[8][8], p_kminus1[8][8];
   double epsilon, lambda, last_command, time_of_last_command, adapt_flag;
   double k_i, k_p, k_d;
   } STR_DEPTH;

typedef struct {
   STR_SPEED spd;
   STR_HEADING head;
   STR_DEPTH dep;
   double speed_integrator;
   double heading_integrator;
   } STR_VARIABLES;

typedef struct {
   double K1;              /* coefficent used model thruster thrust in lbf */
   double K2;              /* coefficent used to model thruster horsepower */
   double K3;              /* coefficent usted to model thruster tourqe */
   double beta1;           /* hydrodynamic pitch angle */
   double propeller_diam;  /* propeller diameter */
   double point_7_pi_d;    /* = 0.7 * PI * propeller_diam */
   double wake_fraction;   /* wake fraction number */
   double R_a;             /* motor armature resistance */
   double L_a;             /* motor armature inductance */
   double k_phi;           /* motor constant */
   double v_brush;         /* motor brush voltage */
   double J_thruster;      /* thruster inertia */
   double fourier_thrust [2] [21]; /* 20 fourier terms for the 4-quad data */
   double fourier_torque [2] [21]; /* 20 fourier terms for the 4-quad data */
   } PROPULSOR;    /* used in state, must be defined here */

typedef struct  {
     double  density;       /* mass density of water */
     double  mass;          /* mass of submarine, including water */
     double  weight;        /* weight of submarine w/ free flooding spaces */
     double  B;             /* buoyancy force of envelope displacement */
     double  c;             /* modeling thrust,drag to full scale */
     double  u;             /* velocity component in x-axis */
     double u_dash_kminus1;
     double u_dash_kminus2;
     double  v;             /* velocity component in y-axis */
     double  w;             /* velocity component in z-axis */
     double  p;             /* angular velocity about x-axis (roll)  */
     double  q;             /* angular velocity about y-axis (pitch) */
     double  r;             /* angular velocity about z-axis (yaw)   */
     double  u_dot;         /* acceleation component in x-axis */
     double  v_dot;         /* acceleation component in y-axis */
     double  w_dot;         /* acceleation component in z-axis */
     double  p_dot;         /* angular acceleation about x-axis */
     double  q_dot;         /* angular acceleation about y-axis */
     double  r_dot;         /* angular acceleation about z-axis */
     double  theta;         /* angle of pitch */
     double  phi;           /* angle of roll  */
     double  psi;           /* angle of yaw */
     double  psi_dash_kminus1;
     double  psi_dash_kminus2;
     double  theta_dot;     /* rate of change angle of pitch */
     double  phi_dot;       /* rate of change angle of roll */
     double  psi_dot;       /* rate of change angle of yaw */
     double  x_o;           /* a coordinate of displacement re fixed axes */
     double  y_o;           /* a coordinate of displacement re fixed axes */
     double  z_o;           /* a coordinate of displacement re fixed axes */
     double  z_dash_kminus1; /* z~(k-1) for digital controllers */
```

End of vehicle model and state variables

```
  double  x_o_dot;       /* rate of change of coordinate displacement*/
  double  y_o_dot;       /* rate of change of coordinate displacement*/
  double  z_o_dot;       /* rate of change of coordinate displacement */
  double  U;             /* velocity of origin of body re fluid */
  double  alpha;         /* angle of attack */
  double  beta;          /* angle of drift */
  double  I_x;           /* moment of inetia about x-axis */
  double  I_y;           /* moment of inetia about y-axis */
  double  I_z;           /* moment of inetia about z-axis */
  double  I_xy;          /* Prod of inetia w.r.t x and y axes */
  double  I_yz;          /* Prod of inetia w.r.t y and z axes */
  double  I_zx;          /* Prod of inetia w.r.t z and x axes */
  double  X_G;           /* The x coodinate of CG */
  double  Y_G;           /* The y coodinate of CG */
  double  Z_G;           /* The z coodinate of CG */
  double  Z_B;           /* >>> NOT LISTED <<< */
  double  X_B;           /* >>> NOT LISTED <<< */
  double  Y_B;           /* >>> NOT LISTED <<< */
  double  deltar;        /* deflection of rudder */
  double  deltar_kminus1; /* deltar(k-1) for digital controllers */
  double  deltas;        /* deflection of sternplane */
  double  deltas_kminus1; /* deltas(k-1) for digital controllers */
  double  deltab;        /* deflection of bowplane or sailplane */
  double  deltaa;        /* differentail deflection of stern planes */
  double  RPS;           /* rev per min of prop >> NOT LISTED << */
  double  RPS_dot;       /* change in RPS */
  double  RPS_kminus1;  /* n(k-1) for digital controllers */
  double  RPS_kminus2;  /* n(k-2) for digital controllers */
  double  V_s;          /* motor supply voltage */
  double  V_s_kminus1;
  double  i_a;           /* motor armature current */
  double  i_a_dot;       /* change in motor armature current */
  double  eta;           /* ratio u_c/u */
  double  Cd;            /* coeff used in integrating forces and */
          /* moments along hull due to local cross-flow */

  double  F_xp;          /* net thrust - drag */

  double  v_s;           /* vel comp. in y-axis dir at the quarter  */
          /*  chord of the sternplanes. v_s= v + x_sr */

  double  w_s;           /* vel comp. in y-axis dir at the quarter */
          /* chord of the sternplanes. w_s= w + x_sq */

  double  Q_p;           /* contribution of propeller torque to K and */
          /* machinery equation */

  PROPULSOR prop;        /* propulsor charistics, here because propulsor */
          /* could be out of water changing charistics */

  double X;              /* X force in vehicle coordinates */
  double Y;              /* Y force in vehicle coordinates */
  double Z;              /* Z force in vehicle coordinates */
  double K;              /* Rolling moment in vehicle coordinates */
  double M;              /* Pitching moment in vehicle coordinates */
  double N;              /* Yawing moment in vehicle coordinates */
  double CFY;            /* Y force from cross flow integral */
  double CFZ;            /* Z force from cross flow integral */
  double CFM;            /* Pitching moment from cross flow integral */
  double CFN;            /* Yawing moment from cross flow integral */
} STATE;
```

Axial (i.e. mostly surge) hydrodynamic coefficient variables for vehicle model

Lateral (i.e. mostly sway) coefficient variables

Normal (i.e. mostly heave) coefficient variables

Roll coefficient variables

```c
typedef struct {
     double  X_qq;             /* coeff representing X as a func of q^2 */
     double  X_rr;             /* coeff representing X as a func of r^2 */
     double  X_rp;             /* coeff representing X as a func of rp */
     double  X_u_dot;          /* coeff representing X as a func of du/dt */
     double  X_vr;             /* coeff representing X as a func of vr */
     double  X_wq;             /* coeff representing X as a func of wq */
     double  X_vv;             /* coeff representing X as a func of vv */
     double  X_ww;             /* coeff representing X as a func of w^2 */
     double  X_deltar_deltar;  /* coeff rep. X as a func of u^2(dr)^2 */
     double  X_deltas_deltas;  /* coeff rep. X as a func of u^2(ds)^2 */
     double  X_deltab_deltab;  /* coeff rep. X as a func of u^2(db)^2 */
     double  X_uu;             /* coeff from Draper Labs NOT DTNSRDC Eq */

     /* these coeff below added on 1-29-91 for Draper sub */

     double  X_deltaa_deltaa;  /* coeff from Draper Labs NOT DTNSRDC Eq */
     double  X_w_deltas;       /* coeff from Draper Labs NOT DTNSRDC Eq */
     double  X_q_deltas;       /* coeff from Draper Labs NOT DTNSRDC Eq */
     double  X_v_deltar;       /* coeff from Draper Labs NOT DTNSRDC Eq */
     double  X_r_deltar;       /* coeff from Draper Labs NOT DTNSRDC Eq */
   } AXIAL_COEFF;

typedef struct {
     double  Y_r_dot;          /* coeff representing Y func of dr/dt */
     double  Y_p_dot;          /* coeff representing Y func of dp/dt */
     double  Y_v_dot;          /* coeff representing Y func of dv/dt */
     double  Y_p_p;            /* coeff representing Y func of p|p|  */
     double  Y_pq;             /* coeff representing Y func of pq    */
     double  Y_r;              /* coeff representing Y func of ur    */
     double  Y_p;              /* coeff representing Y func of up    */
     double  Y_wp;             /* coeff representing Y func of wp    */
     double  Y_star;           /* coeff representing Y func of u^2 Y* */
     double  Y_v;              /* coeff representing Y func of uv    */
     double  Y_v_v_R;          /* coeff representing Y func of v|v|R */
     double  Y_deltar;         /* coeff representing Y func of u^2(delta r) */
     double  Y_deltar_eta;     /* coeff rep. Y func of u^2dr(eta-1/c)c */
   } LATERAL_COEFF;

typedef struct {
     double  Z_q;              /* coeff representing Z func of uq */
     double  Z_vp;             /* coeff representing Z func of vp */
     double  Z_q_dot;          /* coeff representing Z func of dq/dt */
     double  Z_w_dot;          /* coeff representing Z func of dw/dt */
     double  Z_star;           /* coeff representing Z func of u^2 */
     double  Z_w;              /* coeff representing Z func of uw */
     double  Z_w_;             /* coeff representing Z func of u|w| */
     double  Z_ww;             /* coeff representing Z func of w*MAG(v,w) */
     double  Z_deltas;         /* coeff representing Z func of u^2(delta s) */
     double  Z_deltab;         /* coeff representing Z func of u^2(delta b) */
     double  Z_deltas_eta;     /* coeff rep. Y func of u^2(ds)(nc -1) */

     /* coeff added on 1-29-91 for Draper sub only */

     double  Z_pr;             /* coeff from Draper Labs NOT DTNSRDC Eq */
   } NORMAL_COEFF;

typedef struct {
     double  K_p;              /* coeff representing K func of p */
     double  K_p_dot;          /* coeff representing K func of dp/dt */
     double  K_i;              /* coeff rep. due to interference effects of */
             /* of vortices from the bridge fairwater on */
```

Pitch coefficient variables

Yaw coefficient variables

```
            /* the stern control surfaces */
    double  K_vp;               /* coeff representing K func of vp */
    double  K_star;             /* coeff representing K func of u^2 */
    double  K_r;                /* coeff representing K func of ur */
    double  K_r_dot;            /* coeff representing K func of dr/dt */
    double  K_p_p;              /* coeff representing K func of p|p| */
    double  K_qr;               /* coeff representing K func of qr */
    double  K_vR;               /* coeff representing K func of uv */
    double  K_v_dot;            /* coeff representing K func of dv/dt */
    double  K_wp;               /* coeff representing K func of wp */
    double  K_deltar;           /* coeff representing K func of u^2(delta r) */
    double  K_deltar_eta;       /* coeff rep. Y func of u^2(dr)(nc -1) */
    double  K_4S;               /* coeff representing K due to phi_s at stern */
    double  K_8S;               /* coeff representing K due to phi_s at stern */

    /* coeff added on 1-29-91 for Draper sub only */

    double  K_wr;               /* coeff from Draper Labs NOT DTNSRDC Eq */
    double  K_deltaa;
   } ROLL_COEFF;

typedef struct {
    double  M_star;             /* coeff representing M func of u^2 */
    double  M_q;                /* coeff representing M func of uq */
    double  M_q_dot;            /* coeff representing M func of dq/dt */
    double  M_rp;               /* coeff representing M func of rp */
    double  M_w;                /* coeff representing M func of uw */
    double  M_w_dot;            /* coeff representing M func of dw/dt */
    double  M_w_;               /* coeff representing M func of u|w| */
    double  M_w_w_R;            /* coeff representing M func of w*MAG(v,w) */
    double  M_ww;               /* coeff representing M func of|w*MAG(v,w)| */
    double  M_deltab;           /* coeff representing M func of u^2(delta b) */
    double  M_deltas;           /* coeff representing M func of u^2(delta s) */
    double  M_deltas_eta;       /* >>> Not listed <<< */

    /* coeff added on 1-29-91 for Draper sub only */
    double  M_vp;               /* coeff from Draper Labs NOT DTNSRDC Eq */
    double  M_v_deltaa;         /* coeff from Draper Labs NOT DTNSRDC Eq */
    double  M_r_deltaa;         /* coeff from Draper Labs NOT DTNSRDC Eq */
   } PITCH_COEFF;

typedef struct {
    double  N_star;             /* coeff representing N func of u^2 */
    double  N_p;                /* coeff representing N func of up */
    double  N_p_dot;            /* coeff representing N func of dp/dt */
    double  N_pq;               /* coeff representing N func of pq */
    double  N_r;                /* coeff representing N func of ur */
    double  N_r_dot;            /* coeff representing N func of dr/dt */
    double  N_v;                /* coeff representing N func of uv */
    double  N_v_dot;            /* coeff representing N func of dv/dt */
    double  N_v_v_R;            /* coeff representing N func of v*MAG(v,w) */
    double  N_deltar;           /* coeff representing N func of u^2(delta r) */
    double  N_deltar_eta;       /* coeff rep. N func of u^2(dr)(nc -1) */

    /* coeff added on 7-14-91 for Draper sub only */

    double  N_w_deltaa;         /* coeff from Draper Labs NOT DTNSRDC Eq */
    double  N_q_deltaa;         /* coeff from Draper Labs NOT DTNSRDC Eq */
    double  N_deltas_deltaa;    /* coeff from Draper Labs NOT DTNSRDC Eq */
   } YAW_COEFF;
```

All the hydrodynamic coefficient variables in one structure

Structure to hold data from the last time step for the Adams method (not used)

Hull shape variables for the cross-flow calculations

Model variables for the cross-flow calculations

Simulation state variables

```
typedef struct {
     AXIAL_COEFF   axial;     /* right hand side coefficients */
     LATERAL_COEFF lateral;
     NORMAL_COEFF  normal;
     ROLL_COEFF    roll;
     PITCH_COEFF   pitch;
     YAW_COEFF     yaw;
   } HYDRO_COEFF;

typedef struct {
     double YPnm1[MATRIX_SIZE]; /* values from rt side of eq's at Y'n-1 */
     double YPnm2[MATRIX_SIZE]; /* values from rt side of eq's at Y'n-2 */
     double YPnm3[MATRIX_SIZE]; /* values from rt side of eq's at Y'n-3 */
     double YPnm4[MATRIX_SIZE]; /* values from rt side of eq's at Y'n-4 */
     int    error;              /* true if difference between Two adams */
            /* method is > RELERR  */
   } LAST_STEP;

typedef struct {
     double x[NO_STATIONS]; /* distance along the hull station 0 start */
     double R[NO_STATIONS]; /* radius at position x[] body of revolution */
     double length;         /* overall length of submarine */
     double x_B;            /* distance from CG to the bow */
     double x_AP;           /* distance from CG to the AP */
     double x_s;            /* x-co-ord of quarter cord of the sternplanes */
     double inc;            /* distance between stations */
   } HULL_SHAPE;

typedef struct {
     double c1;    /* coefficients: f1 = v1 + x*c1  */
     double c2;
     double c3;
     double v1;    /* velocity terms, v,w, etc...  */
     double v2;
     double v3;
     int  xp;      /* boolean true if xp = x the variable */
   } FUNC_DATA;

typedef struct {
   double step_size;      /* step size in [sec] used to solve the ODE's */
   double sim_time;       /* simulation time in [sec] */
   double print_time;     // time in s to print header or send state over socket
   long int sim_count;    /* number of times solve_ODE has been called in */
                               /* simulation since last reset */
   long int print_count; /* print interval converted to counts */
   int ODE_method;        /* method used to sove the ODE's */
   int CFD_method;        /* method used to integrate cross flow */
   int dev_flag;          /* to have Y'n values stored in b[]  */
         /* RUNGE_KUTTA method only */

   int num_eq;            /* number of eq */
   } STATUS;

/* structures for use with turn radius calculation in sub_misc.c */
/* NOT SUPPORTED YET JOHN GOT LAZY AS 11-25-91 */

typedef struct {
     double x;    /* independent variable */
     double y;    /* dependent variable */

   } POINT;       /* Note z = f(x,y) ==> both independent variables */
```

Some global function prototypes

'Sensor data' variables — i.e.representing the sensor data available on Subzero II

Structures and variables holding data for the three Kalman filters: speed, heading and depth

Tether model and state variables

```
typedef struct {
     POINT old;      /* first set of data points */
     POINT center;  /* mid data points */
     POINT new;      /* last set of data points collected */

  } TURN_DATA;

    /* DYNAMICS FUNCTION PROTOTYPES */

    void int_status(STATUS *stat);
    void int_eq_motion(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull,
              STATUS *stat, STR_VARIABLES *str_var,
              double A[MATRIX_SIZE][MATRIX_SIZE], int debug_flag);

    void solve_eq_motion( STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull,
            STATUS *stat, double A[MATRIX_SIZE][MATRIX_SIZE],
            double b[MATRIX_SIZE] );

    void int_last( STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull,
          STATUS *stat, LAST_STEP *last, double A[MATRIX_SIZE][MATRIX_SIZE],
          double b[MATRIX_SIZE] );

// The following are global variables

typedef struct {
    double speed, u_dot;
    double heading, r;
    double depth, pitch, q;
    double roll;
    } SENSOR_DATA;

typedef struct {
    double h[2][2], f[2][2], delta[2][2], q[2][2];
    double s_hat[2];
    double p[2][2];
    } KALMAN_SPEED_DATA;

typedef struct {
    double h[4][4], f[4][4], delta[4][4], q[4][4];
    double s_hat[4];
    double p[4][4];
    } KALMAN_HEADING_DATA;

typedef struct {
    double h[5][5], f[5][5], delta[5][5], q[5][5];
    double s_hat[5];
    double p[5][5];
    } KALMAN_DEPTH_DATA;

typedef struct {
    KALMAN_SPEED_DATA speed;
    KALMAN_HEADING_DATA head;
    KALMAN_DEPTH_DATA depth;
    } KALMAN_FILTER;

typedef struct {
    double cable_length;
    double space_step;
    double t_i[TETHER_POINTS];
    double u_i[TETHER_POINTS], u_iplus1[TETHER_POINTS];
    double v_i[TETHER_POINTS], v_iplus1[TETHER_POINTS];
    double phi_i[TETHER_POINTS], phi_iplus1[TETHER_POINTS],
```

End of tether model and state variables

Current time variables, external for easy access when debugging

```
        phi_iminus1[TETHER_POINTS];
    double x_i[TETHER_POINTS], x_iplus1[TETHER_POINTS];
    double y_i[TETHER_POINTS], y_iplus1[TETHER_POINTS];
    double m, ma, ml;
    double Cdn, Cdt;
    double diam;
    double max_diff;
    double precision;
    double EI;
    double sl_length;    // straight line tether length
    } TETHER;

extern double time;

#endif   /* endif for include safety */
```

4

ROV_EXT.H

External declerations of the global variables used in the program

```
// Author:    R.K.Lea
// Date:      28 April 1997
// File:      ROV_EXT.H
// Notes:     Global variables (!) for AUTOROV simulation program.  Used instead
//            of the existing structure as there appears to be problems with stack
//            overflow when passing parameters - may be due to the use of local
//            variables everywhere.

extern SENSOR_DATA sen;
extern KALMAN_FILTER kf;
extern double smc_speed_int_term;
extern double smc_heading_int_term;
extern double smc_depth_int_term;
extern TETHER t;
```

5

SUB.C

Global variables declared here

Start of **main()** function

Simulation control variables are declared locally here

Simulation data variables are declared locally here

```c
/* John Kloske  4/21/91    Rev: 11-25-91 */
/* Andy Shein */
/* Rev: 4-3-96 by Roy Lea */
/* File: sub.c Version: 2.1  */
/* This file contains the main function ONLY */
/* this program starts up the dynamics process */

#include <stdio.h>         /* fprintf() plus fopen & fclose */
#include <dos.h>           // for stack size
#include <math.h>          // for fmod in seeing whether to do control
#include <stdlib.h>        // malloc

#include <sub.h>           /* header file for general constants etc... */
#include <sim.h>           /* simulation control structures */
#include <opt.h>           /* simulation options */

FILE *out_file;
FILE *out_teth;
SENSOR_DATA sen;
KALMAN_FILTER kf;
double smc_speed_int_term;
double smc_heading_int_term;
double smc_depth_int_term;
extern unsigned _stklen=65000;    // big stack
double cable_length=0.0;
double time;

void main(void) {
   void int_sim_control(SIM_CONTROL *ctrl);
   void int_sim_options(SIM_OPT_FLAGS *opt);
   void parse_sim_args(STATUS *status, SIM_OPT_FLAGS *opt, SIM_COMMAND_LIST *cmds);
   void set_actuators(STATE *s, SIM_CONTROL *ctrl, STATUS *stat, int actuator_flag);
   void print_info (STATE *s, SIM_CONTROL *ctrl, STR_VARIABLES *str_var, int method,
          long int count, double step);
   void flight_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat, STR_VARIABLES
          *str_var);
   int invmat(int n, double a[MATRIX_SIZE][MATRIX_SIZE]);
   void get_mass_mat(double A[MATRIX_SIZE][MATRIX_SIZE], STATE *s, HYDRO_COEFF *c);

   register int i, j;

   /* SIMULATION FLOW AND CONTROL VARIABLES */

   SIM_OPT_FLAGS opt;               /* command line options */
   SIM_CONTROL ctrl;                /* current simulation control commands */
   SIM_COMMAND_LIST cmds;           /* list of commands to exicute */
   STR_VARIABLES str_var;

   /* DYNAMICS VARIABLES */
   STATE state;          /* General coefficients/variables */
   HYDRO_COEFF coeff;    /* right hand side coefficents */
   HULL_SHAPE hull;      /* shape of hull and any positions */
   STATUS stat;          /* simulation status info */

   double A[MATRIX_SIZE][MATRIX_SIZE];  /* mass matrix [6x6] */

   double b[MATRIX_SIZE];    /* force vector */
           /* array used 1..6 ==> 7 elements, */
           /* not using position 0, yet */
```

Output files for vehicle data and tether data are opened here

Call functions to initialize simulation

Whilst we're not flagged as finished, keep looping
(The 'time' variable is global for easy reference whilst debugging.)

If we've got to the end of the command file, then flag as finished

Set the current vehicle commands based on time and the command list file

If the vehicle's dynamics are being reset, call the initialize function

Every 0.1s, call the autopilot function

Do the actuator dynamics — find current settings based on demand and last position

Solve the equations of motion to find new accelerations, velocities and positions

Print out the current vehicle positions, etc., if at correct time interval

Loop back to start

Finished the main simulation loop, so close output data files

End **main()** function and program

```
out_file=fopen("rov.out","w");
out_teth=fopen("tether.out","w");

/* SIMULATION INITILIZATIONS AND DEFAULTS */

int_status(&stat);                  /* initilize default dynamics status info */
int_sim_options(&opt);              /* set command line flag defaults */
int_sim_control(&ctrl);             /* set control defaults */
parse_sim_args(&stat, &opt, &cmds); /* parse command line */

/* INITILIZE DYNAMICS */

int_eq_motion(&state, &coeff, &hull, &stat, &str_var, A, opt.debug_flag);

if(opt.header_flag==TRUE) {
   print_info(&state, &ctrl, &str_var, stat.ODE_method, stat.sim_count,
         stat.step_size);
   }   /* run description line at t = 0.0 */

/* MAIN SIMULATION LOOP */

while( ctrl.exit != TRUE ) {
   time=stat.sim_time;
   if( opt.command_file_flag == TRUE ) {
      if( cmds.current_index >= (cmds.max_index-1)) {
         ctrl.exit = TRUE;
         }
      else {
         if((cmds.cmd_lst[cmds.current_index+1].time)<=(stat.sim_time))
               cmds.current_index++;
         else {
            ctrl.course = cmds.cmd_lst[cmds.current_index].course;
            ctrl.depth = cmds.cmd_lst[cmds.current_index].depth;
            ctrl.speed = cmds.cmd_lst[cmds.current_index].speed;
            }
         }
      }

   if( ctrl.reset == TRUE)
         int_eq_motion(&state, &coeff, &hull, &stat, &str_var, A, opt.debug_flag);

   if(fmod(stat.sim_time+0.001,0.1)<stat.step_size) flight_control(&ctrl, &state,
         &stat, &str_var);

   set_actuators(&state, &ctrl, &stat, opt.actuator_flag);    /* set fins */

   if(ctrl.pause!=TRUE) solve_eq_motion(&state, &coeff, &hull, &stat, A, b);

   if( (stat.sim_count%stat.print_count) == 0 ) {
      if( opt.header_flag == TRUE )
            print_info(&state, &ctrl, &str_var, stat.ODE_method, stat.sim_count,
            stat.step_size);

      if( opt.socket_flag == TRUE ) {
         /* send state and simtime out socket */
         }
      }
   }   /* end main while loop */

fclose(out_file);
fclose(out_teth);
}   /* end main */
```

6

SIM_INT.C

Start of **parse_sim_args()** function

If an external water current disturbance is being used (see OPT.H), then use the Euler method of integration...

...otherwise use the Improved Euler method

Set output printing interval to 0.1s

Call **int_sim_cmds()** to check that command file CMDS.DAT exists and load the commands into memory

End of **parse_sim_args()** function

Start of **int_sim_options()** function

Set simulation options

End of **int_sim_options()** function

```
/* John Kloske  4/21/91    Rev: 11-25-91 */
/* Andy Shein */
/* File: sim_int.c Version: 2.0  */
/* functions to initilize structures for simulation flow control */
/* and parse the command line */

#include <stdio.h>  /* fprintf() */
#include <process.h> /* exit() */

#include <sub.h>
#include <sim.h>
#include <opt.h>

void parse_sim_args(STATUS *status, SIM_OPT_FLAGS *opt, SIM_COMMAND_LIST *cmds){
   int int_sim_cmds(char *file, SIM_COMMAND_LIST *cmds);

   int c;                  /* returned command from getopt */
   extern char *optarg;    /* defined for getopt() see man */
   extern int optind;      /* defined for getopt() */
   extern int opterr;      /* defined for getopt() */
   char temp[100];         /* temp path name */

#if(DISTURBANCE==TRUE)
   status->ODE_method = EULER;
#else
   status->ODE_method = IMP_EULER;
#endif
   status->print_time = 0.101;
   opt->header_flag = TRUE;
   opt->command_file_flag = TRUE;
   (void)printf("Trying To Read COMMAND file %s \n", "cmds.dat");
   if( int_sim_cmds("cmds.dat", cmds) == ERROR ) {
      (void)fprintf(stderr,"Couldn't read command file \n");
      exit(ERROR);
      }
   /* set print count now that print time and step size are set */
   status->print_count = (long)(double)(status->print_time / status->step_size);
   } /* end sim parse args */


/*------------------------------------------------------------------------------*/

/*
    void int_options( OPTIONS *opt)
    initilize simulation options structure with defaults
*/

void int_sim_options(SIM_OPT_FLAGS *opt){
   opt->socket_flag = FALSE;          /* communication through sockets */
   opt->header_flag = FALSE;          /* print a header to stdout every print time */
   opt->debug_flag = FALSE;           /* print mass matrix and other stuff */
   opt->actuator_flag = FALSE;        /* dont use actuator dynamics */
   opt->graphics_flag = FALSE;        /* use graphics display */
   opt->full_screen_flag = FALSE;     /* use full screen display */
   opt->command_file_flag = FALSE;    /* read in a list of commands */
   }

/*------------------------------------------------------------------------------*/
```

Start of **int_sim_control()** function

Initialize simulation flags and current vehicle command variables

Initialize actuator time delay stack

End of **int_sim_control()** function

Start of **int_sim_cmds()** function
Returns a value depending on whether the command file has been read correctly

Try to open the command file
If it can't be opened, exit the function with an error

```
/*
    void int_sim_control( SIM_CONTROL *ctrl)
    initilize simulation control structure with defaults
*/

void int_sim_control(SIM_CONTROL *ctrl) {
   int i;
   ctrl->reset = FALSE;        /* boolean usedt to reset the dynamics */
   ctrl->pause = FALSE;        /* boolean used to pause thr dynamics */
   ctrl->exit = FALSE;         /* boolean used to exit the main simulation loop */
   ctrl->deltar = 0.0;         /* commanded rudder position used to set the state */
   ctrl->deltas = 0.0;         /* commanded stern plane position  */
   ctrl->deltab = 0.0;         /* commanded bow plane  position */
   ctrl->deltaa = 0.0;         /* commanded aileron angle */
   ctrl->RPS = 0.0;            /* commanded main prop RPS */
   ctrl->course = 0.0;         /* commanded course in [deg] */
   ctrl->depth = 0.0;          /* commanded depth in [m] */
   ctrl->speed = 0.0;          /* commanded speed in [m/s] */
   ctrl->pitch = 0.0;          /* commanded pitch angle in [deg] */
   ctrl->manual_flag = 0;      /* manual overide flag, state set by */
                               /* deltar, s, a, and RPS, insted of using */
                               /* closed loop controller */
   ctrl->pitch_flag = 0;       /* flag used to cut out depth error to pitch */
                               /* controller and use SIM_CONTROL pitch as command */
   for(i=0;i<MAX_TIME_DELAY;i++) {
      ctrl->cmd_stack[i][0]=0.0;
      ctrl->cmd_stack[i][1]=0.0;
      ctrl->cmd_stack[i][2]=0.0; // command stack for time delay use
      }
   }

/*
    int int_sim_cmds(SIM_COMMAND cmd_lst[10])
    initilize simulation control list array. Opens
    File with commands. with format

    time[sec] deltar[deg] deltas[deg] surge_velocity

      #.# #.# #.# #.#

    kind of cryptic but simple for a first draft. Reports back the
    commands read in then converts all angles to radians for later use.
    Also checks command times to make sure that later commands don't
    have erlier times.
    Only supports dr, ds, and RPS right now can later expand for use
    with testing stability controllers.

       char *file : the name of the command file to open
    SIM_COMMAND cmd_lst[10] : pointer to list of commands to fill
*/

int int_sim_cmds(char *file, SIM_COMMAND_LIST *cmds) {
   FILE *fp;
   int i;          /* read loop counter */
   int j;          /* print loop counter */
   int result;     /* number of characters read in by fscanf */

   if( (fp=fopen(file, "r")) == NULL ) {
      (void)fprintf(stderr, "Can't open command file: %s", file);
      return(ERROR);
      }
```

Otherwise read in commands until the end of the file is reached

Print the commands to screen

Close the command file

Convert heading commands from degrees to radians

Go through command list and check that each command occurs after the previous one
If not, exit from function with an error

End of **int_sim_cmds()** function
Returns 0 as successful

```
i = 0;          /* initilize read loop */
result = 0;

while( result != EOF ) {
    result = fscanf(fp, "%lf %lf %lf %lf", &(cmds->cmd_lst[i].time),
            &(cmds->cmd_lst[i].course), &(cmds->cmd_lst[i].depth),
            &(cmds->cmd_lst[i].speed));
    i++;
    }

cmds->max_index = i - 1;    /* number of commands read in */

/* echo list of commands */

(void)printf("Time [sec]  course  depth    speed \n");
for( j = 0; j <= (cmds->max_index-1); j++) {
    (void)printf("%4.0lf      %4.0lf    %4.0lf        %4.1lf \n",
        cmds->cmd_lst[j].time, cmds->cmd_lst[j].course,
        cmds->cmd_lst[j].depth, cmds->cmd_lst[j].speed);
    }

(void)printf("Read in %d Commands \n", cmds->max_index);

fclose(fp);

/* convert angles to [RAD] */
for( j = 0; j <= cmds->max_index - 1; j++) cmds->cmd_lst[j].course *= TORAD;

/* check times */
for( j = 0; j <= cmds->max_index - 2; j++) {
    if( cmds->cmd_lst[j].time > cmds->cmd_lst[j+1].time ) {
        (void)fprintf(stderr, "Command %d has time greater than Command %d \n",
                j, j+1 );
        return(ERROR);
        }
    }

cmds->current_index = 0;    /* set to first command */
return(0);
}
```

7

ROV_SYST.C

Declare the global variables in this module for fin offsets

Start of **set_actuators**() function

Only do this section if we're using the actuator dynamics model

Update the actuator command delay stack

Call the function to calculate the new motor speed
Call the function to calculate the new rudder position
Call the function to calculate the new sternplane position

If using the fin noise + offset model, add the offset to the 'true' position...

...otherwise the actual position is equal to the 'true' one

End of **set_actuators**() function

```
/* Roy Lea */
/* File: rov_syst.c   Version 1.1 */
/* this file contains functions to simulate the various subsystems of the */
/* ROV such as the actuators and motors */

#include <opt.h>

#include <math.h>  /* fabs() */
#include <stdlib.h>
#include <stdio.h>
#include <sub.h>
#include <sim.h>

double rudder_offset=0.0, splane_offset=0.0;
double rudder_pos=0.0, splane_pos=0.0;

/* void set_actuators( STATE *s, SIM_CONTROL *ctrl, int actuator_flag )

   function to set the fin positions and RPS an the state structure.
   if actuator flag is TRUE then the actuator models are not used

   STATE *s;             pointer to the state of the vehicle
   SIM_CONTROL *ctrl;    the commanded actuator positions
   STATUS *stat;         pointer to simulation status info
   int actuator_flag;    if true then don't use the actuator models
*/

void set_actuators(STATE *s, SIM_CONTROL *ctrl, STATUS *stat, int actuator_flag) {
   double motor(double des_rps, double rps, double time_step);
   void new_motor(double des_v_arm, double time_step, STATE *s);
   double actuator(double des_angle, double angle, double time_step,char rudder);
   register int i;
   int max_time_delay;

   if (!actuator_flag) {
      max_time_delay=max(MTR_TIME_DELAY,max(RUD_TIME_DELAY,SPL_TIME_DELAY));
      for(i=0;i<max_time_delay;i++) {
         ctrl->cmd_stack[i][0]=ctrl->cmd_stack[i+1][0];
         ctrl->cmd_stack[i][1]=ctrl->cmd_stack[i+1][1];
         ctrl->cmd_stack[i][2]=ctrl->cmd_stack[i+1][2];
         }
      ctrl->cmd_stack[MTR_TIME_DELAY-1][0]=ctrl->RPS;
      ctrl->cmd_stack[RUD_TIME_DELAY-1][1]=ctrl->deltar;
      ctrl->cmd_stack[SPL_TIME_DELAY-1][2]=ctrl->deltas;

      new_motor(ctrl->cmd_stack[0][0], stat->step_size, s);
      rudder_pos = actuator(ctrl->cmd_stack[0][1], rudder_pos, stat->step_size, TRUE);
      splane_pos = actuator(ctrl->cmd_stack[0][2], splane_pos, stat->step_size, FALSE);
      s->deltab = actuator(ctrl->deltab, s->deltab, stat->step_size, FALSE);
      s->deltaa = actuator(ctrl->deltaa, s->deltaa, stat->step_size, FALSE);
#if(DODGY_FINS==TRUE)
      s->deltar=rudder_pos+rudder_offset;
      s->deltas=splane_pos+splane_offset;
#else
      s->deltar=rudder_pos;
      s->deltas=splane_pos;
#endif
      }
   }
```

Start of **actuator()** function. This returns the new fin position, based on demand, current position, the time step and whether it's a rudder

The fin dynamics model is a first-order lag, which has been discretized for t=0.01s

If within 0.5° of the desired position, produce new random offsets from the 'true' position (can be different for rudders and sternplanes)

The dynamic model
Final position is 90% of the commanded position

Limit fin positions to ±20° for the rudder or ±30° for everything else

End of **actuator()** function
Return the new 'true' fin position

Start of **new_motor()** function; des_speed variable is actually the motor command

Calculate k_ϕ from motor speed using model obtained from experimental motor tests

The voltage applied to the motor is proportional to the motor command, less the brush voltage and the back emf due to the motor speed

The maximum current through the motor is based on the applied voltage…
…with the addition of a dead zone around zero

```
double actuator(double des_angle, double angle, double time_step, char rudder){

    if(time_step!=0.01) printf("Exponential fin model not valid");
    // Fin model below has been discretized from the analogue model

    if (fabs(0.9*des_angle-angle)>0.5*TORAD){
            if (rudder==TRUE)
                    rudder_offset=TORAD*(4.0*(double)(rand()%1000)/1000-2.0);
            else splane_offset=TORAD*(4.0*(double)(rand()%1000)/1000-2.0);
            }
        // Calculate a random fin offset that will be fixed once the fin is near
        // the commanded position.  Final position is 90% of the commanded.

    if (rudder==TRUE) angle=0.926*angle+0.067*des_angle;
    else angle=0.89137*angle+0.0981*des_angle;
    if (rudder==TRUE) {                       // limit actuator travel to +/- 20 deg
        if( angle > 0.35 ) angle = 0.35;
        if( angle < -0.35 ) angle = -0.35;
        }
    else {                                    // limit actuator travel to +/- 30 deg
        if( angle > 0.524 ) angle = 0.524;
        if( angle < -0.524 ) angle = -0.524;
        }

    return(angle);
    }


/* void new_motor(double des_speed, double time_step, STATE *s)

  This function is a better model of the main motor dynamics.  It solves
  the ODEs for a DC motor,

  L_a.i_a_dot = -R_a.i_a - 2.PI.k_phi.n + v_arm - v_brush
  2.PI.J_thruster.n_dot = k_phi.i_a - propeller torque needed

  See Fossen P.97

  double des_v_arm    the desired armature voltage of the DC motor
  double time_step    simulation time step [sec]

*/

void new_motor(double des_speed, double time_step, STATE *s) {

    double dead_zone (double input, double zone_size);
    int sign(double input);
    double torque (STATE *s);
    double omega_dash;
    double omega, back_emf, vs_applied;

    omega=s->RPS*TWO_PI;
    s->prop.k_phi=0.0374-6.17e-5*fabsl(omega)+5.12e-7*omega*omega
            -2.27e-9*fabsl(omega)*omega*omega+7.72e-12*omega*omega*omega*omega;

    s->V_s=9.6*(float)sign(des_speed);
    back_emf=s->prop.k_phi*omega;
    vs_applied=dead_zone(s->V_s,s->prop.v_brush)-back_emf;
                                                    // dead zone due to brush voltage
    s->i_a = vs_applied/s->prop.R_a*fabsl(des_speed)/2500.0;
    s->i_a = dead_zone(s->i_a,9.6*0.01/s->prop.R_a);
```

The increase in speed is equal to the difference in torque developed by the current and the torque required by the propeller at the current speed, all divided by the thruster's inertia

End of **new_motor()** function

Start of **new_thrust()** function

Call function to calculate β, based on prop speed, vehicle speed and propeller size

Calculate thrust force using $F = C_T^* \frac{1}{2}\rho\left[V_A^2 + (0.7\pi nD)^2\right]\frac{\pi}{4}D^2$

End of **new_thrust()** function; return thrust force value

Start of **c_t_star()** function

Calculate C_T^* from fourier data held in state->prop.fourier_thrust[][]

Endt of **c_t_star()** function; return C_T^*

Start of **torque()** function

Call function to calculate β, based on prop speed, vehicle speed and propeller size

```
  if(s->i_a!=0.0) s->RPS_dot=
        (dead_zone(s->prop.k_phi*s->i_a,fabs(0.001*s->RPS))-torque(s))
        /(TWO_PI*s->prop.J_thruster);
        // 0.001*s->RPS term represents frictional losses in the motor
  else s->RPS_dot = 0.0;
  if(des_speed>0 && s->RPS<0) s->RPS=0.0;
  s->RPS += s->RPS_dot * time_step;
  }


double new_thrust (STATE *s) {
  /* This calculates the thrust force using the four quadrant data expression for
     thrusters, i.e.
     Thrust = 0.5*rho*(Va^2+(0.7*pi*n*D)^2)*pi/4*D^2 * C_T_star
     The actual equation has been streamlined for execution speed */
  double c_t_star (STATE *s);
  double calc_beta (double n, double u, double point_7_pi_d);

  double thrust_force;
  double V_a;             /* advance speed */

  thrust_force=0.0;
  V_a = (1.0 - s->prop.wake_fraction) * s->u;
  s->prop.beta1=calc_beta(s->RPS, s->u, s->prop.point_7_pi_d);

  thrust_force = s->density * PI * SQR(s->prop.propeller_diam) *
     0.125 * ( SQR(V_a) + SQR(s->prop.point_7_pi_d) * SQR(s->RPS) ) *
     c_t_star (s);
  return thrust_force;
  }


double c_t_star (STATE *s) {
  double thrust_coeff;
  int count;

  thrust_coeff = s->prop.fourier_thrust[0][0]*0.5;
  for (count=1; count<20; count++)
     thrust_coeff+=s->prop.fourier_thrust[0][count]*cos(count*s->prop.beta1)+
              s->prop.fourier_thrust[1][count]*sin(count*s->prop.beta1);

  return thrust_coeff;
  }


double torque (STATE *s) {
  /* This calculates the torque using the four quadrant data expression for thrusters:
     Torque = 0.5*rho*(Va^2+(0.7*pi*n*D)^2)*pi/4*D^3 * C_Q_star
     The actual equation has been streamlined for execution speed */

  double c_q_star (double beta1, STATE *s);
  double calc_beta (double n, double u, double point_7_pi_d);
  double torque;
  double n;               /* n = propeller revolutions per second */
  double V_a;             /* advance speed */

  torque=0.0;
  n = s->RPS;             /* n = RPS */
  V_a = (1.0 - s->prop.wake_fraction) * s->u;
  s->prop.beta1 = calc_beta(s->RPS, s->u, s->prop.point_7_pi_d);
```

Calculate propeller torque using $Q = C_Q^* \frac{1}{2} \rho \left[V_A^2 + (0.7\pi n D)^2 \right] \frac{\pi}{4} D^3$

End of **torque**() function; return thrust force value

Start of **c_q_star**() function

Calculate C_Q^* from fourier data held in state->prop.fourier_torque[][]

Endt of **c_q_star**() function; return C_Q^*

Start of **dead_zone**() function

Given a certain dead zone or size zone_size around zero, this returns the effect on input; i.e. if $|input| < zone\_size$ you get zero, otherwise it's input less zone_size with the appropriate sign corrections

End of **dead_zone**() function; return the output

Start of **sign**() function

End if **sign**() function; returns -1 if input is -ve, 0 if zero and +1 if +ve

Start of **calc_beta**() function

Calculates β given propeller speed and vehicle speed; makes the appropriate corrections for quadrant

End of **calc_beta**() function; returns β

```
      torque = s->density * PI * SQR(s->prop.propeller_diam) *
      s->prop.propeller_diam * 0.125 *
      ( SQR(V_a) + SQR(s->prop.point_7_pi_d) * SQR(n) ) *
      c_q_star (s->prop.beta1, s);
   return torque;
   }


double c_q_star (double beta1, STATE *s) {
   double torque_coeff;
   int count;

   torque_coeff = s->prop.fourier_torque[0][0]*0.5;
   for (count=1; count<20; count++)
         torque_coeff+=s->prop.fourier_torque[0][count]*cos(count*beta1)+
         s->prop.fourier_torque[1][count]*sin(count*beta1);

   return torque_coeff;
   }


double dead_zone (double input, double zone_size) {
   double output=0.0;

   if (fabs(zone_size)>fabs(input)) output=0.0;
   else if (input>0.0) output=input-zone_size;
   else if (input<0.0) output=input+zone_size;
   return output;
   }


int sign (double input) {
   int temp;
   if (fabsl(input)==input) temp=1;
   else if (fabsl(input)==-input) temp=-1;
   else temp=0;
   return temp;
   }


double calc_beta (double n, double u, double point_7_pi_d) {
   int sign(double input);
   double beta;
   if (n==0) {
      if (u>0) beta=90*TORAD;
      if (u<0) beta=270*TORAD;
      }
   else {
      beta = atan ( u / ( point_7_pi_d * n ));
      if (sign(n)==-1) beta+=PI;
      else if (sign(n)==1 && sign(u)==-1) beta+=TWO_PI;
      }
   return beta;
   }
```

8

INTC_ROV.C

Start of **int_const()** function; this sets the hydrodynamic coefficients for the vehicle

The coefficients are given in their non-dimensional form;
these are the dimensionalising factors based on length;
e.g. $X_{uu} = \frac{1}{2}\rho l^2 X'_{uu}$

Coefficients marked CHANGED have been altered from the FAU data for *Subzero II*

```
/* Roy Lea */
/* File: intc_ROV.c Version: 1.0 */
/* coefficients for Southampton University ROV, based on FAU AUV */
/* To set hydrodynaminc constants */

#include <sub.h>
#include <opt.h>

/*   void int_const(double density, double length, HYDRO_COEFF *c)
     where
              density: is the density of the water, state->density
              length : is the length of the sub,    state->length
                   c : is a pointer to the coefficents structure */

void int_const(double density, double length, HYDRO_COEFF *c) {
   double L2;
   double L3;
   double L4;
   double L5;

   /* to calculate 'normalizing^-1'  factors f(density,length) */
   L2 = 0.5 * density * length * length;
   L3 = L2 * length;
   L4 = L3 * length;
   L5 = L4 * length;

   /*   To set constants for axial state */
   c->axial.X_qq = -3.19547e-3*L4;            /* X as a func of q^2 */
   c->axial.X_rr = -3.19547e-3*L4;            /* X as a func of r^2 */
   c->axial.X_rp = 0.0*L4;                    /* X as a func of rp */
   c->axial.X_u_dot = -1.76505e-4*L3;         /* X as a func of du/dt */
   c->axial.X_vr = -3.01945e-3*L3;            /* X as a func of vr */
   c->axial.X_wq = 3.01945e-3*L3;             /* X as a func of wq */
   c->axial.X_vv = -1.3746e-2*L2;             /* X as a func of vv */
   c->axial.X_ww = -1.3746e-2*L2;             /* X as a func of w^2 */
   c->axial.X_deltar_deltar = -7.0e-3*L2;     // X as a func of u^2(dr)^2 CHANGED
   c->axial.X_deltas_deltas = -7.0e-3*L2;     // X as a func of u^2(ds)^2 CHANGED
   c->axial.X_deltab_deltab = 0.0*L2;         /* X as a func of u^2(db)^2 */
   c->axial.X_uu = 2.0e-3*L2;                 // See SUB_RTSI for Xuu function

   /* NOTE: X_uu was given as '-' but that is fucked up */
   /* since drag would ADD to forward motion 11/7/90 */

   /* Draper coefficents not used for FAU AUV */
   c->axial.X_w_deltas = 0.0*L2;   /* coeff from Draper Labs NOT DTNSRDC Eq */
   c->axial.X_q_deltas = 0.0*L2;   /* coeff from Draper Labs NOT DTNSRDC Eq */
   c->axial.X_v_deltar = 0.0*L2;   /* coeff from Draper Labs NOT DTNSRDC Eq */
   c->axial.X_r_deltar = 0.0*L2;   /* coeff from Draper Labs NOT DTNSRDC Eq */
   c->axial.X_deltaa_deltaa = 0.0*L2;
   /* To set constants for lateral state */
   c->lateral.Y_r_dot = 1.31057e-4*L4;      /* Y func of dr/dt */
   c->lateral.Y_p_dot = 0.0*L4;             /* Y func of dp/dt */
// c->lateral.Y_v_dot = -1.07003e-2*L3;     /* Y func of dv/dt */
   c->lateral.Y_v_dot = -1.5e-2*L3;
   c->lateral.Y_p_p = 0.0*L4;               /* Y func of p|p|   */
   c->lateral.Y_pq = 5.97965e-5*L4;         /* Y func of pq     */
   c->lateral.Y_r = 1.75817e-2*L3;          /* Y func of ur     */
   c->lateral.Y_p = 0.0*L3;                 /* Y func of up     */
   c->lateral.Y_wp = 1.04282e-2*L3;         /* Y func of wp     */
```

```
c->lateral.Y_star = 0.0*L2;             /* Y func of u^2 Y* */
c->lateral.Y_v = -3.6838e-2*L2;         /* Y func of uv    */
c->lateral.Y_v_v_R = 0.0*L2;            /* Y func of v|v|R */
c->lateral.Y_deltar = 2.15515e-2*L2;    /* Y func of u^2(delta r)*/
c->lateral.Y_deltar_eta = 0.0*L2;       /* Y func of u^2dr(eta-1/c)c */

/* To set constants for normal state */
c->normal.Z_q = -1.75817e-2*L3;        /* Z func of uq */
c->normal.Z_vp = -1.04282e-2*L3;       /* Z func of vp */
c->normal.Z_q_dot = -1.31057e-4*L4;    /* Z func of dq/dt*/
c->normal.Z_w_dot = -1.07003e-2*L3;    /* Z func of dw/dt*/
c->normal.Z_star = 0.0*L2;             /* Z func of u^2 */
c->normal.Z_w = -3.6838e-2*L2;         /* Z func of uw*/
c->normal.Z_w_ = 0.0*L2;               /* Z func of u|w| */
c->normal.Z_ww = 0.0*L2;               /* Z func of w*MAG(v,w) */
c->normal.Z_deltas = -2.15515e-2*L2;   /* Z func of u^2(delta s) */
c->normal.Z_deltab = 0.0*L2;           /* Z func of u^2(delta b) */
c->normal.Z_deltas_eta = 0.0*L2;       /* Z func of u^2(ds)(nc -1)*/

/* Draper coefficents not used for FAU AUV */
c->normal.Z_pr = 0.0*L3;   /* coeff from Draper Labs NOT DTNSRDC Eq */

/* To set constants for roll state */
c->roll.K_p = -2.85826e-4*L4;           /* K func of p */
c->roll.K_p_dot = -2.64017e-6*L5;       /* K func of dp/dt */
c->roll.K_i = 0.0*L2;                   /* K due to interference effects */
c->roll.K_vp = 0.0*L4;                  /* K func of vp */
c->roll.K_star = 0.0*L3;                /* K func of u^2 */
c->roll.K_r = 0.0*L4;                   /* K func of ur*/
c->roll.K_r_dot = 0.0*L5;               /* K func of dr/dt */
c->roll.K_p_p = 0.0*L5;                 /* K func of p|p| */
c->roll.K_qr = 0.0*L5;                  /* K func of qr */
c->roll.K_vR = 0.0*L3;                  /* K func of uv */
c->roll.K_v_dot = 0.0*L4;               /* K func of dv/dt */
c->roll.K_wp = 0.0*L4;                  /* K func of wp */
c->roll.K_deltar = 0.0*L3;              /* K func of u^2(delta r) */
c->roll.K_deltar_eta = 0.0*L3;          /* K func of u^2(dr)(nc -1)*/
c->roll.K_4S = 0.0*L3;                  /* K due to phi_s at stern */
c->roll.K_8S = 0.0*L3;                  /* K due to phi_s at stern */
/* Draper coefficents not used for FAU AUV */
c->roll.K_wr = 0.0*L4;           /* coeff from Draper Labs NOT DTNSRDC Eq */
c->roll.K_deltaa = 0.0*L3;    /* coeff from Draper Labs NOT DTNSRDC Eq */

/*  To set constants for pitch state */
c->pitch.M_star = 0.0*L3;              /* M func of u^2 */
c->pitch.M_q = -8.37328e-3*L4;         /* M func of uq */
c->pitch.M_q_dot = -7.71194e-4*L5;     /* M func of dq/dt */
c->pitch.M_q_dot *=5.0;
c->pitch.M_rp = 7.51053e-4*L5;         /* M func of rp */
c->pitch.M_w = -7.33571e-3*L3;         /* M func of uw */
c->pitch.M_w_dot = -1.31057e-4*L4;     /* M func of dw/dt */
c->pitch.M_w_ = 0.0*L3;                /* M func of u|w| */
c->pitch.M_w_w_R = 0.0*L3;             /* M func of w*MAG(v,w) */
c->pitch.M_ww = 0.0*L3;                /* M func of|w*MAG(v,w)|*/
c->pitch.M_deltab = 0.0*L3;            /* M func of u^2(delta b) */
c->pitch.M_deltas = -1.2e-2*L3         /* M func of u^2(delta s) CHANGED */
c->pitch.M_deltas_eta = 0.0*L3;        /* >>> Not listed <<< */

/* Draper coefficents not used for FAU AUV */
c->pitch.M_vp = -3.976e-5*L4;      /* coeff from Draper Labs NOT DTNSRDC Eq */
c->pitch.M_v_deltaa = 0.0*L4;      /* coeff from Draper Labs NOT DTNSRDC Eq */
c->pitch.M_r_deltaa = 0.0*L4;      /* coeff from Draper Labs NOT DTNSRDC Eq */
```

End of **int_const()** function

```
/* To set constants for yaw state */
c->yaw.N_star = 0.0*L3;               /* N func of u^2 */
c->yaw.N_p = 0.0*L4;                  /* N func of up */
c->yaw.N_p_dot = 0.0*L5;              /* N func of dp/dt */
c->yaw.N_pq = -7.51053e-4*L5;         /* N func of pq */
c->yaw.N_r = -8.37328e-3*L4;          /* N func of ur */
c->yaw.N_r_dot = -7.71194e-4*L5;      /* N func of dr/dt */
c->yaw.N_r_dot *=5.0;
c->yaw.N_v = 7.33571e-3*L3;           /* N func of uv */
c->yaw.N_v_dot = 1.31057e-4*L4;       /* N func of dv/dt */
c->yaw.N_v_v_R = 0.0*L3;              /* N func of v*MAG(v,w) */
c->yaw.N_deltar = -1.2e-2*L3;         /* N func of u^2(delta r) CHANGED */
c->yaw.N_deltar_eta = 0.0*L3;         /* N func of u^2(dr)(nc -1)*/

/* Draper coefficents not used for FAU AUV */
c->yaw.N_w_deltaa = 0.0*L4;         /* coeff from Draper Labs NOT DTNSRDC Eq */
c->yaw.N_q_deltaa = 0.0*L4;         /* coeff from Draper Labs NOT DTNSRDC Eq */
c->yaw.N_deltas_deltaa = 0.0*L4;    /* coeff from Draper Labs NOT DTNSRDC Eq */
}
```

9

INTS_ROV.C

Start **int_state()** function

These have been changed from the FAU data for *Subzero II*

```c
/* Roy Lea */
/* File: ints_ROV.c Version 1.00 */
/* To set all the state structure variables to initial values  */
/* The effective values are set in dimensional form   */
/* For use with the Southampton University ROV */
/* ----------- */

#include <math.h>
#include <stdio.h>

#include <sub.h>
#include <rov_ext.h>

void int_state(STATE *s, HULL_SHAPE *h) {
   double x;                /* length along the hull in [m] */
   int count;               /* loop counter used to fill CFD stations */
   int count1;              /* and fourier coefficients for 4-quad */
   double temp[2][21];      // dummy variable for fourier coefficients

   s->density = 1000.0;     /* mass density of water in kg/m3 */
   s->mass = 7.0;           /* mass of vehicle in kg */
   s->weight = 69.1;        /* weight of vehicle in N */
   s->B = 69.0;             /* buoyancy force of envelope displacement in N */
   s->c = 1.0;              /* modeling thrust,drag to full scale */
   s->u = 0.0;              /* velocity component in x-axis */
   s->u_dash_kminus1 = 0.0;
   s->u_dash_kminus2 = 0.0;
   s->v = 0.0;              /* velocity component in y-axis */
   s->w = 0.0;              /* velocity component in z-axis */
   s->p = 0.0;              /* angular velocity about x-axis (roll)  */
   s->q = 0.0;              /* angular velocity about y-axis (pitch) */
   s->r = 0.0;              /* angular velocity about z-axis (yaw)   */
   s->u_dot = 0.0;          /* acceleation component in x-axis */
   s->v_dot = 0.0;          /* acceleation component in y-axis */
   s->w_dot = 0.0;          /* acceleation component in z-axis */
   s->p_dot = 0.0;          /* angular acceleation about x-axis */
   s->q_dot = 0.0;          /* angular acceleation about y-axis */
   s->r_dot = 0.0;          /* angular acceleation about z-axis */
   s->theta = -10.0*TORAD;         /* angle of pitch */
   s->phi = 0.0;            /* angle of roll  */
   s->psi = 40.0*TORAD;           /* angle of yaw */
   s->psi_dash_kminus1 = 0.0;
   s->psi_dash_kminus2 = 0.0;
   s->theta_dot = 0.0;      /* rate of change angle of pitch */
   s->phi_dot = 0.0;        /* rate of change angle of roll */
   s->psi_dot = 0.0;        /* rate of change angle of yaw */
   s->x_o = 0.0;            /* a coordinate of displacement re fixed axes */
   s->y_o = 0.0;            /* a coordinate of displacement re fixed axes */
   s->z_o = 0.38;            /* a coordinate of displacement re fixed axes */
   s->z_dash_kminus1 = 0.0;
   s->x_o_dot = 0.0;        /* rate of change of coordinate displacement*/
   s->y_o_dot = 0.0;        /* rate of change of coordinate displacement*/
   s->z_o_dot = 0.0;        /* rate of change of coordinate displacement*/
   s->U = 0.0;              /* velocity of origin of body re fluid */
   s->alpha = 0.0;          /* angle of attack */
   s->beta = 0.0;           /* angle of drift */

   /* Unormalize moments of inertia in slug ft^2 */
```

The moments of inertia have been calculated for *Subzero II*

As have the centres of mass and buoyancy

```
s->I_x = 0.007;          /* moment of inetia about x-axis */
s->I_y = 0.39;           /* moment of inetia about y-axis */
s->I_z = 0.39;           /* moment of inetia about z-axis */
s->I_xy = 0.0;           /* Prod of inetia w.r.t x and y axes */
s->I_yz = 0.0;           /* Prod of inetia w.r.t y and z axes */
s->I_zx = -0.003;        /* Prod of inetia w.r.t z and x axes */

/* Unormalize CG and CB coordinates in ft. */

s->X_G = 0.025;          /* The x coodinate of CG (origin is centre of vehicle) */
                         // CoG is currently 45.5cm behind the nose
s->Y_G = 0.0;            /* The y coodinate of CG */
s->Z_G = 0.014;          /* The z coodinate of CG */
s->X_B = 0.025;
s->Y_B = 0.0;
s->Z_B = 0.0;

s->deltar = 0.0;        /* deflection of rudder in rad. */
s->deltar_kminus1 = 0.0;
s->deltas = 0.0;        /* deflection of sternplane in rad. */
s->deltas_kminus1 = 0.0;
s->deltab = 0.0;        /* deflection of bowplane or sailplane in rad.*/
s->deltaa = 0.0;        /* aileron deflection of stern planes in rad. */
s->RPS = 0.0;           /* rev per sec of prop >> NOT LISTED << */
s->RPS_dot = 0.0;
s->RPS_kminus1 = 0.0;
s->RPS_kminus2 = 0.0;
s->V_s = 0.0;
s->V_s_kminus1 = 0.0;
s->i_a = 0.0;
s->i_a_dot = 0.0;
s->eta = 1.0;           /* ratio u_c/u */

s->Cd = 1.9*SLUGFT3toKGM3;        /* coeff used in integrating forces and */
                                  /* moments along hull due to local cross-flow */

s->F_xp = 0.0;          /* net thrust - drag */
s->v_s = 0.0;           /* vel comp. in y-axis dir at the quarter  */
                          /* chord of the sternplanes. v_s= v + x_sr */

s->w_s = 0.0;           /* vel comp. in y-axis dir at the quarter */
                          /* chord of the sternplanes. w_s= w + x_sq */

s->Q_p = 0.0;           /* contribution of propeller torque to K and */
                        /* machinery equation */
sen.speed=s->u;
sen.u_dot=0.0;
sen.depth=s->z_o;
sen.pitch=s->theta;
sen.q=0.0;
sen.roll=s->phi;
sen.heading=s->psi;
sen.r=0.0;
smc_speed_int_term=0.0;
smc_heading_int_term=0.0;
smc_depth_int_term=0.0;

s->prop.K1 = 7.6e-3/3600.0;   /* for thrust v. rpm */
s->prop.K2 = 0.0;             /* coefficent used to model thruster horsepower */
s->prop.K3 = 0.0;             /* coefficent usted to model thruster tourqe */
s->prop.beta1 = 0.0;
s->prop.propeller_diam = 0.1; /* ROV propeller is 0.1m in diameter */
```

Fourier terms for the thrust coefficient

Fourier terms for the torque coefficient

```
s->prop.point_7_pi_d = 0.7*PI*s->prop.propeller_diam;
s->prop.wake_fraction = 0.0;  /* i.e. has no effect */
s->prop.R_a = 1.8;
s->prop.L_a = 74e-6;          /* I think! */
s->prop.k_phi = 0.034;
s->prop.v_brush = 0.019;
s->prop.J_thruster = 165e-6;  /* I think! */

// cos terms                 sin terms
temp[0][0]=3.7890E-02;      temp[1][0]=0;
temp[0][1]=5.6541E-02;      temp[1][1]=-5.2494E-02;
temp[0][2]=-5.3121E-03;     temp[1][2]=4.3486E-03;
temp[0][3]=1.6108E-02;      temp[1][3]=-1.5705E-02;
temp[0][4]=1.2673E-03;      temp[1][4]=1.0364E-03;
temp[0][5]=4.6195E-03;      temp[1][5]=7.2586E-03;
temp[0][6]=-1.1210E-03;     temp[1][6]=-2.9248E-03;
temp[0][7]=1.6218E-03;      temp[1][7]=1.5062E-03;
temp[0][8]=5.1023E-05;      temp[1][8]=1.2025E-03;
temp[0][9]=-4.6598E-04;     temp[1][9]=-7.2472E-04;
temp[0][10]=1.2832E-03;     temp[1][10]=-3.0287E-05;
temp[0][11]=-3.0581E-04;    temp[1][11]=6.8235E-04;
temp[0][12]=-2.0177E-04;    temp[1][12]=-1.6953E-04;
temp[0][13]=3.8367E-04;     temp[1][13]=-3.8835E-04;
temp[0][14]=7.6063E-05;     temp[1][14]=3.0408E-04;
temp[0][15]=1.1454E-04;     temp[1][15]=-4.4988E-04;
temp[0][16]=3.9951E-04;     temp[1][16]=1.1371E-04;
temp[0][17]=1.6368E-04;     temp[1][17]=8.7497E-05;
temp[0][18]=-7.2696E-05;    temp[1][18]=-1.7994E-05;
temp[0][19]=3.4252E-04;     temp[1][19]=-5.7587E-06;
temp[0][20]=-1.6216E-04;    temp[1][20]=2.0364E-05;
for (count=0;count<21;count++) {
   for (count1=0;count1<2;count1++)
         s->prop.fourier_thrust[count1][count]=temp[count1][count];
   }

 temp[0][0]= 1.2684e-3;  temp[1][0]= 0.0;
 temp[0][1]= 4.0814e-3;  temp[1][1]=-3.9452E-03;
 temp[0][2]=-5.7959e-4;  temp[1][2]=-2.5945E-04;
 temp[0][3]=-3.3078e-5;  temp[1][3]=-1.0692E-03;
 temp[0][4]=-4.3096e-4;  temp[1][4]= 7.2242E-04;
 temp[0][5]= 2.5227e-4;  temp[1][5]= 8.0111E-04;
 temp[0][6]= 7.3861e-5;  temp[1][6]=-4.5252E-05;
 temp[0][7]=-6.8829e-5;  temp[1][7]= 2.6193E-04;
 temp[0][8]= 1.7370e-4;  temp[1][8]= 1.1533E-04;
 temp[0][9]= 7.5918e-5;  temp[1][9]=-6.9891E-05;
temp[0][10]= 1.0064e-5; temp[1][10]=-6.3303E-05;
temp[0][11]=-5.0233E-6; temp[1][11]=-2.5880E-05;
temp[0][12]= 2.4663E-5; temp[1][12]=-1.5836E-05;
temp[0][13]= 2.7808E-5; temp[1][13]=-2.1279E-05;
temp[0][14]= 3.3590E-5; temp[1][14]=-2.6672E-05;
temp[0][15]= 1.8183E-5; temp[1][15]=-1.4481E-05;
temp[0][16]= 2.2763E-5; temp[1][16]=-2.2023E-06;
temp[0][17]= 1.2086E-5; temp[1][17]=-3.4391E-06;
temp[0][18]= 1.1185E-5; temp[1][18]= 1.2311E-05;
temp[0][19]= 1.1605E-5; temp[1][19]= 9.1907E-07;
temp[0][20]=-3.4426E-6; temp[1][20]= 3.5035E-06;
for (count=0;count<21;count++) {
   for (count1=0;count1<2;count1++)
         s->prop.fourier_torque[count1][count]=temp[count1][count];
   }

s->X = 0.0;            /* X force in vehicle coordinates */
```

Sets up the hull shape for the cross-flow integral function using a three-part model —
the nose which is hemispherical,
the centre hull which is cylindrical
and the tail section which is conical;
hull->R[] is the radius out from the centre in metres
(so it only works for an axisymmetric hull)

Although the Fourier thrust and torque coefficients are currently hardcoded in the program, they could be loaded from a file if so wanted

End of **int_state**() function

```
    s->Y = 0.0;              /* Y force in vehicle coordinates */
    s->Z = 0.0;              /* Z force in vehicle coordinates */
    s->K = 0.0;              /* Rolling moment in vehicle coordinates */
    s->M = 0.0;              /* Pitching moment in vehicle coordinates */
    s->N = 0.0;              /* Yawing moment in vehicle coordinates */
    s->CFY = 0.0;            /* Y force from cross flow integral */
    s->CFZ = 0.0;            /* Z force from cross flow integral */
    s->CFM = 0.0;            /* Pitching moment from cross flow integral */
    s->CFN = 0.0;            /* Yawing moment from cross flow integral */

    /* INITILIZE THE HULL STRUCTURE */
    /* To fill radius of hull along length starting from the nose station 0 */
    /* to NO_STATIONS-1 or calculate the values from line(s) of best fit. */

    h->length = 0.97; /* overall length of vehicle */
    h->x_B = 0.43;     /* 0.5*s->length - s->X_G;  distance from CG to the Bow */
    h->x_AP = -0.54;   /* -(0.5*s->length + s->X_G); from CG to the AP (stern) */
    h->x_s = -0.49;    /* x-co-ord of quarter cord of the sternplanes */

    /* assign coeff for poly-best-fit for tail sections ADD LATER*/
    /* right now use imbeded magic numbers AS JK 4-16-91 */

    h->inc = (double) (h->length / (NO_STATIONS-1));
    count = 0;
    x = 0.0;
    while( count < NO_STATIONS ) {
        h->x[count] = x;
        h->R[count] = 0.0;
        if((x>=0.0)&&(x<=0.05)) { /* NOSE only  */
          h->R[count] = (sqrt(2.501e-3-SQR(x-0.05)));
          if(h->R[count]>0.05)  h->R[count] = 0.05;
          }
        if((x>0.05)&&(x<=0.86))  /* mid body section */
            h->R[count]=0.05;
        if((x>0.86)&&(x<=h->length))   /* tail section */
            h->R[count]=0.05-0.04*(x-0.86)/0.11;
        if( h->R[count] < 0.0 ) h->R[count] = 0.0;
        x += h->inc;
        count++;
        }

    /* fourier_thrust[1,n] are the sin coefficients for thrust
       fourier_thrust[2,n] are the cos coefficients for thrust */
/*
    for (count=0; count<=1; count ++) {
        for (count1=0; count1<=20; count1 ++)
            fscanf("4-quad.dat", "%f", &s->prop.fourier_thrust[count][count1]);
        }

    for (count=0; count<=1; count ++) {
        for (count1=0; count1<=20; count1 ++)
            fscanf("4-quad.dat", "%f", &s->prop.fourier_torque[count][count1]);
        }
*/

    } /* void int_state() */
```

10

ROV_INT.C

Start of **int_eq_motion()** function

Only compile the fuzzy logic and self-tuning initialization functions if we're using those autopilots (to save memory)

Initialize and set the state and other model variables

Initialize and set the hydrodynamics coefficients

Initialize and set the fuzzy logic variables if we're using it

Initialize and set the self-tuner variables if we're using it

Initialize and set the Kalman filter variables
Initialize and set the tether variables
Initialize and set the mass matrix (masses, inertias and added masses/inertias)

```
/* John Kloske  4/21/91    Rev: 11-25-91 */
/* Andy Shein */
/* File: int_params.c Version: 2.0  */
/* functions to initilize dynamics  */

#include <stdio.h>  /* fprintf() */
#include <process.h>  /* exit() */

#include <sub.h>
#include <opt.h>

/*void int_eq_motion(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull,
                     STATUS *stat, double A[MATRIX_SIZE][MATRIX_SIZE],
                     int debug_flag)
   Initilize simulation dynamics for a run.
   If debug flag TRUE prints out mass matrix and inverse of the mass
   matrix.
*/

void int_eq_motion(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull, STATUS *stat,
      STR_VARIABLES *str_var, double A[MATRIX_SIZE][MATRIX_SIZE], int debug_flag){
   void int_state(STATE *s, HULL_SHAPE *h);
   void int_const(double density, double length, HYDRO_COEFF *c);
#if(CONTROL_TYPE==FUZZY)
   void fuzzy_init (void);
#endif
#if(CONTROL_TYPE==STR)
   void str_init (STR_VARIABLES *str_var);
#endif
   void print_mass_mat(double A[MATRIX_SIZE][MATRIX_SIZE]);
   int invmat(int n, double a[MATRIX_SIZE][MATRIX_SIZE]);
   void get_mass_mat(double A[MATRIX_SIZE][MATRIX_SIZE], STATE *s, HYDRO_COEFF *c);
   void kalman_init(STATE *s);
   void tether_init(STATE *s);

   int_state(state, hull);                /* initilize state structure */

   int_const(state->density, hull->length, coeff);   /* load hydro coeff */

#if(CONTROL_TYPE==FUZZY)
   fuzzy_init();
#endif
#if(CONTROL_TYPE==STR)
   str_init(str_var);
#endif
   kalman_init(state);
   tether_init(state);
   get_mass_mat(A, state, coeff);                    /* Load mass matrix */

   if( debug_flag == TRUE) {          /* if debuging print out mass matrix */
      (void)printf("MASS MATRIX \n");
      print_mass_mat(A);
      }

   if( invmat(stat->num_eq, A) == ERROR ) {   /* get inverse of matrix A[][] */
      (void)fprintf(stderr, "ERROR IN inv(A)\n");
      exit(ERROR);
      }
```

End of **int_eq_motion()** function

Start of **int_status()** function

End of **int_status()** function

```
   if (debug_flag == TRUE) {     /* if debuging print out inverse mass matrix */
      (void)printf("INVERSE MASS MATRIX \n");
      print_mass_mat(A);
      }
   }

/*---------------------------------------------------------------------------*/

/*
     void int_status( STATUS *stat)
     initilize simulation status structure with defaults
*/

void int_status(STATUS *stat) {
   stat->step_size = 0.01;       /* step size in [sec] used to solve the ODE's */
   stat->sim_time = 0.0;         /* simulation time in [sec] */
   stat->print_time = 5.0;       /* time in sec[] to print header or send state */
                                 /* over socket */

   stat->sim_count = 0;          /* number of times solve_ODE has been called in */
                                 /* simulation since last reset */

   stat->print_count = 0;        /* print interval converted to counts */
   stat->ODE_method = EULER;     /* method used to sove the ODE's */
   stat->CFD_method = SIMPSON;   /* method used to integrate cross flow */
   stat->num_eq = NUM_EQ;        /* six equations to be solved */
   stat->dev_flag = FALSE;       /* flag for initilization of Runge-Kutta */
   }
```

11

SUB_SOLV.C

The last two numerical methods have been removed

Start of **solv_eq_motion**() function

Call apropriate solving function depending on the numerical method being used

Update the global variables (Euler angles/transformation)...
...and call the tether solving function (function is empty if not being used)

End of **solv_eq_motion**() function

```
/* John Kloske 12-12-90  Rev: 11-25-91 */
/* Andrew Shein */
/* File: sub_solv.c Version: 2.0 */
/* Functions to solve the ode's and update the auxillary */
/* equations */

#include <math.h>   /* sin(), cos(), sqrt(), atan(), asin() */
#include <stdio.h>  /* fprintf() */
#include <process.h>  /* exit() */

#include <sub.h>
#include <opt.h>
#include <rov_ext.h>

/* void solve_eq_motion(STATE *s, HYDRO_COEFF *c, HULL_SHAPE *h, STATUS *stat,
          double A[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE] )

    I. To solve the set of 6 ODE's using method >
         1) Euler ---> DEFAULT method
         2) Improved Euler
         3) Runge_kutta
         4) Adams-Moulton (predictor-corrector method)

   II. Velocities [u,v,w,p,q,r] solved for are assigined in var (s).

  III. All auxiliary equations are updated by call to:  aux_eq()

   IV. Notes:
        method      : number from 1-4 for methods listed above.
        A[][]       : the inverse of the apperent mass matrix.
        b[]         : right hand side of the 6 equations.
        stat        : provides method of integration for cross flow and time step.
*/

void solve_eq_motion( STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull, STATUS *stat,
      double A[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE]) {
   void euler(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull, STATUS *stat,
         double A[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE], double time_step,
         int CFD_method);
   void improved_euler(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull, STATUS *stat,
         double A[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE], double time_step,
         int CFD_method);

   void aux_eq(STATE *s, double time_step);
   void tether(STATE *s, double time_step);
   switch(stat->ODE_method) {
      case EULER: euler(state, coeff, hull, stat, A, b, stat->step_size,
            stat->CFD_method); break;
      case IMP_EULER: improved_euler(state, coeff, hull, stat, A, b, stat->step_size,
            stat->CFD_method); break;
      default: euler(state, coeff, hull, stat, A, b, stat->step_size, stat->CFD_method);
      }

   aux_eq(state, stat->step_size);   /* set euler angles and world position */
   tether(state, stat->step_size);   // do the tether dynamics

   stat->sim_count++;                 /* calculate current time */
   stat->sim_time = stat->sim_count * stat->step_size;
   }
```

Start of **euler()** function

Find the vector of forces...
...then $F = ma \Rightarrow a = m^{-1}F$ (or rather, the 6DOF version $\dot{\boldsymbol{v}} = \boldsymbol{M}^{-1}\boldsymbol{f}$)

If an external water current disturbance is being used (see OPT.H), add it to the force vector

Update the velocities, based on
$\boldsymbol{v}_{k+1} = \boldsymbol{v}_k + \Delta t . \dot{\boldsymbol{v}}_k$

If the straight-line tether model is being used, update the length based on $length_{k+1} = length_k + \Delta t . vehicle\ velocity_k$

End of **euler()** function

```
/* void euler(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull, STATUS *stat,
          double A[MATRIX_SIZE][MATRIX_SIZE],
          double b[MATRIX_SIZE],
          double time_step, int CFD_method)

      To solve the set of 6 ODE's using Euler method one time step.
      Velocities [u,v,w,p,q,r] solved for are assigined in var (s).

        *s          : struct holding velocities.
        A[][]       : the inverse of the apperent mass matrix.
        b[]         : right hand side of the 6 equations.

       time_step    : integration time step in [sec]
       method       : method of integration for cross flow.
*/

void euler(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull, STATUS *stat,
      double A[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE], double time_step,
      int CFD_method) {

   void get_const_mat(double const_mat[MATRIX_SIZE], STATE *state,
         HYDRO_COEFF *coeff, HULL_SHAPE *hull, STATUS *stat, int CFD_method);
   void mult_mat(double a[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE],
         double c[MATRIX_SIZE]);
   double sum[MATRIX_SIZE];    /* results at f(Yn) = Y'n, position 0 not used */

   /* To solve right hand side of equations (constants) for next step */

   get_const_mat(b, state, coeff, hull, stat, CFD_method);
   mult_mat(A,b,sum);    /* mass matrix^-1 * b[] */

   /* The forcing function is acually -0.5sin(0.2t)sin(PI/2-psi) from 90deg
      laterally but of course it has to be differentiated. */
#if (DISTURBANCE==TRUE)
   sum[AXIAL] += -0.1 * cos(0.2*stat->sim_time) * sin(state->psi);
   sum[LATERAL] += -0.1 * cos(0.2*stat->sim_time) * sin(PIBY2 - state->psi);
#endif

   state->u_dot = sum[AXIAL];
   state->u += time_step * sum[AXIAL];
   state->v += time_step * sum[LATERAL];
   state->w += time_step * sum[NORMAL];
   state->p += time_step * sum[ROLL];
   state->q += time_step * sum[PITCH];
   state->r += time_step * sum[YAW];
   state->r_dot = sum[YAW];

#if(SL_TETHER_DRAG)
   t.sl_length += time_step*state->u;
#endif
   }


/* void improved_euler(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull, STATUS *stat,
          double A[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE],
          double time_step,int CFD_method)

       To solve the set of 6 ODE's using Improved Euler method one time step.
     Velocities [u,v,w,p,q,r] solved for are assigined in var (s).

        *state       : struct holding velocities.
        A[][]        : the inverse of the apperent mass matrix.
```

Start of **improved_euler()** function

Calculate $\dot{\boldsymbol{v}}_k$

Temporarily update the velocities, based on
$\boldsymbol{v}_{k+1} = \boldsymbol{v}_k + \Delta t.\dot{\boldsymbol{v}}_k$

Calculate $\dot{\boldsymbol{v}}_{k+1}$ based on the estimates of $\boldsymbol{v}_{k+1}$

Update the velocities, based on

$$\boldsymbol{v}_{k+1} = \boldsymbol{v}_k + \frac{\Delta t}{2}\left(\dot{\boldsymbol{v}}_k + \dot{\boldsymbol{v}}_{k+1}\right)$$

If the straight-line tether model is being used,
update the length based on $length_{k+1} = length_k + \Delta t.\,vehicle\ velocity_k$

End of **improved_euler()** function

```
        b[]           : right hand side of the 6 equations.
        CFD_method    : method of integration for cross flow.
*/

void improved_euler(STATE *state, HYDRO_COEFF *coeff, HULL_SHAPE *hull,
       STATUS *stat, double A[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE],
       double time_step, int CFD_method){
   void get_const_mat(double const_mat[MATRIX_SIZE], STATE *state, HYDRO_COEFF *coeff,
          HULL_SHAPE *hull, STATUS *stat, int CFD_method);
   void mult_mat(double a[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE],
          double c[MATRIX_SIZE]);

   double sum[MATRIX_SIZE];          /* f(Yn + hY'n), position 0 not used */
   double vel[MATRIX_SIZE];          /* array to hold current velocities */
   double last_step[MATRIX_SIZE];    /* f(Yn) = Y'n  */

   /* To copy current velocities into array vel[] */
   vel[AXIAL] = state->u;
   vel[LATERAL] = state->v;
   vel[NORMAL] = state->w;
   vel[ROLL] = state->p;
   vel[PITCH] = state->q;
   vel[YAW] = state->r;

   /* To solve right hand side of equations (constants) for next step */

   get_const_mat(b, state, coeff, hull, stat, CFD_method);
   mult_mat(A,b,last_step); /* mass matrix^-1 * b[]   f(Yn) = Y'n */

   /* To assign new velocities to *s to get f(Yn + hY'n)  */

   state->u = vel[AXIAL]   + time_step*last_step[AXIAL];
   state->v = vel[LATERAL] + time_step*last_step[LATERAL];
   state->w = vel[NORMAL]  + time_step*last_step[NORMAL];
   state->p = vel[ROLL]    + time_step*last_step[ROLL];
   state->q = vel[PITCH]   + time_step*last_step[PITCH];
   state->r = vel[YAW]     + time_step*last_step[YAW];

   /* To solve right hand side of equations (constants) for next step */

   get_const_mat(b, state, coeff, hull, stat, CFD_method);
   mult_mat(A,b,sum); /* mass matrix^-1 * b[]    f(Yn + hY'n) */

   state->u = vel[AXIAL]   + 0.5*time_step*(last_step[AXIAL]   + sum[AXIAL]);
   state->v = vel[LATERAL] + 0.5*time_step*(last_step[LATERAL] + sum[LATERAL]);
   state->w = vel[NORMAL]  + 0.5*time_step*(last_step[NORMAL]  + sum[NORMAL]);
   state->p = vel[ROLL]    + 0.5*time_step*(last_step[ROLL]    + sum[ROLL]);
   state->q = vel[PITCH]   + 0.5*time_step*(last_step[PITCH]   + sum[PITCH]);
   state->r = vel[YAW]     + 0.5*time_step*(last_step[YAW]     + sum[YAW]);
   state->u_dot = (state->u-vel[AXIAL])/time_step;
   state->r_dot = (state->r-vel[YAW])/time_step;

#if(SL_TETHER_DRAG)
   t.sl_length += time_step*state->u;
#endif
   }


// RUNGE_KUTTA AND ADAMS HAVE BEEN REMOVED
```

Start of **aux_eq()** function

Watch out for the singularity when the vehicle is vertical and θ=90° ($\cos\theta = 0$)

Calculate vehicle's velocities relative to the earth frame

Update the Euler angles of global roll, pitch and heading

```
/* void aux_eq( STATE *s, double time_step )

  To calculated and update the following variables:

      phi, theta, psi              --> angle of: roll, pitch and yaw
      phi_dot, theta_dot, psi_dot, --> rate of change in angles
      alpha, beta                  --> angle of attack and angle of drift
      x_o, y_o, z_o                --> a co-ord of the displacement
      x_o_dot, y_o_dot, z_o_dot,   --> rate of change of co-ord`s
      U                            --> velocity of origin of body

   From:

      STATE *s              State of the vehicle
      double time_step;     Integration time step [sec]

   Order Checked at DTRC by That guy Rick that works for Jerry 6-91

   1. Solve phi_dot, theta_dot and psi_dot with current values of p,q,r, phi,theta and
      psi.

   2. Solve for x_dot, y_dot, and z_dot with current values of p,q,r, phi,theta and psi.

   3. Solve for phi,theta, psi, x,y, and z.

   4. calculate U, alpha, and beta.

*/

void aux_eq(STATE *s, double time_step) {
   double temp;

   s->phi_dot = s->p + s->psi_dot*sin(s->theta);
   s->theta_dot = s->q*cos(s->phi) - s->r*sin(s->phi);

   temp = cos(s->theta);
   if(temp!=0.0) s->psi_dot=(s->r*cos(s->phi)+s->q*sin(s->phi))/cos(s->theta);
   else s->psi_dot = 0.0;  /* check default  Look into this JK 2/4/91 */

   s->x_o_dot = s->u*cos(s->theta)*cos(s->psi)
          + s->v*( sin(s->phi)*sin(s->theta)*cos(s->psi) - cos(s->phi)*sin(s->psi) )
          + s->w*( sin(s->phi)*sin(s->psi) + cos(s->phi)*sin(s->theta)*cos(s->psi) );

   s->y_o_dot = s->u*cos(s->theta)*sin(s->psi)
          + s->v*( cos(s->phi)*cos(s->psi) + sin(s->phi)*sin(s->theta)*sin(s->psi) )
          + s->w*( cos(s->phi)*sin(s->theta)*sin(s->psi) - sin(s->phi)*cos(s->psi) );

   s->z_o_dot = -s->u*sin(s->theta) + s->v*cos(s->theta)*sin(s->phi)
      + s->w*cos(s->theta)*cos(s->phi);
   /* update angles and distances */
   s->phi += time_step * s->phi_dot;
   s->theta += time_step * s->theta_dot;   /* time [sec] * rate [rad/sec] */
                    /* I THINK the + or - should fall */
                    /* out from calculating theta_dot */
   s->psi += time_step * s->psi_dot;

   /* To make sure that the Yaw angles in the correct range 0-360 deg */

   if( s->psi > TWO_PI) s->psi = s->psi - TWO_PI;
   if( s->psi < 0.0 ) s->psi = TWO_PI + s->psi;

   /* update distance */
```

Update the vehicle's position relative to the earth

Check that the vehicle is moving fast enough for certain angle calculations to be valid

End of **aux_eq**() function

```
s->x_o += time_step * s->x_o_dot;
s->y_o += time_step * s->y_o_dot;
s->z_o += time_step * s->z_o_dot;
s->U = sqrt( SQR(s->u) + SQR(s->v) + SQR(s->w) );

/* velocity U,u must be greater for alpha */
/* and beta to be calculated. [ft/sec] */
/* Added on 3-14-91 to correct problem */
/* of alpha and beta holding some very small */
/* number even if sub is not moving. */
/* possibly incorrect */

if( s->u >= VEL_REL_ERR ) s->alpha = atan(s->w/s->u);
else s->alpha = 0.0;  /* check default */

if( s->U >= VEL_REL_ERR) s->beta = asin( -s->v / s->U);
else s->beta = 0.0;  /* check default */
}
```

12

SUB_MATH.C

Start of **gauss()** function

```
/* John Kloske  11-12-90   Rev: 11-25-91     */
/* Andrew Shein */
/* File: sub_math.c Version: 2.0 */
/* methods to integrate cross flow and mass matrix manipulation */

#include <math.h>   /* fabs() */

#include <sub.h>

/* int gauss(int n, int m, double a[][2*COEFF_SIZE])
      Computes the solution for a system of equation with (n) equations and (n) unknowns
      using Gaussian elimination.

          n      : Number of equation/unknows
          m      : number of systems
          a[][] : matrix of [n x m]

          function returns 0 if no errors are detected,
          returns -1 if matrix is singular or division by zero.

Note: General program taken from: "An Introduction to Numerical Computations", Sidney
Yakowits and Ferenc Szidarovszky.
*/

int gauss(int n, int m, double a[MATRIX_SIZE][2*MATRIX_SIZE]) {
   double u, x;   /* temp variables */
   int k, kk, in, ie, i, j;      /* loop counters etc... */

   if( n > 1 ) {
      for( k = 1; k < n; k++) {
         u = fabs(a[k][k]);
         kk = k + 1;
         in = k;

         /* search for index in of maximum pivot value */
         for( i = kk; i <= n; i++) {
            if( fabs(a[i][k]) > u) {
               u = fabs(a[i][k]);
               in = i;
               }
            } /* end for i */

         if( k != in ) {
            for( j = k; j <= n+m; j++) {    /* interchange rows k and index in */
               x = a[k][j];
               a[k][j] = a[in][j];
               a[in][j] = x;
               }
            }

         if( u < PRECISION ) {    /* check if pivot too small */
            return(-1);  /* matrix is singular */
            }

         for( i = kk; i <= n; i++) {  /* forward elimination step */
            for( j = kk; j <= n+m; j++ ) {
                if( a[k][k] != 0.0 ) a[i][j] += -a[i][k]*a[k][j] / a[k][k];
                else return(-1);  /* division by zero */
                }
```

End of **gauss()** function

Start of **invmat()** function

Note that the matrices used here go from [1] to [6], not [0] to [5]

The gaussian elimination function requires as its input

$$M = \begin{bmatrix} x & x & x & x & x & x & x & x & x & x & x & x & x & x \\ x & a & b & c & e & d & g & x & 1 & 0 & 0 & 0 & 0 & 0 \\ x & h & i & j & k & l & m & x & 0 & 1 & 0 & 0 & 0 & 0 \\ x & n & o & p & q & r & s & x & 0 & 0 & 1 & 0 & 0 & 0 \\ x & t & u & v & w & y & z & x & 0 & 0 & 0 & 1 & 0 & 0 \\ x & aa & bb & cc & dd & ee & ff & x & 0 & 0 & 0 & 0 & 1 & 0 \\ x & gg & hh & ii & jj & kk & ll & x & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

where x is don't care, and a, b, etc are part of the matrix to be inverted

End of **inv_mat()** function; returns 0 if successful

```
            }
         } /* end for k */

      if( fabs(a[n][n]) < PRECISION ) return(-1);  /* division by zero */

      for( k = 1; k <= m; k++) {  /* back substitution */
         a[n][n+k] = a[n][n+k] / a[n][n];
         for( ie = 1; ie < n; ie++) {
            i = n - ie;
            in = i + 1;
            for( j = in; j <= n; j++) a[i][n+k] += -a[j][n+k]*a[i][j];
            a[i][n+k] = a[i][n+k] / a[i][i];
            }
         }
      return(0);   /* solution */
      }
   else {  /* n > 1 */
      if( fabs(a[1][1]) < PRECISION) return(-1); /* division by zero */
      for(j = 1; j <= m; j++) a[1][n+j] = a[1][n+j] / a[1][1];
      return(0);
      }
   }

/*-----------------------------------------------------------------------------*/

/* int invmat(int n, double a[MATRIX_SIZE][MATRIX_SIZE])
      Computes the inverse matrix using Gauss elimination,

         int gauss(n,a)

      n         : Number of equation/unknows
      a[][n+1]  : Matrix to be inverted.

      function returns a -1 if matrix is sigular or division by zero
      else function returns 0.

Note: General program taken from: "An Introduction to Numerical
      Computations", Sidney Yakowits and Ferenc Szidarovszky.
*/
int invmat(int n, double a[MATRIX_SIZE][MATRIX_SIZE]) {
   int gauss(int n, int m, double a[MATRIX_SIZE][2*MATRIX_SIZE]);

   double b[MATRIX_SIZE][2*MATRIX_SIZE];     /* work space matrix */
   register int i;                           /* loop counters */
   register int j;

   for( i = 1; i <= n; i++) {    /* append identity matrix */
      for( j = 1; j <= n; j++) {
         b[i][j] = a[i][j];
         b[i][n+j] = 0.0;
         if( i == j )  b[i][n+j] = 1.0;
         }
      }

   i = gauss(n,n,b);     /* Compute matrix inverse by Gaussian Elimination */
   if( i == -1 ) return(-1);

   for( i = 1; i <= n; i++) {
      for( j = 1; j <= n; j++) a[i][j] = b[i][j+n];
      }
   return(0);
   }
```

Start of **multmat()** function

Calculates ***C***[7][7]=***A***[7][7]****B***[7][7], again from [1] to [7]

End of **multmat()** function

Start of **sim()** function

```
/*---------------------------------------------------------------------------*/

/* void mult_mat( double a[][MATRIX_SIZE], double b[], double c[])
      Created: 3-14-91    Rev: 3-14-91

      Multiplies two matrices:
            c[n] = a[n][n] * b[n]

     MATRIX_SIZE: for this program is 7. The number of ODEs to solve + 1
                  since the zero th position in the arrays are not used Yet.
*/

void mult_mat(double a[MATRIX_SIZE][MATRIX_SIZE], double b[MATRIX_SIZE],
               double c[MATRIX_SIZE]) {
   register int row;  /* loop counter */
   register int col;  /* loop counter */

   for( row = 1; row < MATRIX_SIZE; row++) c[row] = 0.0;  /* zero out vector */

   for( row = 1; row < MATRIX_SIZE; row++ ) {
      for( col = 1; col < MATRIX_SIZE; col++) c[row] += a[row][col] * b[col];
      }
   } /* void mult_mat() */

/*---------------------------------------------------------------------------*/
/*  double sim(double left, double right, int num, FUNC_DATA *fd,
               HULL_SHAPE *hull)

     Simpson's method to integrate the func cross_flow()

         left   : left end point.
         right  : right end point.
         num    : Total number of samples (MUST BE ODD)
                : number of points NOT Intervals, need -.1 for that
         *fd    : struct which cotaines coeff c1,c2,c3 and velocities v1,v2,v3
                  used in function cross_flow().
         *hull  : containes the radius of the hull at a given position x.
*/

double sim(double left, double right, int num, FUNC_DATA *fd, HULL_SHAPE *hull){
   double cross_flow(double sum_x, FUNC_DATA *fd, HULL_SHAPE *hull);

   double ans;         /* THE answer */
   double sum;         /* the sum x values along the hull */
   double step;        /* step size */
   register int i;     /* loop counter */


   ans = cross_flow(left,fd,hull) + cross_flow(right,fd,hull);
   step = (double)( fabs(right - left) / (num - 1) );   /* number INTERVALS */

   /* integrate from negative to positive  X_AP is negative */

   if( left > right) left = right;

   sum = left - step;
   for( i = 1; i < num - 1; i += 2) {     /* sum odd terms  */
      sum += 2.0*step;
      ans += 4.0*cross_flow(sum,fd,hull);
      }

   sum = left;
```

End of **sim**() function; returns the integral value

Start of **romb**() function

```
   for(i = 2; i < num-2; i += 2) {   /* sum even terms */
      sum += 2.0*step;
      ans += 2.0*cross_flow(sum,fd,hull);
      }

   return(((ans*step)/3.0));
   }    /* double sim() */

/*-----------------------------------------------------------------------------*/


/*  double romb(double left, double right, int num, FUNC_DATA *fd,
                HULL_SHAPE *hull)

    Romberg's method to integrate the func cross_flow()

        left   : left end point.
        right  : right end point.
        num    : Total number of iterative steps
        *fd    : struct which cotaines coeff c1,c2,c3 and velocities v1,v2,v3
                 used in function cross_flow().
        *hull  : containes the radius of the hull at a given position x.
*/

double romb(double left, double right, int num, FUNC_DATA *fd, HULL_SHAPE *hull) {
   double cross_flow(double sum_x, FUNC_DATA *fd, HULL_SHAPE *hull);

   double step;  /* step size or h */
   double sum;   /* current value of independent variable */
   double ans;   /* current answer */
   double term[12][12];  /* Romberg integration terms */

   register int k, la, i, j;   /* counters/index variables */

   num += 1;
   k = 1;

   if( left > right ) {        /* To change limits of integration if   */
      step = right;            /*   lower limit (left) is > then upper */
      right = left;            /*   limit right.                       */
      left = step;
      }

   step = right - left;

   term[1][1] = 0.5*step*(cross_flow(left,fd,hull) + cross_flow(right,fd,hull));

   for( i = 2; i <= num; i++) {
      /* compute trapezoidal term */
      k *= 2;
      step *= 0.5;
      sum = left - step;
      ans = 0.0;
      term[1][i] = 0.5 * term[1][i-1];

      for(j = 1; j < k; j +=2) {
         sum += 2.0 * step;
         ans += cross_flow(sum,fd,hull);
         }

      /* Richardson extrapolation */
```

End of **romb**() function; returns the integral value

Start of **adaptive_Asim**() function

```
        term[1][i] += ans * step;
        la = 1;
        for( j = 2; j <= i; j++) {
            la *= 4;
            term[j][i] = (double) (la*term[j-1][i] - term[j-1][i-1]) / (la-1);
            }
        } /* for i */

    return(term[num][num]);
    } /* doub romb() */

/*-------------------------------------------------------------------------------*/
/*  double adaptive_Asim(double left, double right, double tol,FUNC_DATA *fd,
                             HULL_SHAPE *hull)

     To estimate the integral of the cross_flow() using an adaptive Simpson's
        method.

     NOTE: This is a very very simple adaptive method!

          left   : left end point.
          right  : right end point.
          tol    : tolerance
          *fd    : struct which cotaines coeff c1,c2,c3 and velocities v1,v2,v3
                   used in function cross_flow().
          *hull  : containes the radius of the hull at a given position x.
*/

double adaptive_Asim(double left, double right, double tol,FUNC_DATA *fd,
        HULL_SHAPE *hull) {
    double cross_flow(double sum_x, FUNC_DATA *fd, HULL_SHAPE *hull);
    double sim(double left, double right, int num, FUNC_DATA *fd, HULL_SHAPE *hull);

    double step;        /* step size */
    double sum;         /* current est of integral */
    double x;           /* current point */
    double end_limit; /* Most positive limit. Ending limit of integration */
    double error1;      /* error estimate */
    double error2;
    double error;       /* current error between two estimates error1 and error2 */

    register int count1, count2, count;

    /* initialization */

    step = 1.0;    /* for this program this would be 1 foot */
    count1 = 4;    /* 2 and 4 otherwise */
    count2 = 8;
    count = 0;

    if( left > right ) {        /* To change limits of integration if  */
        x = right;              /*  lower limit (left) is > then upper */
        end_limit = left;       /*  limit (right)                      */
        }
    else {                      /* Keep default limits */
        x = left;
        end_limit = right;
        }

    sum = 0.0;
```

End of **adaptive_Asim()** function; returns the integral value

```
while( x < end_limit ) {
    count++;
    error1 = sim(x,x+step,count1,fd,hull);
    error2 = sim(x,x+step,count2,fd,hull);
    error = ((16.0*error2 - error1) / 15.0) - error1;

    /* test if the number of iterations exceeded */
    if( count > MAX_ASIM_ITTER ) return(-1);

    if( fabs(error) < (tol*step) ) {  /* test step size */
        x += step;
        step *= 3.0;
        sum += error2;
        }
    else step *= 0.5;   /* reduce step size by a factor of 1/2 */
    } /* while loop */

x = x - (step / 3.0);
sum = sum - error2;
step = end_limit - x;

sum += sim(x,x+step,count2,fd,hull);

return(sum);
} /* doub adaptive_Asim() */
```

13

SUB_MISC.C

Start of **cross_flow()** function

End of **cross_flow()** function; returns the value of the cross-flow at that point

```
/* John Kloske  9-16-90  Rev: 11-25-91 */
/* Andrew Shein */
/* File: sub_misc.c Version 2.0 */
/* This file contains functions to eveluate the cross flow */
/* integrals and set the mass matrix */

#include <math.h>    /* sin()  sqrt()  atan()  cos() fabs() */
#include <stdio.h>

#include <sub.h>

/* double cross_flow(double sum_x, FUNC_DATA *fd, HULL_SHAPE *hull)
      To solve the following function:

      cross_flow =  Xp * y(x) *f1(x) * [f2(x)^2 + f3(x)^2]^1/2

    where f1,f2,f3 have the follwing format: fi = vi + x*ci

    the values for c1,c2,c3,v1,v2,v3 and xp are in variable fd.

    - sum_x : the distance along the hull in the x-direction.
    - *hull : stations and radius along the hull.

   where,
         Xp : can be the variable x if true, or the constant 1 if false
        y(x): position along the hull   b(x), h(x).
*/

double cross_flow(double sum_x, FUNC_DATA *fd, HULL_SHAPE *hull) {
   double f1;
   double f2;
   double f3;         /* Temp values of function */
   double y;          /* y = y(x) = h(x) or b(x)  */
   double slope;      /* linear model to fit between stations */
   int low;           /* lower position in array x[] for cal */

   sum_x = -(sum_x - hull->x_B);    /* added 1-29-91 JK to convert from hull*/
                                     /* offsets to integrations coord. */
   if( sum_x < 0.0 ) sum_x = 0.0;    /* stations data to sim coord */

   low = (int)(sum_x / hull->inc);  /* lower position in array x[] */
   if(  (low + 1) < NO_STATIONS ) {
      slope = (hull->R[low+1] - hull->R[low]) / hull->inc;
      y =  slope*sum_x - slope*hull->x[low] + hull->R[low]; /* h(x) or b(x) */
      }
   else return(0.0);

   f1 = fd->v1 + (sum_x * fd->c1);      /* f1(x) */
   f2 = fd->v2 + (sum_x * fd->c2);      /* f2(x) */
   f3 = fd->v3 + (sum_x * fd->c3);      /* f3(x) */
   if( fd->xp ) y = (sum_x * y * f1) * sqrt(SQR(f2) + SQR(f3));
         /* should use xp = x, the variable */
   else y = (y * f1) * sqrt(SQR(f2) + SQR(f3));

   return(y);
   }   /* double cross_flow() */

/*-----------------------------------------------------------------------------*/
```

Start of **get_integral()** function

Initialise variables depending on which integral is requested

```
/* double get_integral(STATE *s, HULL_SHAPE *h, int eq_type, int CFD_method)

     To determine the following integral:

        X_AP
       /
       |
       | Xp * y(x)  *f1(x)  * [f2(x)^2 + f3(x)^2]^1/2 dx
       |
       /
        X_B

   where,
          Xp : can be the variable x or the constant 1
         y(x): position along the hull

   the integral is solved using an adaptive simpson's method over
   the length of the sub.

   NOTE: The function `method' will call function cross_flow().

            - method = 1 ==> simpson's (fixed step size), default
            - method = 2 ==> adaptive Simpson's
            - method = 3 ==> Romberg
*/

double get_integral(STATE *s, HULL_SHAPE *h, int eq_type, int CFD_method ) {
                    /* Methods of integration */
   double sim(double left, double right, int num, FUNC_DATA *fd, HULL_SHAPE *hull);
   double adaptive_Asim(double left, double right, double tol,FUNC_DATA *fd,
          HULL_SHAPE *hull);
   double romb(double left, double right, int num, FUNC_DATA *fd, HULL_SHAPE *hull);

   FUNC_DATA fd;           /* Holds coeff c1,c2,c3 and vel v1,v2,v3 */
   double result;          /* Answered returned */
   double lower_limit;     /* left end point ==> x_B station 0 */
   double upper_limit;     /* right end point ==> CG to AP at the stern */
   double tol;             /* tolerance used in adaptive_romb() */
   int num_points;         /* Number of points used to integrated */

   switch(eq_type) {        /*  To set coefficients for eq_type */
      case LATERAL :
         fd.xp = FALSE;
         fd.c1 = s->r;     /* v(x)  = v + xr */
         fd.c2 = -s->q;    /* w(x)  = w - xq */
         fd.c3 = fd.c1;    /* v(x)           */
         fd.v1 = s->v;
         fd.v2 = s->w;
         fd.v3 = fd.v1;
         break;
      case NORMAL :
         fd.xp = FALSE;
         fd.c1 = -s->q;    /* w(x)  = w - xq */
         fd.c2 = fd.c1;    /* w(x)           */
         fd.c3 = s->r;     /* v(x)  = v + xr */
         fd.v1 = s->w;
         fd.v2 = fd.v1;
         fd.v3 = s->v;
         break;
      case PITCH :
         fd.xp = TRUE;     /* xp = x  the variable */
         fd.c1 = -s->q;    /* w(x)  = w - xq */
```

Call apropriate integral solving function

End of **get_integral**() function; returns the integral value

```
         fd.c2 = fd.c1;    /* w(x)            */
         fd.c3 = s->r;     /* v(x) = v + xr */
         fd.v1 = s->w;
         fd.v2 = fd.v1;
         fd.v3 = s->v;
         break;
      case YAW :
         fd.xp = TRUE;     /* xp = x the variable */
         fd.c1 = s->r;     /* v(x) = v + xr */
         fd.c2 = -s->q;    /* w(x) = w - xq */
         fd.c3 = fd.c1;    /* v(x)            */
         fd.v1 = s->v;
         fd.v2 = s->w;
         fd.v3 = fd.v1;
         break;
      default :
         return(-1);      /* passing some type of shit */
      }

   lower_limit = h->x_B;     /* distance from the CG to the bow */
   upper_limit = h->x_AP;    /* distance from the CG to the stern AP */

   switch(CFD_method) {
      case ASIM :                     /* adaptive Simpson's method */
         tol = RELERR;
         result = adaptive_Asim(lower_limit, upper_limit, tol, &fd, h);
         break;
      case ROMB :                     /* Romberg method */
         result = romb(lower_limit,upper_limit,10,&fd,h);
         break;
      case SIMPSON :                  /* Simpson's method */
         num_points = MAX_POINTS;
         result = sim(lower_limit, upper_limit, num_points, &fd, h);
         break;
      default :
         num_points = MAX_POINTS;  /* Simpson's method */
         result = sim(lower_limit, upper_limit, num_points, &fd, h);
      }

   if( result == -1.0 )
         (void)fprintf(stderr, " integration failed  %d \n", (int)result);

   return(result);
   }

/*-----------------------------------------------------------------------------*/

/* void get_const_mat(double const_mat[], STATE *state, HYDRO_COEFF *coeff,
                      HULL_SHAPE *hull, int method)

       To fill the constant matrix B[] using functions for right hand
         side of equations:
            .
         A X  = B,
                     where   A[6x6] : constant coefficient matrix
                             .          .  .  .  .  .  .  T
                             X[6]   : [u, v, w, p, q, r ]

                             B[6]   : constant matrix (vector)

   - method : method of integration.
*/
```

Start of **get_const_mat()** function

(Don't know why it's called this as
a) it's not constant and
b) it's not a matrix.
It actually produces the vector of forces and torques for the 6DOF.)

Fills the const_mat vector with the 6 forces and torques

End of **get_const_mat()** function

Start of **get_mass_mat()** function

Fills the mass matrix with masses, inertias, added masses and added inertias;
note again that this effectively 6x6 matrix runs from [1][1] to [7][7]
and not the normal [0][0] to [6][6] as expected by C

```
void get_const_mat(double const_mat[MATRIX_SIZE], STATE *state, HYDRO_COEFF *coeff,
      HULL_SHAPE *hull, STATUS *stat, int CFD_method) {
   double axial_force_rtside(STATE  *s, AXIAL_COEFF *ac, HULL_SHAPE *h, STATUS *stat);
   double lateral_force_rtside(STATE  *s, LATERAL_COEFF *lc, HULL_SHAPE *h,
         STATUS *stat, int CFD_method);
   double normal_force_rtside(STATE  *s, NORMAL_COEFF *nc, HULL_SHAPE *h, STATUS *stat,
         int CFD_method);
   double rolling_moment_rtside(STATE  *s, ROLL_COEFF *rc, HULL_SHAPE *h, STATUS *stat);
   double pitching_moment_rtside(STATE  *s, PITCH_COEFF *pc, HULL_SHAPE *h,
         STATUS *stat, int CFD_method);
   double yawing_moment_rtside(STATE  *s, YAW_COEFF *yc, HULL_SHAPE *h, STATUS *stat,
         int CFD_method);

   /* GET FORCES AND MOMENTS */
   const_mat[AXIAL] = axial_force_rtside(state, &(coeff->axial), hull, stat);
   const_mat[LATERAL] = lateral_force_rtside(state, &(coeff->lateral), hull, stat,
         CFD_method);
   const_mat[NORMAL] = normal_force_rtside(state, &(coeff->normal), hull, stat,
         CFD_method);
   const_mat[ROLL] = rolling_moment_rtside(state, &(coeff->roll), hull, stat);
   const_mat[PITCH] = pitching_moment_rtside(state, &(coeff->pitch), hull, stat,
         CFD_method);
   const_mat[YAW] = yawing_moment_rtside(state, &(coeff->yaw), hull, stat, CFD_method);
   }   /* void get_const_mat() */

/*----------------------------------------------------------------------------*/

/* void get_mass_mat(double A[MATRIX_SIZE][MATRIX_SIZE], STATE *s,
                        HYDRO_COEFF *c)

     To fill in coefficient matrix A[6x6] for the system:
         .
       A X = B,
                where   A[6x6] : constant coefficient matrix
                          .          .  .  .  .  .  .  T
                          X[6]   : [u, v, w, p, q, r ]

                          B[6]   : constant matrix (vector)
*/

void get_mass_mat(double A[MATRIX_SIZE][MATRIX_SIZE], STATE *s, HYDRO_COEFF *c) {
   double m;   /* mass */
   m = s->mass;

   A[1][1] = (m - c->axial.X_u_dot);             /* row 1 */
   A[1][2] = 0.0;
   A[1][3] = 0.0;
   A[1][4] = 0.0;
   A[1][5] = m*s->Z_G;
   A[1][6] = -m*s->Y_G;

   A[2][1] = 0.0;                                /* row 2 */
   A[2][2] = m - c->lateral.Y_v_dot;
   A[2][3] = 0.0;
   A[2][4] = -(m*s->Z_G + c->lateral.Y_p_dot);
   A[2][5] = 0.0;
   A[2][6] = (m*s->X_G - c->lateral.Y_r_dot);

   A[3][1] = 0.0;                                /* row 3 */
   A[3][2] = 0.0;
   A[3][3] = m - c->normal.Z_w_dot;
   A[3][4] = m*s->Y_G;
```

End of **get_mass_mat()** function

```
A[3][5] = -(m*s->X_G + c->normal.Z_q_dot);
A[3][6] = 0.0;

A[4][1] = 0.0;                                  /* row 4 */
A[4][2] = -(m*s->Z_G + c->roll.K_v_dot);
A[4][3] = m*s->Y_G;
A[4][4] = (s->I_x - c->roll.K_p_dot);
A[4][5] = -s->I_xy;
A[4][6] = -(s->I_zx + c->roll.K_r_dot);

A[5][1] = m*s->Z_G;                             /* row 5 */
A[5][2] = 0.0;
A[5][3] = -(m*s->X_G + c->pitch.M_w_dot);
A[5][4] = -s->I_xy;
A[5][5] = (s->I_y - c->pitch.M_q_dot);
A[5][6] = -s->I_yz;

A[6][1] = -m*s->Y_G;                            /* row 6 */
A[6][2] = (m*s->X_G - c->yaw.N_v_dot);
A[6][3] = 0.0;
A[6][4] = -(s->I_zx + c->yaw.N_p_dot);
A[6][5] = -s->I_yz;
A[6][6] = (s->I_z - c->yaw.N_r_dot);
}
```

14

SUB_RTSI.C

Start of **axial_force_rtside()** function

Calculate X_{uu} as a function of the vehicle's surge speed

Subtract straight-line tether drag from thrust if using that model

```
/* John Kloske  9-16-90    Rev: 11-25-91 */
/* Andy Shein */
// Some additions by Roy Lea
/* File: sub_rtsi.c Version 2.0 */
/* This file contains the functions that decribe the forces and moments
/* experienced by the body. */

#include <math.h>     /* sin()  sqrt()  atan()  cos() */

#include <sub.h>
#include <opt.h>
#include <rov_ext.h>

double axial_force_rtside(STATE *s, AXIAL_COEFF *ac, HULL_SHAPE *h, STATUS *stat) {
   double new_thrust (STATE *s);

   double Aa, Ba, Ca, Da, Ea;  /* temp variables */
   double Fa;  /* Draper stuff only 3-16-91 */
   double sum; /* sum of all values on rigth side of eq (1) ==> const */
   double vr;  /* vel y-axis * ang vel z-axis */
   double q2;  /* q^2 ang vel in y-axis  */
   double r2;  /* r^2 ang vel in z-axis */
   double rp;  /* ang vel z-axis * ang vel x-axis */
   double wq;  /* vel on z-axis * ang vel in y-axis */
   double u2;  /* u^2 vel on x-axis */
   double drag;

   u2 = SQR(s->u);  /* To calculate terms that are used more then once */
   q2 = SQR(s->q);
   r2 = SQR(s->r);
   vr = s->v * s->r;
   wq = s->w * s->q;
   rp = s->r * s->p;

   Aa = ac->X_qq*(q2) + ac->X_rr*(r2) + ac->X_rp*(rp);
   Ba = ac->X_vr*(vr) + ac->X_wq*(wq);
   Ca = ac->X_vv*SQR(s->v) + ac->X_ww*SQR(s->w);

   Da = ac->X_deltar_deltar*(u2)*SQR(s->deltar)
      + ac->X_deltas_deltas*(u2)*SQR(s->deltas)
      + ac->X_deltab_deltab*(u2)*SQR(s->deltab);

      ac->X_uu =0.5*-3.12661+1.44963*cos(s->u)+2.27372*sin(s->u)+0.71716*cos(2*s->u)
         -1.53679*sin(2*s->u)-0.73919*cos(3*s->u)+0.09957*sin(3*s->u)
         +0.13634*cos(4*s->u)+0.15798*sin(4*s->u)+0.00735*cos(5*s->u)
         -0.02748*sin(5*s->u); // X'uu is f(u)
   if(s->u>2.0) ac->X_uu=2.9e-3;
   ac->X_uu *= 0.5*s->density*SQR(h->length); // Generate Xuu from X'uu

   s->F_xp = new_thrust(s);   // Thrust
   drag = ac->X_uu*SQR(s->u); // Drag
#if(SL_TETHER_DRAG)
   drag += 0.5*s->density*t.Cdt*PI*t.diam*t.sl_length*SQR(s->u);
#endif
   Ea = -(s->weight - s->B) * sin(s->theta) + s->F_xp - drag;

   /* Draper only  Fa */

   Fa = ac->X_w_deltas*(s->w)*s->deltas + ac->X_q_deltas*(s->q)*s->deltas;
```

If using the manoeuvring tether model, account for connection point effects

If using bending moments in the tether model, add those on too

End of **axial_force_rtside()** function; return net axial (surge) force

Start of **lateral_force_rtside()** function

If using the manoeuvring tether model, account for connection point effects

If using bending moments in the tether model, add those on too

End of **lateral_force_rtside()** function; return net lateral (sway) force

```
    Fa += ac->X_v_deltar*(s->v)*s->deltar + ac->X_r_deltar*(s->r)*s->deltar;
    Fa += ac->X_deltaa_deltaa*SQR(s->deltaa);

    sum = vr - wq + s->X_G*(q2 + r2) - s->Y_G*(s->q*s->p) - s->Z_G*rp;
    sum = (s->mass * sum) + Aa + Ba + Ca + Da + Ea + Fa;
#if(TETHER_DYNAMICS==TRUE)
    sum -= t.t_i[TETHER_POINTS-1]*cos((PIBY2-t.phi_i[TETHER_POINTS-1])-s->psi);
#if(BENDING==TRUE)
    sum += t.EI*sin((PIBY2-t.phi_i[TETHER_POINTS-1])-s->psi)
            *(-t.phi_i[TETHER_POINTS-4]
            +4*t.phi_i[TETHER_POINTS-3]-5*t.phi_i[TETHER_POINTS-2]
            +2*t.phi_i[TETHER_POINTS-1])/(SQR(t.space_step));
#endif
#endif

    s->X = sum;    /* report total X force */
    return(sum);
    }


double lateral_force_rtside(STATE  *s, LATERAL_COEFF *lc, HULL_SHAPE *h, STATUS *stat,
        int CFD_method) {
    double get_integral(STATE *s, HULL_SHAPE *h, int eq_type, int method);

    double Al, Bl, Cl, Dl, Gl;    /* Temp variables */
    double El;    /* integral over length of body * Cd  */
    double sum;   /* sum of all values on right side of eq (2) ==> constant */
    double u2;    /* u^2 */

    u2 = SQR(s->u);

    Al = lc->Y_p_p*(s->p)*fabs(s->p) + lc->Y_pq*(s->p)*s->q;
    Bl = lc->Y_r*(s->u)*s->r + lc->Y_p*(s->u)*s->p + lc->Y_wp*(s->w)*s->p;
    Cl = lc->Y_star*(u2) + lc->Y_v*(s->u)*s->v + lc->Y_v_v_R*(s->v) * MAG(s->v,s->w);

    Dl = lc->Y_deltar*(u2)*s->deltar +
        lc->Y_deltar_eta*(u2)*(s->deltar)*(s->eta*s->c - 1);

    Gl = (s->weight - s->B)*cos(s->theta)*sin(s->phi);
    El = -s->Cd * get_integral(s, h, LATERAL, CFD_method);

    sum = Al + Bl + Cl + Dl + El + Gl;
    sum += s->mass*( s->w*s->p - s->u*s->r + s->Y_G*(SQR(s->r) + SQR(s->p))
                    - s->Z_G*(s->q)*s->r - s->X_G*(s->q)*s->p );

#if(TETHER_DYNAMICS==TRUE)
    sum -= t.t_i[TETHER_POINTS-1]*sin((PIBY2-t.phi_i[TETHER_POINTS-1])-s->psi);
#if(BENDING==TRUE)
    sum -= t.EI*cos((PIBY2-t.phi_i[TETHER_POINTS-1])-s->psi)
            *(-t.phi_i[TETHER_POINTS-4]
            +4*t.phi_i[TETHER_POINTS-3]-5*t.phi_i[TETHER_POINTS-2]
            +2*t.phi_i[TETHER_POINTS-1])/(SQR(t.space_step));
#endif
#endif
    s->Y = sum;     /* report total Y force */
    s->CFY = El;    /* report Y cross flow contribution */

    return(sum);
    }
```

Start of **normal_force_rtside()** function

End of **normal_force_rtside()** function; return net normal (heave) force

Start of **rolling_moment_rtside()** function

```
double normal_force_rtside(STATE  *s, NORMAL_COEFF *nc, HULL_SHAPE *h, STATUS *stat,
        int CFD_method) {
   double get_integral(STATE *s, HULL_SHAPE *h, int eq_type, int method);

   double An, Bn, Cn, Dn, En, Fn;   /* Temp variables */
   double Gn;   /* Draper stuff only  3-16-91 */
   double sum;  /* sum of all values on right side of eq (3) ==> const */
   double u2;   /* u^2  */


   u2 = SQR(s->u);
   An = nc->Z_q*(s->u)*s->q + nc->Z_vp*(s->v)*s->p;
   Bn = nc->Z_star*u2 + nc->Z_w*(s->u)*s->w;
   Cn = nc->Z_w_*(s->u)*fabs(s->w) + nc->Z_ww*fabs((s->w)*MAG(s->v,s->w));
   Dn = nc->Z_deltas*(u2)*s->deltas + nc->Z_deltab*(u2)*s->deltab
        + nc->Z_deltas_eta*(u2)*(s->deltas)*(s->eta*s->c - 1);

   En = -s->Cd * get_integral(s, h, NORMAL, CFD_method);
   Fn = (s->weight - s->B)*cos(s->theta)*cos(s->phi);

   Gn = nc->Z_pr*(s->p)*s->r;   /* Draper only Gn */

   sum = An + Bn + Cn + Dn + En + Fn + Gn;
   sum += s->mass*( s->u*s->q - s->v*s->p + s->Z_G*(SQR(s->p) + SQR(s->q))
                    - s->X_G*(s->r)*s->p - s->Y_G*(s->r)*s->q );

   s->Z = sum;     /* report total Z force */
   s->CFZ = En;    /* report Z cross flow contribution */

   return(sum);
   }


double rolling_moment_rtside(STATE  *s, ROLL_COEFF *rc, HULL_SHAPE *h, STATUS *stat) {
   double Ar, Br, Cr, Dr, Gr, Er;    /* Temp variables */
   double Fr;    /* Draper stuff only 3-16-91 */
   double sum;   /* sum of all values on right side of eq (4) ==> const */
   double u2;    /* u^2  */
   double Us2;   /* velocity at sternplane x-coord relative to fluid Us^2 */
   double phi_s; /* hydrodyn roll ang at the sternplanes= -atan(w_s/v_s) */

   u2 = SQR(s->u);
   s->w_s = s->w - (h->x_s * s->q);  /* velocity comp in z-dir at the */
                                     /* quarter chord of the stern- */
                                     /* planes w_s = w - x_s*q */

   s->v_s = s->v + (h->x_s * s->r);  /* velocity comp in y-dir at the */
                                     /* quarter chord of the stern- */
                                     /* planes v_s = v + x_s*r */

   if( s->v_s != 0.0 ) phi_s = -atan(s->w_s/s->v_s);
   else phi_s = -PIBY2;   /* atan( inf ) = PIBY2 */

   Ar = rc->K_qr*(s->q)*s->r + rc->K_p_p*s->p*fabs(s->p);
   Br = rc->K_p*(s->u)*s->p + rc->K_r*(s->u)*s->r + rc->K_wp*(s->w)*s->p;
   Cr = rc->K_star*u2 + rc->K_vR*(s->u)*s->v;
   Dr = rc->K_deltar*(u2)*s->deltar
        + rc->K_deltar_eta*(u2)*s->deltar*(s->eta*s->c - 1);

   if( s->u != 0.0 ) {
      Us2 = u2 + SQR(s->v_s) + SQR(s->w_s);
      Er = (u2 + SQR(s->v_s) + SQR(s->w_s)) * SQR( atan( MAG(s->v_s,s->w_s) /s->u ) )
```

End of **rolling_moment_rtside()** function; return net rolling moment (torque)

Start of **pitching_moment_rtside()** function

```
        * (  (rc->K_4S*Us2)*sin((4*phi_s)) + (rc->K_8S*Us2)*sin((8*phi_s)) );
    }
  else Er = PIBY2;

  Gr = (s->Y_G*s->weight - s->Y_B*s->B) * cos(s->theta)*cos(s->phi)
      -(s->Z_G*s->weight - s->Z_B*s->B) * cos(s->theta)*sin(s->phi);
  s->Q_p = s->prop.K2*(s->prop.K3)*SQR(s->RPS*60.0);

  /* draper only Fr */

  Fr = rc->K_wr*(s->w)*s->r + rc->K_deltaa*(u2)*s->deltaa;

  /* add all moments together */

  sum = Ar + Br + Cr + Dr + Er + Fr + Gr + s->Q_p;
  sum += -(s->I_z - s->I_y)*s->q*s->r + s->q*(s->p)*s->I_zx
        - (SQR(s->r) - SQR(s->q))*s->I_yz - s->p*(s->r)*s->I_xy;

  sum += s->mass*( s->Y_G*(s->u)*s->q - s->Y_G*(s->v)*s->p
                 - s->Z_G*(s->w)*s->p + s->Z_G*(s->u)*s->r );

  s->K = sum;    /* report total Rolling moment */

  return(sum);
  }


double pitching_moment_rtside(STATE  *s, PITCH_COEFF *pc, HULL_SHAPE *h, STATUS *stat,
      int CFD_method) {
  double get_integral(STATE *s, HULL_SHAPE *h, int eq_type, int CFD_method);

  double Ap, Bp, Cp, Dp, Ep, Fp, Gp;   /* Temp variables */
  double Hp;   /* Draper stuff only 3-16-91 */
  double sum;  /* sum of all values on right side of eq (5) ==> const */
  double u2;   /* u^2  */

  u2 = SQR(s->u);

  Ap = pc->M_rp*(s->r)*s->p;
  Bp = pc->M_q*(s->u)*s->q;
  Cp = pc->M_star*u2 + pc->M_w*(s->u)*s->w
      + pc->M_w_w_R*(s->w)*MAG(s->v,s->w);

  Dp = pc->M_w_*(s->u)*fabs(s->w) + pc->M_ww*fabs(s->w*MAG(s->v,s->w));
  Ep = pc->M_deltas*(u2)*s->deltas + pc->M_deltab*(u2)*s->deltab
      + pc->M_deltas_eta*(u2)*s->deltas*(s->eta*s->c - 1);

  /* density/2 ?, no Andy 11-12-90 just about 1 anyway */

  Fp = s->Cd * get_integral(s,h,PITCH, CFD_method);

  Gp = -(s->X_G*s->weight - s->X_B*s->B)*cos(s->theta)*cos(s->phi)
     - (s->Z_G*s->weight - s->Z_B*s->B)*sin(s->theta);

  /* For Draper only Hp */

  Hp = pc->M_vp*(s->v)*s->p + pc->M_v_deltaa*(s->v)*s->deltaa;
  Hp += pc->M_r_deltaa*(s->r)*s->deltaa;

  sum = Ap + Bp + Cp + Dp + Ep + Fp + Gp + Hp;
  sum += -(s->I_x - s->I_z)*(s->r)*s->p + s->I_xy*(s->q)*s->r
        - (SQR(s->p) - SQR(s->r))*s->I_zx - s->I_yz*(s->q)*s->p;
```

End of **pitching_moment_rtside()** function; return net pitching moment (torque)

Start of **yawing_moment_rtside()** function

If using the manoeuvring tether model, account for connection point effects

If using bending moments in the tether model, add those on too

End of **yawing_moment_rtside()** function; return net yawing moment (torque)

```
   sum += -s->mass*( (s->Z_G*(s->w*s->q - s->v*s->r))
          + (s->X_G*(s->u*s->q - s->v*s->p)) );

   s->M = sum;     /* report total Pitching moment */
   s->CFM = Fp;    /* report Pitching moment  cross flow contribution */
   return(sum);
   }


double yawing_moment_rtside(STATE  *s, YAW_COEFF *yc, HULL_SHAPE *h, STATUS *stat,
      int CFD_method) {
   double get_integral(STATE *s, HULL_SHAPE *h, int eq_type, int CFD_method);

   double Ay, By, Cy, Dy, Ey, Fy;   /* Temp variables */
   double Gy;   /* draper stuff only 7-14-91 */
   double sum;  /* sum of all values on right side of eq (6) ==> const */
   double u2;   /* u^2  */

   u2 = SQR(s->u);

   Ay = yc->N_pq*(s->p)*s->q;
   By = yc->N_p*(s->u)*s->p +  yc->N_r*(s->u)*s->r;
   Cy = yc->N_star*u2 + yc->N_v*(s->u)*s->v
        + yc->N_v_v_R*(s->v)*MAG(s->v,s->w);
   Dy = yc->N_deltar*(u2)*s->deltar
        + yc->N_deltar_eta*(u2)*s->deltar*(s->eta*s->c - 1);

   Ey = -s->Cd * get_integral(s, h, YAW, CFD_method);

   Fy = (s->X_G*s->weight - s->X_B*s->B)*cos(s->theta)*sin(s->phi) +
        (s->Y_G*s->weight - s->Y_B*s->B)*sin(s->theta);

   /* for draper only Gy */

   Gy = yc->N_w_deltaa*(s->w)*s->deltaa + yc->N_q_deltaa*(s->q)*s->deltaa;
   Gy += yc->N_deltas_deltaa*(s->deltas)*s->deltaa;

   sum = Ay + By + Cy + Dy + Ey + Fy + Gy;
   sum += -(s->I_y - s->I_x)*(s->p)*s->q + s->I_yz*(s->r)*s->p
         - (SQR(s->q) - SQR(s->p))*s->I_xy - s->r*(s->q)*s->I_zx;

   sum += s->mass*( s->X_G*(s->w*s->p - s->u*s->r)
                  - s->Y_G*(s->v*s->r - s->w*s->q) );

#if(TETHER_DYNAMICS==TRUE)
   sum += (0.97*0.5+s->X_G)*t.t_i[TETHER_POINTS-1]
         *sin((PIBY2-t.phi_i[TETHER_POINTS-1])-s->psi);
#if(BENDING==TRUE)
   sum += (0.97*0.5+s->X_G)*t.EI*cos((PIBY2-t.phi_i[TETHER_POINTS-1])-s->psi)
         *(-t.phi_i[TETHER_POINTS-4]
         +4*t.phi_i[TETHER_POINTS-3]-5*t.phi_i[TETHER_POINTS-2]
         +2*t.phi_i[TETHER_POINTS-1])/(SQR(t.space_step));
#endif
#endif
   s->N = sum;     /* report total Yawing moment */
   s->CFN = Ey;    /* report Yawing moment cross flow contribution */

   return(sum);
   }
```

15

SUB_PRIN.C

Start of **print_mass_mat()** function

Prints out the matrix of masses, inertias, added masses and added inertias

End of **print_mass_mat()** function

Start of **print_const_mat()** function

Prints out the vector of forces and moments (torques)

End of **print_const_mat()** function

Start of **print_info()** function

```
/* John Kloske  4/21/91    Rev: 11-25-91 */
/* Andy Shein */
/* Roy Lea 30/5/97 */
/* File: sub_prin.c Version: 3.0  */
/* Functions to print the mass matrix and state info each time step */

#include <stdio.h>  /* fprintf() */

#include <sub.h>
#include <sim.h>
#include <opt.h>
#include <rov_ext.h>

extern FILE *out_file;
extern FILE *out_teth;

// void print_coeff_mat(double A[MATRIX_SIZE][MATRIX_SIZE])
// To print out coefficient matrix

void print_mass_mat(double A[MATRIX_SIZE][MATRIX_SIZE]) {
   register int row;    /* loop counters */
   register int col;

   (void)printf("                                Mass  Matrix\n\n");
   (void)printf
        ("       .          .           .          .           .           .\n");
   (void)printf
      ("        u          v           w          p           q           r\n\n");

   for( row = 1; row < MATRIX_SIZE; row++) {
      (void)printf(" %d ",row);
      for(col = 1; col < MATRIX_SIZE; col++)
            (void)printf(" %-5.3e ", A[row][col]);
      (void)printf("\n\n");
      }
   }   /* void print_coeff_mat() */

/*-------------------------------------------------------------------------------*/

void print_const_mat(double b[MATRIX_SIZE]) {
   // To print constant matrix
   register int i;   /* silly loop counter, once again */

   (void)printf("\n    Axial      Lateral       Normal      Roll");
   (void)printf("       Pitch         Yaw\n");
   for(i = 1; i < MATRIX_SIZE; i++) (void)printf("  %-5.3e ",b[i]);
   (void)printf("\n");
   }


void print_info(STATE *s, SIM_CONTROL *ctrl, STR_VARIABLES *sv, int method,
      long int count, double step) {

   int i;
   double dr;      /* = 180 deg / Pi rad */
   dr = TODEG;

   if( count == 0 ) {
      (void)printf(" Method: ");
```

Prints a header block at the start giving details of the integration method used…

…the commands in the command file…

…and a header line for the simulation output

(Which variables are displayed on screen is controlled by constants set in OPT.H; as they are constants, preprocessor directives are used to compile in the appropriate lines. This avoids a set of run-time if(PRINT_X==TRUE) statements that would result in compiler warnings about expressions always being true or false.)

Variables that are printed to screen (again, determined by OPT.H)

```
    switch(method) {
        case EULER :         (void)printf("Euler          "); break;
        case IMP_EULER :     (void)printf("Imp Euler      "); break;
        default   :             (void)printf("UNKNOWN METHOD ");
        }
    (void)printf("RPS: %4.1f ",(float)(s->RPS));
    (void)printf(" Rudder: %6.2f deg  Plane: %6.2f deg\n",
        (float)(s->deltar*dr), (float)(s->deltas*dr));
    (void)printf(" Time step: %4.3f    Relative Error: %6.5f\n",
          (float)(step), (float)(RELERR));

  // Header line for results
    (void)printf("Time , ");
#if(PRINT_COURSE)
    (void)printf("course, ");
#endif
#if(PRINT_RUDDER)
    (void)printf("  dr   , ");
#endif
#if(PRINT_DEPTH)
    (void)printf("   z   , ");
#endif
#if(PRINT_Z_DOT)
    (void)printf("z_dot, ");
#endif
#if(PRINT_PTCH_D)
    (void)printf("thetad, ");
#endif
#if(PRINT_PITCH)
    (void)printf(" theta, ");
#endif
#if(PRINT_Q)
    (void)printf("  q  , ");
#endif
#if(PRINT_STERNP)
    (void)printf("  ds   , ");
#endif
#if(PRINT_SPEED)
    (void)printf("   u   , ");
#endif
#if(PRINT_U_DOT)
    (void)printf(" u_dot, ");
#endif
#if(PRINT_RPS)
    (void)printf("  RPM   , ");
#endif
#if(PRINT_MTRCMD)
    (void)printf("Mtr_Cmd, ");
#endif
#if(PRINT_STR)
    (void)printf("   a1   ,    b0     ,  k_p   , k_i ,");
#endif
    (void)printf("\n");
    } // End of header section when time=0

  (void)printf("%05.1f, ",(float)(count*step));
#if(PRINT_COURSE)
  (void)printf("%6.2f, ",(float)(s->psi*TODEG));
#endif
#if(PRINT_RUDDER)
  (void)printf("%6.2f, ",(float)(s->deltar*TODEG));
#endif
```

More variables that are printed to screen

All the vehicle variables (and 'sensor' information) is saved to the output file

The position of each tether node...
...and the position of the vehicle
is saved to a separate file

End of **print_info()** function

```
#if(PRINT_DEPTH)
   (void)printf("%6.2f, ",(float)(s->z_o));
#endif
#if(PRINT_Z_DOT)
   (void)printf("%5.2f, ",(float)(s->z_o_dot));
#endif
#if(PRINT_PTCH_D)
   (void)printf("%6.2f, ",(float)(ctrl->pitch*TODEG));
#endif
#if(PRINT_PITCH)
   (void)printf("%6.2f, ",(float)(s->theta*TODEG));
#endif
#if(PRINT_Q)
   (void)printf("%5.2f, ",(float)(s->q));
#endif
#if(PRINT_STERNP)
   (void)printf("%6.2f, ",(float)(s->deltas*TODEG));
#endif
#if(PRINT_SPEED)
   (void)printf("%6.3f, ",(float)(s->u));
#endif
#if(PRINT_U_DOT)
   (void)printf("%6.3f, ",(float)(s->u_dot));
#endif
#if(PRINT_RPS)
   (void)printf("%7.1f, ",(float)(s->RPS*60.0));
#endif
#if(PRINT_MTRCMD)
   (void)printf("%7.0f, ",(float)(ctrl->RPS));
#endif
#if(PRINT_STR)
   (void)printf("%7.4f, %7.4g, %6.2f, %6.3f",(float)(sv->spd.theta[0]),
         (float)(sv->spd.theta[1]),(float)(sv->spd.k_p),(float)(sv->spd.k_i));
#endif
   (void)printf("\n");

   fprintf(out_file,"%-.2f %.3f %.3f %.3f %.3f ",(float)(count*step), s->u,
         s->u_dot, s->v, s->w);
   fprintf(out_file,"%.3f %.3f %.3f ",s->p*TODEG, s->q*TODEG, s->r*TODEG);
   fprintf(out_file,"%.3f %.3f %.3f %.3f ",s->z_o,s->psi*TODEG, s->theta*TODEG,
         s->phi*TODEG);
   fprintf(out_file,"%.3f 0 0 %.1f ",s->u, s->RPS*60.0);
   fprintf(out_file,"%.0f %.2f 0 0 %.2f ",ctrl->RPS, ctrl->deltar*TODEG,
         ctrl->deltas*TODEG);
   fprintf(out_file,"%.2f %.1f %.2f %.2f ",ctrl->speed, ctrl->course*TODEG,
         ctrl->depth, s->u);
   fprintf(out_file,"%.3f %.3f %.3f ",kf.head.s_hat[1], kf.head.s_hat[2]*TODEG,
         kf.head.s_hat[3]*TODEG);
   fprintf(out_file,"%.3f %.3f ",sen.r*TODEG, sen.heading*TODEG);
   fprintf(out_file,"%.3f %.3f ",kf.speed.s_hat[0], kf.speed.s_hat[1]);
   fprintf(out_file,"%.3f %.3f ",sen.u_dot, sen.speed);
   fprintf(out_file,"%.3f %.3f %.3f %.3f ",kf.depth.s_hat[1],
         kf.depth.s_hat[2]*TODEG, kf.depth.s_hat[3]*TODEG, kf.depth.s_hat[4]);
   fprintf(out_file,"%.3f %.3f %.3f\n",sen.q*TODEG, sen.pitch*TODEG, sen.depth);

   for(i=0;i<TETHER_POINTS;i++) fprintf(out_teth,"%.2f ", t.x_i[i]);
   fprintf(out_teth,"%.2f ", s->x_o);
   for(i=0;i<TETHER_POINTS;i++) fprintf(out_teth,"%.2f ", t.y_i[i]);
   fprintf(out_teth,"%.2f\n", s->y_o);
   }  /* void print_info() */
```

16

ROV_CTRL.C

Start of **flight_control()** function

Autopilot type is set in OPT.H, so again use preprocessor directives to compile appropriate functions to call

Calculate sensor readings which may be corrupted by noise...
...as well as kalman filter estimates

Call appropriate controller

End of **flight_control()** function

Start of **sensor_noise()** function

```c
/* Roy Lea  1/7/97 */
/* File: rov_ctrl.c Version: 1.11  */
/* Control and sensor stuff! */

#include <math.h>
#include <stdlib.h>

#include <sub.h>
#include <sim.h>
#include <opt.h>
#include <rov_ext.h>

void flight_control (SIM_CONTROL *ctrl1, STATE *s1, STATUS *stat,
       STR_VARIABLES *str_var) {
   void sensor_noise (STATE *s);
#if(CONTROL_TYPE==FUZZY)
   void fuzzy_flight_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat);
#endif
#if(CONTROL_TYPE==PID)
   void pid_flight_control (SIM_CONTROL *ctrl, STATE *s);
#endif
#if(CONTROL_TYPE==SLIDING_MODE)
   void sliding_mode_flight_control (SIM_CONTROL *ctrl, STATE *s);
#endif
#if(CONTROL_TYPE==FIXED)
   void fixed_flight_control (SIM_CONTROL *ctrl);
#endif
#if(CONTROL_TYPE==STR)
   void str_flight_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat,
          STR_VARIABLES *str_var);
#endif
   void kalman_filter(void);

   sensor_noise(s1);
   kalman_filter();

#if(CONTROL_TYPE==PID)
   pid_flight_control (ctrl1, s1);
#endif
#if(CONTROL_TYPE==FUZZY)
   fuzzy_flight_control (ctrl1, s1, stat);
#endif
#if(CONTROL_TYPE==SLIDING_MODE)
   sliding_mode_flight_control (ctrl1,s1);
#endif
#if(CONTROL_TYPE==FIXED)
   fixed_flight_control (ctrl1);
#endif
#if(CONTROL_TYPE==STR)
   str_flight_control (ctrl1, s1, stat, str_var);
#endif
   }


void sensor_noise(STATE *s) {
   /* s->u etc. are the actual states of the vehicle as determined by the
      simulation.  sen.speed etc. are the inputs to the autopilots and represent
      data from sensors that may be corrupted with noise.  If SENSOR_REAL
      (set in OPT.H) is TRUE, then sensor data is noisy, otherwise it's clean */
```

Speed sensor noise

Surge acceleration sensor reading is differenced speed sensor readings

Depth sensor

Pitch sensor

Pitch rate sensor reading is difference pitch sensor readings

Roll sensor
(although this is not used anywhere)

Heading sensor

Note that heading sensor is more noisy than pitch and roll

End of **sensor_noise()** function

Start of **head_diff()** function

End of **head_diff()** function

```
   double head_diff (double a, double b);
   double integer, fraction;

   sen.u_dot = sen.speed; // sen.speed at this point is from last control time
   sen.speed = s->u;
#if(SENSOR_REAL==TRUE)  // Speed has random noise from -0.1 m/s to +0.1m/s added
   if(fabs(sen.speed)<0.3) sen.speed=0.0;
   else if(fabs(sen.speed)<0.6) sen.speed+=0.2*(rand()%1000)/1000-0.1;
   else sen.speed+=0.1*(rand()%1000)/1000-0.05;
#endif
   sen.u_dot = (sen.speed-sen.u_dot)/0.1;

   sen.depth = s->z_o;
#if(SENSOR_REAL==TRUE)                       // Depth has noise from -1cm to 1cm,
   sen.depth += 0.02*((double)(rand()%1000))/1000.0-0.01;  // quantized to 2.5cm
   fraction = modf(sen.depth/0.025, &integer);
   sen.depth = 0.025*integer;
#endif

   sen.q=sen.pitch;
   sen.pitch = s->theta;
#if(SENSOR_REAL==TRUE)              // Pitch has noise from -0.15 to 0.15 deg,
   sen.pitch *= TODEG;              // resolution of 0.1deg
   sen.pitch += 0.3*((double)(rand()%1000))/1000.0-0.15;
   fraction = modf(sen.pitch/0.1, &integer);
   sen.pitch = 0.1*integer*TORAD;
#endif
   sen.q = (sen.pitch-sen.q)/0.1;

   sen.roll = s->phi;
#if(SENSOR_REAL==TRUE)              // Roll has noise from -0.2 to 0.2 deg,
   sen.roll *= TODEG;               // resolution of 0.1deg
   sen.roll += 0.4*((double)(rand()%1000))/1000.0-0.2;
   fraction = modf(sen.roll/0.1, &integer);
   sen.roll = 0.1*integer*TORAD;
#endif

   sen.r = s->r;
   sen.heading = s->psi;
#if(SENSOR_REAL==TRUE)              // Heading has noise from -2.0 to 2.0 deg,
   sen.heading *= TODEG;            // resolution of 0.1deg
   sen.heading += 4.0*((double)(rand()%1000))/1000.0-2.0;
   fraction = modf(sen.heading/0.1, &integer);
   sen.heading = 0.1*integer*TORAD;
   sen.r += (0.5*((double)(rand()%1000))/1000.0-0.25)*TORAD;
#endif
   }


double head_diff(double a, double b) {
   /* This function subtracts heading 'b' from heading 'a' i.e. a-b.  It then
      corrects for going through 360 deg, i.e. 20-10=10; 5-355=10 as well. */
   double difference;
   difference=a-b;
   if (fabs(difference)>=180.0*TORAD) {
      if (difference<0.0) difference=difference+360.0*TORAD;
      else difference=difference-360.0*TORAD;
      }
   return difference;
   }
```

quantize() function is not used

```
double quantize (variable, limit, bits){
    double output;
    output=((int)(variable*2^bits/limit))*limit/2^bits;
    return output;
    }
```

17

ROV_PID.C

Define the integrator variables that are essentially static in this module

Start of **pid_flight_control()** function

Call the autopilots for the three subsystems

End of **pid_flight_control()** function

Start of **pid_surge_speed_control()** function

PI controller:

$$motor = K_p u_e + K_i \sum_0^k \frac{u_{e_n}}{T}$$

Note motor command limiting and integrator antiwindup

alternative transfer function (z) based controller:

$$motor_k = motor_{k-1} + K\left(u_{e_k} - a.u_{e_{k-1}}\right)$$

End of **pid_surge_speed_control()** function

Start of **pid_course_control()** function

```
/* Roy Lea  15/9/97 */
/* File: rov_pid.c Version: 2.1  */
/* PID control stuff! */

#include <math.h>
#include <stdlib.h>

#include <sub.h>
#include <sim.h>
#include <opt.h>
#include <rov_ext.h>

double speed_integrator=0.0;
double heading_integrator=0.0;
double depth_integrator=0.0;
double pitch_integrator=0.0;
double z_error_kminus1=0.0;

void pid_flight_control (SIM_CONTROL *ctrl, STATE *s) {
   void pid_surge_speed_control (SIM_CONTROL *ctrl, STATE *s);
   void pid_course_control (SIM_CONTROL *ctrl, STATE *s);
   void pid_depth_control (SIM_CONTROL *ctrl, STATE *s);

   pid_surge_speed_control (ctrl, s);
   pid_course_control (ctrl,s);
   pid_depth_control (ctrl,s);
   }


void pid_surge_speed_control (SIM_CONTROL *ctrl, STATE *s) {
   double u_error, n_dot_commanded, gain;
   double K=4000;
   double Ki=1200;
   int i,j;

   u_error = ctrl->speed - sen.speed;

   ctrl->RPS = K*u_error+Ki*speed_integrator;
   if (ctrl->RPS>2100.0) ctrl->RPS=2100.0;
   else if (ctrl->RPS<-2100.0) ctrl->RPS=-2100.0;
   else speed_integrator+=0.1*u_error;
   s->RPS_kminus1 = ctrl->RPS;

// ctrl->RPS=s->RPS_kminus1+4000*(u_error - 0.96*s->u_dash_kminus1);
// if (ctrl->RPS>2100.0) ctrl->RPS=2100.0;
// if (ctrl->RPS<-2100.0) ctrl->RPS=-2100.0;
// s->RPS_kminus1 = ctrl->RPS;
// s->u_dash_kminus1 = u_error;
   }


void pid_course_control (SIM_CONTROL *ctrl, STATE *s) {
   double head_diff(double a, double b);
   double psi_error;
   double K=-0.6;
   double Ki=-0.05;
   double Kd=-0.1;

   psi_error = head_diff(ctrl->course,sen.heading);
```

Full PID controller...

...with rudder command limiting (±20°) and integrator antiwindup

End of **pid_course_control()** function

Start of **pid_depth_control()** function

There are two nested controllers here;
this one produces a commanded pitch angle based on depth error...

(Dive and climb angles limited to ±40°)

...and this one produces a commanded sternplane angle based on pitch error

Sternplane command limited to ±30°, also integrator antiwindup

End of **pid_depth_control()** function

```
    ctrl->deltar = K*psi_error
                   +Ki*heading_integrator
                   +Kd*sen.r;

    if (ctrl->deltar>20*TORAD) ctrl->deltar=20*TORAD;
    else if (ctrl->deltar<-20*TORAD) ctrl->deltar=-20*TORAD;
    else if (fabs(psi_error<10.0*TORAD)) heading_integrator+=0.1*psi_error;

    s->deltar_kminus1 = ctrl->deltar;
    s->psi_dash_kminus1 = psi_error;
    }


void pid_depth_control (SIM_CONTROL *ctrl, STATE *s) {
    double K=-0.8;
    double Ki=-0.05;
    double Kd=-0.3;
    double z_error, theta_demanded, pitch_error;

    z_error = ctrl->depth - sen.depth;
    theta_demanded = -0.5*z_error
                     -0.05*depth_integrator
                     -0.1*(sen.pitch*sen.speed);

    if (theta_demanded>40.0*TORAD) theta_demanded=40.0*TORAD;
    else if (theta_demanded<-40.0*TORAD) theta_demanded=-40.0*TORAD;
    else if (fabs(z_error<1.0)) depth_integrator+=0.1*z_error;

    pitch_error=theta_demanded-sen.pitch;
    ctrl->deltas = K*pitch_error
                   +Ki*pitch_integrator
                   +Kd*(pitch_error-s->z_dash_kminus1)/0.1;

    if (ctrl->deltas>30.0*TORAD) ctrl->deltas=30.0*TORAD;
    else if (ctrl->deltas<-30.0*TORAD) ctrl->deltas=-30.0*TORAD;
    else if (fabs(pitch_error<10.0*TORAD)) pitch_integrator+=0.1*pitch_error;

    z_error_kminus1 = z_error;
    s->deltas_kminus1 = ctrl->deltas;
    s->z_dash_kminus1 = pitch_error;
    }
```

18

ROV_FUZZ.C

Only compile this module if the fuzzy logic autopilot is being used

Start of **fuzzy_init()** function

Sets up the output rules for speed (7x5)...

```
/* Roy Lea  1/7/97 */
/* File: rov_fuzz.c Version: 2.01  */
/* Fuzzy logic control stuff! */

#include <math.h>
#include <stdlib.h>

#include <sub.h>
#include <sim.h>
#include <opt.h>
#include <rov_ext.h>

#if(CONTROL_TYPE==FUZZY)

typedef struct {
   double ss_output_rules[5][7];          /* surge speed output rules */
   double steering_output_rules[5][5];    /* steering output rules */
   double diving_output_rules[5][5];      /* diving output rules */
   double heading_integrator;
   double depth_integrator;
   } FUZZY_CONTROL;

FUZZY_CONTROL f;


void fuzzy_init (void) {
   int i, j;
   double temp, t[5][7];

   f.heading_integrator=0.0;
   f.depth_integrator=0.0;

   /* Sets up the surge speed output rules.  Note that the actual values used
   are a multiple of those listed (see the routine after the definitions).
   These all assume a controller frequency of 10Hz.  */
   t[0][6]=100; t[0][5]=-200; t[0][4]=-300; t[0][3]=-400; t[0][2]=-450; t[0][1]=-500;
t[0][0]=-800;
   t[1][6]=200; t[1][5]=   0; t[1][4]= -30; t[1][3]= -40; t[1][2]= -50; t[1][1]=-200;
t[1][0]=-600;
   t[2][6]=400; t[2][5]= 150; t[2][4]=  25; t[2][3]=   0; t[2][2]= -25; t[2][1]=-150;
t[2][0]=-400;
   t[3][6]=600; t[3][5]= 200; t[3][4]=  50; t[3][3]=  40; t[3][2]=  25; t[3][1]=   0;
t[3][0]=   0;
   t[4][6]=800; t[4][5]= 500; t[4][4]= 450; t[4][3]= 400; t[4][2]= 300; t[4][1]= 200;
t[4][0]=   0;

   for (i=0; i<5; ++i) for (j=0; j<7; ++j) f.ss_output_rules[i][j]=t[i][j];

   /* Sets up the steering output rules. Note that the order of the array is
   different to the one above in that this one follows the layout of the printed
   version, i.e.
                LN SN ZE SP LP psi
               ----------------
      LN       |  |  |  |  |  |
   psi_dot     ----------------
               |  |  |  |  |  |
               ----------------
               |  |  |  |  |  |
               ----------------*/
```

...heading (5x5)...

...and depth (5x5)

End of **fuzzy_init()** function

Start of **fuzzy_flight_control()** function

Call the autopilots for the three subsystems

End of **fuzzy_flight_control()** function

Start of **fuzzy_surge_speed_control()** function

Find the set memberships for the current value of speed error...

...and acceleration

```
    // High gain values will be noisy.
    t[0][0]=-20; t[0][1]=-10; t[0][2]=-5;      t[0][3]= 2.5;    t[0][4]= 20;
    t[1][0]=-20; t[1][1]=-7.5; t[1][2]=-2.5;   t[1][3]=  0;     t[1][4]= 20;
    t[2][0]=-20; t[2][1]= -5;  t[2][2]=  0;    t[2][3]=  5;     t[2][4]= 20;
    t[3][0]=-20; t[3][1]=  0;  t[3][2]=2.5;    t[3][3]= 7.5;    t[3][4]= 20;
    t[4][0]=-20; t[4][1]=2.5;  t[4][2]= 5;     t[4][3]= 10;     t[4][4]= 20;
    for (i=0; i<5; ++i) for (j=0; j<5; ++j) {
       f.steering_output_rules[i][j] = t[i][j]*TORAD;
       }

    t[0][0]=  0; t[0][1]= -5; t[0][2]=-20; t[0][3]=-30;    t[0][4]=-30;
    t[1][0]= 30; t[1][1]=  0; t[1][2]=-10; t[1][3]=-20;    t[1][4]=-30;
    t[2][0]= 30; t[2][1]= 15; t[2][2]=  0; t[2][3]=-15;    t[2][4]=-30;
    t[3][0]= 30; t[3][1]= 20; t[3][2]= 10; t[3][3]=  0;    t[3][4]=-30;
    t[4][0]= 30; t[4][1]= 30; t[4][2]= 20; t[4][3]=  5;    t[4][4]=  0;
    for (i=0; i<5; ++i) for (j=0; j<5; ++j) {
       f.diving_output_rules[i][j] = t[i][j]*TORAD;
       }
    }


void fuzzy_flight_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat) {

    // Fuzzy logic controller.  The output rules are initialised in fuzzy_init.

    void fuzzy_surge_speed_control (SIM_CONTROL *ctrl);
    void fuzzy_course_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat);
    void fuzzy_depth_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat);

    fuzzy_surge_speed_control (ctrl);
    fuzzy_course_control (ctrl, s, stat);
    fuzzy_depth_control (ctrl, s, stat);
    }


void fuzzy_surge_speed_control (SIM_CONTROL *ctrl) {
    /* Fuzzy logic surge speed controller. */
    double fuzzy (double a, double b, double c, double d, double variable);
    double u_error, u_dot, n_dot_commanded, set_total;
    double speed_error_set[7], u_dot_set[5], speed_set[2];
    int i,j,k;

    u_error = ctrl->speed - sen.speed;
    u_dot=sen.u_dot;
    speed_error_set[0] = fuzzy (-100,-100,-0.75,-0.4,u_error);
    speed_error_set[1] = fuzzy (-0.75,-0.4,-0.4,-0.15,u_error);
    speed_error_set[2] = fuzzy (-0.4,-0.15,-0.15,0.0,u_error);
    speed_error_set[3] = fuzzy (-0.15,0.0,0.0,0.15,u_error);
    speed_error_set[4] = fuzzy (0.0,0.15,0.15,0.4,u_error);
    speed_error_set[5] = fuzzy (0.15,0.4,0.4,0.75,u_error);
    speed_error_set[6] = fuzzy (0.4,0.75,100,100,u_error);

    u_dot_set[4] = fuzzy (-10,-10,-0.7,-0.3,u_dot);
    u_dot_set[3] = fuzzy (-0.7,-0.3,-0.2,0.0,u_dot);
    u_dot_set[2] = fuzzy (-0.2,0.0,0.0,0.2,u_dot);
    u_dot_set[1] = fuzzy (0.0,0.2,0.3,0.7,u_dot);
    u_dot_set[0] = fuzzy (0.3,0.7,10,10,u_dot);

    n_dot_commanded = 0.0;
    set_total = 0.0;
```

Calculate the output value based on the membership of each cell

motor command is increased by this value

End of **fuzzy_surge_speed_control()** function

Start of **fuzzy_course_control()** function

Sets are based on heading error and yaw rate

An additional integration term when the heading error is near zero
removes any offsets due to rudder zero position being off

End of **fuzzy_course_control()** function

Start of **fuzzy_depth_control()** function

```
    for (i=0; i<5; ++i) {
        for (j=0; j<7; ++j) {
            n_dot_commanded+=
                    f.ss_output_rules[i][j]*min(speed_error_set[j], u_dot_set[i]);
            set_total+=min(speed_error_set[j],u_dot_set[i]);
            }
        }
    n_dot_commanded = n_dot_commanded / set_total;
    ctrl->RPS = ctrl->RPS+n_dot_commanded;
    if (ctrl->RPS>2100.0) ctrl->RPS=2100.0;
    if (ctrl->RPS<-2100.0) ctrl->RPS=-2100.0;
    }


void fuzzy_course_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat) {
    /* Fuzzy logic course (heading) controller. */
    double head_diff(double a, double b);
    double fuzzy (double a, double b, double c, double d, double variable);

    double psi_tilde, psi_dot, set_total, deltar_commanded;
    double psi_tilde_set[5], psi_dot_set[5];
    double error_near_zero, error_rate_near_zero;
    int i,j;

    psi_tilde = head_diff(sen.heading,ctrl->course);
    psi_dot = sen.r;
    psi_tilde_set[0] = fuzzy (-180.1*TORAD,-180.1*TORAD,-15*TORAD,-7.5*TORAD, psi_tilde);
    psi_tilde_set[1] = fuzzy (-15*TORAD,-7.5*TORAD,-7.5*TORAD,0*TORAD,psi_tilde);
    psi_tilde_set[2] = fuzzy (-7.5*TORAD,0*TORAD,0*TORAD,7.5*TORAD,psi_tilde);
    psi_tilde_set[3] = fuzzy (0*TORAD,7.5*TORAD,7.5*TORAD,15*TORAD,psi_tilde);
    psi_tilde_set[4] = fuzzy (7.5*TORAD,15*TORAD,180.1*TORAD,180.1*TORAD, psi_tilde);

    psi_dot_set[0] = fuzzy (-250*TORAD,-250*TORAD,-20*TORAD,-10*TORAD,psi_dot);
    psi_dot_set[1] = fuzzy (-20*TORAD,-10*TORAD,-10*TORAD,-5*TORAD,psi_dot);
    psi_dot_set[2] = fuzzy (-10*TORAD,-5*TORAD,5*TORAD,10*TORAD,psi_dot);
    psi_dot_set[3] = fuzzy (5*TORAD,10*TORAD,10*TORAD,20*TORAD,psi_dot);
    psi_dot_set[4] = fuzzy (10*TORAD,20*TORAD,250*TORAD,250*TORAD,psi_dot);

    deltar_commanded = 0;
    set_total = 0;

    for (i=0; i<5; ++i) {
        for (j=0; j<5; ++j) {
            deltar_commanded+=f.steering_output_rules[i][j]
                * min(psi_tilde_set[j],psi_dot_set[i]);
            set_total+=min(psi_tilde_set[j],psi_dot_set[i]);
            }
        }
    deltar_commanded = deltar_commanded / set_total;
    ctrl->deltar = deltar_commanded;

    error_near_zero=fuzzy(-10*TORAD,0,0,10*TORAD,psi_tilde);
    error_rate_near_zero=fuzzy(-1*TORAD,0,0,1*TORAD,psi_dot);
    f.heading_integrator+=
            0.1*psi_tilde*min(error_near_zero, error_rate_near_zero);
    ctrl->deltar+=0.25*f.heading_integrator;
    }


void fuzzy_depth_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat) {
    /* Fuzzy logic depth (diving) controller. */
    double fuzzy (double a, double b, double c, double d, double variable);
```

Sets are based on depth error...

...and vehicle pitch

Another integration term when the depth error is near zero removes any offsets due to sternplane misalignment

End of **fuzzy_depth_control()** function

Start of **fuzzy()** function

End of **fuzzy()** function

```
double z_tilde, pitch_commanded, theta_tilde, deltas_commanded, set_total;
double z_tilde_set[5], z_dot_set[5], theta_set[5], q_set[5];
double error_near_zero, error_rate_near_zero;
int i,j,k;

z_tilde = sen.depth - ctrl->depth;

z_tilde_set[0] = fuzzy (-70,-70,-3,-1.5,z_tilde);
z_tilde_set[1] = fuzzy (-3,-1.5,-1,0,z_tilde);
z_tilde_set[2] = fuzzy (-1,0,0,1,z_tilde);
z_tilde_set[3] = fuzzy (0,1,1.5,3,z_tilde);
z_tilde_set[4] = fuzzy (1.5,3,70,70,z_tilde);

z_dot_set[0] = fuzzy (-80*TORAD,-80*TORAD,-40*TORAD,-25*TORAD,sen.pitch);
z_dot_set[1] = fuzzy (-40*TORAD,-25*TORAD,-10*TORAD,0*TORAD,sen.pitch);
z_dot_set[2] = fuzzy (-10*TORAD,0*TORAD,0*TORAD,10*TORAD,sen.pitch);
z_dot_set[3] = fuzzy (0*TORAD,10*TORAD,25*TORAD,40*TORAD,sen.pitch);
z_dot_set[4] = fuzzy (25*TORAD,40*TORAD,80*TORAD,80*TORAD,sen.pitch);

pitch_commanded = 0;
set_total = 0;

for (i=0; i<5; ++i) {
    for (j=0; j<5; ++j) {
            pitch_commanded+=f.diving_output_rules[i][j]
                * min(z_tilde_set[j], z_dot_set[i]);
            set_total+=min(z_tilde_set[j], z_dot_set[i]);
        }
    }
pitch_commanded = pitch_commanded / set_total;
ctrl->deltas=pitch_commanded;

error_near_zero=fuzzy(-1,0,0,1,z_tilde);
error_rate_near_zero=fuzzy(-5*TORAD,0,0,5*TORAD,sen.pitch);
f.depth_integrator+=0.1*z_tilde*min(error_near_zero, error_rate_near_zero);
ctrl->deltas+=-0.05*f.depth_integrator;
}


double fuzzy (double a, double b, double c, double d, double variable) {
    /* This takes the input 'variable' and outputs its membership of the set
    defined by the trapezoid:
             1      /----\
                   /      \
             0 -------------------------
                  a b   c d                       */

    double membership;
    if (variable <= a) membership = 0.0;
        else if (variable >= d) membership = 0.0;
            else if (variable>=b & variable<=c) membership = 1.0;
                else if (variable < b) membership = (variable-a)/(b-a);
                    else membership = (variable-d)/(c-d);
    return membership;
    }
#endif
```

19

ROV_SMC.C

Only compile this module if the sliding mode autopilot is being used

Start of **sliding_mode_flight_control()** function

Call the autopilots for the three subsystems

End of **sliding_mode_flight_control()** function

Start of **sliding_mode_speed_control()** function

Speed error

Sliding surface offset;
note that acceleration is demanded, based on speed error

Predicted drag value

Calculate β for the propeller + vehicle, but limit to a small area of the four quadrants

Calculate required propeller speed, given the predicted drag and using a simplified propeller thrust model (note additional integral term)

Calculate required motor demand, given open-loop model

```
/* Roy Lea  05/08/97 */
/* File: rov_smc.c Version: 1.1  */
/* SMC control stuff!
   Now with integrators! */

#include <math.h>
#include <stdlib.h>

#include <sub.h>
#include <sim.h>
#include <opt.h>
#include <rov_ext.h>

#if(CONTROL_TYPE==SLIDING_MODE)  // Don't compile this file if not using SMC

void sliding_mode_flight_control (SIM_CONTROL *ctrl, STATE *s) {
   void sliding_mode_speed_control (SIM_CONTROL *ctrl, STATE *s);
   void sliding_mode_course_control (SIM_CONTROL *ctrl);
   void sliding_mode_depth_control (SIM_CONTROL *ctrl);

   sliding_mode_speed_control (ctrl, s);
   sliding_mode_course_control (ctrl);
   sliding_mode_depth_control (ctrl);
   }


void sliding_mode_speed_control (SIM_CONTROL *ctrl, STATE *s) {

   int sgn (double variable);
   double sat (double variable);
   double calc_beta (double n, double u, double point_7_pi_d);

   double u_tilde, sigma, eta, sigma_dot, phi;
   double drag, RPM_demanded, beta, temp, u_dot_d;
   int i,j;

   eta=1.0;  phi=0.5;     // Nonlinear gain and boundary layer thickness

// u_tilde = sen.speed - ctrl->speed;
   u_tilde = kf.speed.s_hat[1] - ctrl->speed;
   sigma = u_tilde;
   sigma_dot = eta*sat(sigma/phi);
   u_dot_d=-u_tilde/0.5;

   drag=0.5*sen.speed*sen.speed
      +3.5*sen.speed*sen.speed
      *(ctrl->deltar*ctrl->deltar+ctrl->deltas*ctrl->deltas);

   beta=calc_beta(s->RPS, sen.speed, s->prop.point_7_pi_d);
   if (beta>-10*TORAD && beta<-50*TORAD) beta=beta;
      else if (beta>170*TORAD && beta<230*TORAD) beta=beta-180*TORAD;
         else beta=0.0;

   temp=77400*((-smc_speed_int_term/4.0 +7.1*u_dot_d+drag-7.1*sigma_dot)
         /(3.927*(0.0973-0.13*beta))-sen.speed*sen.speed);
   RPM_demanded =sqrt(fabs(temp))*sgn(temp);

   ctrl->RPS = sgn(RPM_demanded)*2.9e-5*pow(fabs(RPM_demanded),2.4545);
   if (ctrl->RPS>2100.0) ctrl->RPS=2100.0;
```

End of **sliding_mode_speed_control()** function

Start of **sliding_mode_course_control()** function

This uses full state feedback; non-measured states are provided by the Kalman filter

The sliding mode control law;
note the additional integration term to remove stead-state errors due to rudder offsets

End of **sliding_mode_course_control()** function

Start of **sliding_mode_depth_control()** function

This uses full state feedback; non-measured states are provided by the Kalman filter again

This is the control law assuming that the dive and climb angle limits have not been reached;
note the additional integration term to remove stead-state errors due to sternplane offsets

```
    else if (ctrl->RPS<-2100.0) ctrl->RPS=-2100.0;
    else smc_speed_int_term+=u_tilde;
    }


void sliding_mode_course_control (SIM_CONTROL *ctrl) {
    double head_diff (double a, double b);
    int sgn (double variable);
    double sat (double variable);

    double v_tilde, r_tilde, psi_tilde, sigma, eta, phi;
    double v_hat, r_hat, x_hat, x_tilde;
    int i,j,k;

    x_hat=kf.head.s_hat[0];
    v_hat=kf.head.s_hat[1];
    r_hat=kf.head.s_hat[2];

    eta=0.8;
    phi=0.5;                /* Boundary layer thickness */

    psi_tilde = head_diff(sen.heading, ctrl->course);
    x_tilde = x_hat;
    v_tilde = v_hat;
    r_tilde = r_hat-(-psi_tilde*0.4);

    sigma = -0.9492*x_tilde + 0.1776*v_tilde - 0.001*r_tilde + 0.2597*psi_tilde;
    ctrl->deltar =0.1*smc_heading_int_term+3.6065*x_hat -0.2989*v_hat
           +0.2569*r_tilde +0.778*eta*sat(sigma/phi);
    if (ctrl->deltar>20.0*TORAD) ctrl->deltar=20.0*TORAD;
    if (ctrl->deltar<-20.0*TORAD) ctrl->deltar=-20.0*TORAD;
    if((fabs(psi_tilde)<10.0*TORAD)&&(fabs(r_hat)<1.0*TORAD))
           smc_heading_int_term+=0.1*psi_tilde;
    }


void sliding_mode_depth_control (SIM_CONTROL *ctrl) {

    int sgn (double variable);
    double sat (double variable);

    double x_hat, w_hat, q_hat;
    double x_tilde, w_tilde, q_tilde, theta_tilde, z_tilde, sigma, eta, phi;
    double deltas_main, deltas_dive_limit, deltas_climb_limit;

    x_hat=kf.depth.s_hat[0];
    w_hat=kf.depth.s_hat[1];
    q_hat=kf.depth.s_hat[2];

    eta=0.8; phi=0.3;

    z_tilde = sen.depth - ctrl->depth;
    theta_tilde = sen.pitch-0.0;
    q_tilde = q_hat - 0.0;
    w_tilde = w_hat - 0.0;
    x_tilde = x_hat - 0.0;

    sigma = 0.912*x_tilde + 0.203*w_tilde -0.008*q_tilde -0.339*theta_tilde
           + 0.113*z_tilde;
    deltas_main =-0.04*smc_depth_int_term+3.03*x_tilde +0.175*w_tilde
           +0.232*q_tilde +0.122*theta_tilde - 0.873*eta*sat(sigma/phi);
```

If the vehicle is near the dive or climb limit, these control laws take over and limit the vehicle to ±40°

End of **sliding_mode_depth_control**() function

Start of **sgn**() function

End of **sgn**() function

Start of **sat**() function

This returns -1 if x<-1
+1 if x>+1
or otherwise x

End of **sat**() function

```
    if (sen.pitch<-35*TORAD) {
        deltas_dive_limit=3.607*x_tilde-0.299*w_tilde+0.257*q_tilde
                +0.778*eta*sat((-0.949*x_tilde+0.178*w_tilde-0.01*q_tilde
                +0.26*(theta_tilde+40*TORAD))/phi);
        ctrl->deltas=min(deltas_main, deltas_dive_limit);
        }
    else if (sen.pitch>35*TORAD) {
        deltas_climb_limit=3.607*x_tilde-0.299*w_tilde+0.257*q_tilde
                +0.778*eta*sat((-0.949*x_tilde+0.178*w_tilde-0.01*q_tilde
                +0.26*(theta_tilde-40*TORAD))/phi);
        ctrl->deltas=max(deltas_main, deltas_climb_limit);
        }
    else ctrl->deltas=deltas_main;

    if (ctrl->deltas>30.0*TORAD) ctrl->deltas=30.0*TORAD;
    if (ctrl->deltas<-30.0*TORAD) ctrl->deltas=-30.0*TORAD;

    if((fabs(z_tilde)<1.0)&&(fabs(sen.pitch)<5.0*TORAD))
            smc_depth_int_term+=0.1*z_tilde;
    }


int sgn (double variable){
    int output;
    if (variable==0) output= 0;
    if (variable>0)  output= 1;
    if (variable<0)  output=-1;
    return output;
    }


double sat (double variable){
    double output;
    if (abs(variable)<1) output=variable;
        else if (variable<=-1) output=-1;
            else if (variable>=1) output=1;
    return output;
    }


#endif
```

20

SUB_FIX.C

Start of **fixed_flight_control**() function
This allows the motor command, ...
...rudder command...
...and sternplane command
to be specified directly (thruster + actuator dynamics may still apply)

```
/* Roy Lea  9/5/96 */
/* File: sub_fix.c Version: 1.0  */
/* Direct command control stuff! */

#include <sub.h>
#include <sim.h>
#include <opt.h>

void fixed_flight_control (SIM_CONTROL *ctrl) {
   ctrl->RPS = ctrl->speed;
   ctrl->deltar = ctrl->course;
   ctrl->deltas = ctrl->depth;
   }
```

21

ROV_STR.C

Only compile this module if the self-tuning autopilot is being used

Start of **str_flight_control**() function

Note that there is no STR for depth control
End of **str_flight_control**() function

Start of **str_init**() function

Forgetting factor λ set to 0.995

Initial speed PI controller values

```
// Roy Lea  27/8/97
// File: rov_str.c Version: 2.0
// Self-tuning regulator (STR) control!

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include <sub.h>
#include <sim.h>
#include <opt.h>
#include <rov_ext.h>

#if(CONTROL_TYPE==STR) // Don't compile this file if not using STR

static int blah=1;

void str_flight_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat,
      STR_VARIABLES *str_var) {
   void str_surge_speed_control (SIM_CONTROL *ctrl, STR_VARIABLES *str_var);
   void str_course_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat,
         STR_VARIABLES *str_var);
   // void str_depth_control (SIM_CONTROL *ctrl, STATE *s);

   str_surge_speed_control (ctrl, str_var);
   str_course_control (ctrl, s, stat, str_var);
   ctrl->deltas=0.0;
   }


void str_init(STR_VARIABLES *sv) {
   int i,j;

   // Surge speed control variables
   sv->spd.theta[0]=-0.95;          sv->spd.theta[1]=0.01;
   sv->spd.k[0]=0;                  sv->spd.k[1]=0;
   for (i=0; i<2; ++i) for (j=0; j<2; ++j) {
      if (i==j) sv->spd.p[i][j]=1;
         else sv->spd.p[i][j]=0;
      sv->spd.p_kminus1[i][j]=sv->spd.p[i][j];
      }
   sv->spd.epsilon=0;               sv->spd.lambda=0.995;
   sv->spd.time_of_last_command=0;  sv->spd.last_command=0;
   sv->spd.adapt_flag=TRUE;
   sv->spd.k_p=5000;                sv->spd.k_i=300;
   for (i=0; i<2; ++i) {
      sv->spd.theta_kminus1[i]=sv->spd.theta[i];
      sv->spd.phi_kminus1[i]=0;
      }
   sv->speed_integrator=0.0;

   // Heading control variables
   for (i=0; i<12; ++i) {
      sv->head.controller_data[i]=0.0;
      sv->head.theta[i]=0.0;
      sv->head.k[i]=0.0;
      sv->head.theta_kminus1[i]=sv->head.theta[i];
      sv->head.phi_kminus1[i]=0;
      for (j=0; j<12; ++j) {
```

Forgetting factor λ set to 0.995

End of **str_init()** function

Start of **str_surge_speed_control()** function

Set the adaption flag if the vehicle is going faster than 0.5m/s

The RLS to estimate parameteres from vehicle speed and motor command
This uses a model with n=1, m=0

```
        if (i==j) sv->head.p[i][j]=1;
            else sv->head.p[i][j]=0;
        sv->head.p_kminus1[i][j]=sv->head.p[i][j];
        }
    }
sv->head.epsilon=0;                 sv->head.lambda=0.995;
sv->head.time_of_last_command=0;  sv->head.last_command=0;
sv->head.adapt_flag=FALSE;         sv->head.times_round=0;
sv->heading_integrator=0.0;
}


void str_surge_speed_control (SIM_CONTROL *ctrl, STR_VARIABLES *sv) {

    double u_error, temp[3][3], total, a1, b0, b1, r1, s1, c, d;
    double k1, k2;
    int i,j,k;

    // Supervisory logic section
    if(sen.speed>0.5) sv->spd.adapt_flag=TRUE;
    else sv->spd.adapt_flag=FALSE;

    /*      The section below implements the following equations:
                K(t) = P(t-1)rho(t-1)[lambda+rho'(t-1)P(t-1)rho(t-1)]^-1
                P(t) = [I-K(t)rho'(t-1)]P(t-1)/lambda
                epsilon(t) = u(t)-rho'(t-1)theta(t-1)
                theta(t) = theta(t-1)+K(t)epsilon(t)
            where ' is the matrix/vector transpose operator
    */

    // This bit does the bit in brackets: [lambda+rho'(t-1)P(t-1)rho(t-1)]^-1
    total = 0;
    for (i=0; i<2; ++i) for (j=0; j<2; ++j)
        total+=sv->spd.phi_kminus1[i]*(sv->spd.p_kminus1[i][j]
                *sv->spd.phi_kminus1[j]);
    total = 1/(sv->spd.lambda+total);
    // Next bit is K(t) = P(t-1)rho(t-1)*[bit in brackets]
    for (i=0; i<2; ++i) {
        sv->spd.k[i]=0;
        for (j=0; j<2; ++j) sv->spd.k[i]+=sv->spd.p_kminus1[i][j]
                *sv->spd.phi_kminus1[j];
        sv->spd.k[i]*=total;
        }

    // On to P(t) now, this does [I-K(t)rho'(t-1)]
    for (i=0; i<2; ++i) for (j=0; j<2; ++j) {
        if (i==j) temp[i][j]=1;
            else temp[i][j]=0;
        temp[i][j]-=sv->spd.k[i]*sv->spd.phi_kminus1[j];
        }
    // And this is P(t)=[brackets]P(t-1)/lambda
    for (i=0; i<2; ++i) for (j=0; j<2; ++j) {
        sv->spd.p[i][j]=0;
        for (k=0; k<2; ++k) sv->spd.p[i][j]+=temp[i][k]*sv->spd.p_kminus1[k][j];
        sv->spd.p[i][j]/=sv->spd.lambda;
        }

    // epsilon(t)=u(t)-rho'(t-1)theta(t-1)
    sv->spd.epsilon=kf.speed.s_hat[1];
    for (i=0; i<2; ++i)
            sv->spd.epsilon-=sv->spd.phi_kminus1[i]*sv->spd.theta_kminus1[i];
```

The rest of the speed RLS estimator

If the adaption flag is set, then
use pole placement algorithm to produce control law from parameter estimates
(otherwise controller has parameters from when the adaption flag was last set, or initial values)

The actual PI controller

End of **str_surge_speed_control()** function

Start of **str_course_control()** function

Set adaption flag if the vehicle is going faster than 0.8m/s

The RLS parameter estimator;
this is an identical algorithm to the speed estimator which has its own comments

```
    // theta(t) = theta(t-1)+K(t)epsilon(t)
    for (i=0; i<2; ++i)
          sv->spd.theta[i]=sv->spd.theta_kminus1[i]+sv->spd.k[i]*sv->spd.epsilon;

    // Store estimator variables for next time step
    for (i=0; i<2; ++i) {
       sv->spd.theta_kminus1[i]=sv->spd.theta[i];
       for (j=0; j<2; ++j) sv->spd.p_kminus1[i][j]=sv->spd.p[i][j];
       }
    sv->spd.phi_kminus1[1]=ctrl->RPS;
    sv->spd.phi_kminus1[0]=-kf.speed.s_hat[1];

    // Creates coefficients for PI controller,
    // design poles should be 0.9+0.05i etc.

    if(sv->spd.adapt_flag) {
       a1 = sv->spd.theta[0]; b0 = sv->spd.theta[1];
       k1=(-1.8-a1+1)/b0;  k2=(0.81+a1)/b0;
       sv->spd.k_p=-1*k2;  sv->spd.k_i=k1+k2;
       }

    // PI controller
    u_error = ctrl->speed-sen.speed;
    ctrl->RPS = sv->spd.k_p*u_error
              + sv->spd.k_i*sv->speed_integrator;
    if (ctrl->RPS>2100) ctrl->RPS=2100;
    else if (ctrl->RPS<-2100) ctrl->RPS=-2100;
    else sv->speed_integrator+=0.1*u_error;
    }


void str_course_control (SIM_CONTROL *ctrl, STATE *s, STATUS *stat, STR_VARIABLES *sv) {
    double head_diff(double a, double b);
    int inv_big_mat(int n, double a[13][13]);
    double psi_error, temp[12][12], total, a1, b0, a2, c, d;
    double big_matrix[13][13];
    double poles[13], solution[13];
    int i,j,k;

    // Supervisory logic section
    if(sv->head.last_command!=ctrl->course) {
       sv->head.last_command=ctrl->course;
       sv->head.time_of_last_command=stat->sim_time;
       }
    if(sen.speed>0.8) sv->head.adapt_flag=TRUE;
    else sv->head.adapt_flag=FALSE;

    total = 0;
    for (i=0; i<12; ++i) for (j=0; j<12; ++j)
       total+=sv->head.phi_kminus1[i]*(sv->head.p_kminus1[i][j]
              *sv->head.phi_kminus1[j]);
    total = 1/(sv->head.lambda+total);
    for (i=0; i<12; ++i) {
       sv->head.k[i]=0;
       for (j=0; j<12; ++j)
              sv->head.k[i]+=sv->head.p_kminus1[i][j]*sv->head.phi_kminus1[j];
       sv->head.k[i]*=total;
       }
    for (i=0; i<12; ++i) for (j=0; j<12; ++j) {
       if (i==j) temp[i][j]=1;
       else temp[i][j]=0;
       temp[i][j]-=sv->head.k[i]*sv->head.phi_kminus1[j];
```

The rest of the RLS heading estimator

Note that the parameter model here is n=*6*, m=*5* so has 12 parameters in total

If the adaption flag is set, then do pole placement;
algorithm here involves inverting the matrix of parameters:
the Diophantine matrix

```
    }
for (i=0; i<12; ++i) for (j=0; j<12; ++j) {
    sv->head.p[i][j]=0;
    for (k=0; k<12; ++k)
            sv->head.p[i][j]+=temp[i][k]*sv->head.p_kminus1[k][j];
    sv->head.p[i][j]/=sv->head.lambda;
    }
if (s->psi>350*TORAD && sv->head.phi_kminus1[1]<10*TORAD)
        sv->head.times_round-=1;
if (s->psi<10*TORAD && sv->head.phi_kminus1[1]>350*TORAD)
        sv->head.times_round+=1;
sv->head.epsilon=sen.heading+sv->head.times_round*360*TORAD;
for (i=0; i<12; ++i)
        sv->head.epsilon-=sv->head.phi_kminus1[i]*sv->head.theta_kminus1[i];
for (i=0; i<12; ++i)
        sv->head.theta[i]=
                sv->head.theta_kminus1[i]+sv->head.k[i]*sv->head.epsilon;
for (i=0; i<12; ++i) {
    sv->head.theta_kminus1[i]=sv->head.theta[i];
    for (j=0; j<12; ++j) sv->head.p_kminus1[i][j]=sv->head.p[i][j];
    }

for(i=11; i>0; i--) sv->head.phi_kminus1[i]=sv->head.phi_kminus1[i-1];
sv->head.phi_kminus1[6]=ctrl->deltar;
sv->head.phi_kminus1[0]=-sen.heading;

// Pole placement algorithm
if(sv->head.adapt_flag) {
    for(i=1; i<=12; i++) for(j=1; j<=6; j++) {
        if(i==j) {
            big_matrix[i][j]=1.0;
            big_matrix[i][j+6]=0.0;
            }
        else if((i-j)<0) {
            big_matrix[i][j]=0.0;
            big_matrix[i][j+6]=0.0;
            }
        else if((i-j)<7) {
            big_matrix[i][j]=sv->head.theta[i-j-1];
            big_matrix[i][j+6]=sv->head.theta[i-j+5];
            }
        else {
            big_matrix[i][j]=0.0;
            big_matrix[i][j+6]=0.0;
            }
        }

    if(inv_big_mat(12,big_matrix)==ERROR) {
        printf("Error in inverting Diophantine matrix\n");
        exit(ERROR);
        }
    poles[1]=1.0; poles[2]=-1.8; poles[3]=0.81;
    for (i=4; i<=12; i++) poles[i]=0.0;
    for(i=1;i<=12;i++) {
        solution[i]=0.0;
        for (j=1;j<=12;j++) solution[i]+=big_matrix[i][j]*poles[j];
        }

    psi_error = head_diff(ctrl->course,sen.heading);
    for(i=11; i>0; i--)
            sv->head.controller_data[i]=sv->head.controller_data[i-1];
    sv->head.controller_data[0]=-ctrl->deltar;
```

This gives a controller of the form

$$\frac{\delta r_c}{\psi_e} = \frac{s_0 + s_1 z^{-1} + \cdots + s_5 z^{-5}}{1 + r_1 z^{-1} + \cdots + r_5 z^{-5}}$$

i.e. $\delta r_k = s_0 \psi_{e_k} + s_1 \psi_{e_{k-1}} + \cdots + s_5 \psi_{e_{k-5}} - \left(r_1 \delta r_{k-1} + r_2 \delta r_{k-2} + \cdots + r_5 \delta r_{k-5}\right)$

If the adaption flag isn't set, then have random commands to give the RLS data

End of **str_course_control**() function

```
      sv->head.controller_data[6]=psi_error;
      /* controller_data[0]=-deltar(k-1)       controller_data[6] =e(k)
         controller_data[1]=-deltar(k-2)       controller_data[7] =e(k-1)
         controller_data[2]=-deltar(k-3)       controller_data[8] =e(k-2)
         controller_data[3]=-deltar(k-4)       controller_data[9] =e(k-3)
         controller_data[4]=-deltar(k-5)       controller_data[10]=e(k-4)
         controller_data[5]=-deltar(k-6)       controller_data[11]=e(k-5) */

      ctrl->deltar=0.0;
      for(i=0;i<12;i++) ctrl->deltar+=solution[i+1]*sv->head.controller_data[i];
      if (ctrl->deltar>20*TORAD) ctrl->deltar=20*TORAD;
      else if (ctrl->deltar<-20*TORAD) ctrl->deltar=-20*TORAD;
      }
   else ctrl->deltar=TORAD*((rand()%100)/1.5-30.0);
   }
#endif
```

22

KALMAN.C

Start of **kalman_init()** function

Sets the parameters for the speed filter

Heading filter parameters;
note that this uses the system + time delay model

$$\begin{bmatrix} \dot{x} \\ \dot{v} \\ \dot{r} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} -2.86 & 0 & 0 & 0 \\ 1.82\times 5.714 & -2.12 & 0.35 & 0 \\ -13.43\times 5.714 & 8.02 & -8.62 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \\ r \\ \psi \end{bmatrix} + \begin{bmatrix} 1 \\ -1.82 \\ 13.43 \\ 0 \end{bmatrix} \delta r_d$$

where x is the dummy variable to do with the time delay

```
// Author:    R.K.Lea
// Date:      3 July 1997
// File:      KALMAN.C
// Notes:     Kalman filter for speed, heading and depth

#include <math.h>

#include <sub.h>
#include <rov_ext.h>

void kalman_init (STATE *s) {
   // h is the system behaviour matrix
   // f is the sensor coupling matrix
   // delta is the sensor noise covariance matrix (the larger the number,
   //     the more noisy)
   // q is the system noise covariance matrix (???)
   double h[5][5], f[5][5], delta[5][5], q[5][5];
   int i, j;

   kf.speed.h[0][0]=1.0;       kf.speed.h[0][1]=0.0;
   kf.speed.h[1][0]=0.1;       kf.speed.h[1][1]=1.0;

   kf.speed.f[0][0]=1.0;       kf.speed.f[0][1]=0.0;
   kf.speed.f[1][0]=0.0;       kf.speed.f[1][1]=1.0;

   kf.speed.delta[0][0]=2.0;  kf.speed.delta[0][1]=0.0;
   kf.speed.delta[1][0]=0.0;  kf.speed.delta[1][1]=0.1;

   kf.speed.q[0][0]=0.01;      kf.speed.q[0][1]=0.01;
   kf.speed.q[1][0]=0.01;      kf.speed.q[1][1]=0.01;

   kf.speed.s_hat[0]=0.0;      kf.speed.s_hat[1]=0.0;

   for (i=0;i<2;i++) {
      for (j=0;j<2;j++) kf.speed.p[i][j]=kf.speed.delta[i][j];
      }

   kf.head.h[0][0]=1.0-0.286; kf.head.h[0][1]=0.0;
         kf.head.h[0][2]=0.0;           kf.head.h[0][3]=0.0;
   kf.head.h[1][0]=1.04;       kf.head.h[1][1]=1.0-0.212;
         kf.head.h[1][2]=0.035;     kf.head.h[1][3]=0.0;
   kf.head.h[2][0]=-7.674;     kf.head.h[2][1]=0.802;
         kf.head.h[2][2]=1.0-0.862; kf.head.h[2][3]=0.0;
   kf.head.h[3][0]=0.0;           kf.head.h[3][1]=0.0;
         kf.head.h[3][2]=0.1;           kf.head.h[3][3]=1.0;

   kf.head.f[0][0]=0.0; kf.head.f[0][1]=0.0; kf.head.f[0][2]=0.0; kf.head.f[0][3]=0.0;
   kf.head.f[1][0]=0.0; kf.head.f[1][1]=0.0; kf.head.f[1][2]=0.0; kf.head.f[1][3]=0.0;
   kf.head.f[2][0]=0.0; kf.head.f[2][1]=0.0; kf.head.f[2][2]=1.0; kf.head.f[2][3]=0.0;
   kf.head.f[3][0]=0.0; kf.head.f[3][1]=0.0; kf.head.f[3][2]=0.0; kf.head.f[3][3]=1.0;

   kf.head.delta[0][0]=0.1; kf.head.delta[0][1]=0.0; kf.head.delta[0][2]=0.0;
         kf.head.delta[0][3]=0.0;
   kf.head.delta[1][0]=0.0; kf.head.delta[1][1]=0.1; kf.head.delta[1][2]=0.0;
         kf.head.delta[1][3 ]=0.0;
   kf.head.delta[2][0]=0.0; kf.head.delta[2][1]=0.0; kf.head.delta[2][2]=0.5;
         kf.head.delta[2][3]=0.0;
   kf.head.delta[3][0]=0.0; kf.head.delta[3][1]=0.0; kf.head.delta[3][2]=0.0;
         kf.head.delta[3][3]=0.01;
```

More heading parameteres

Depth filter parameteres;
again this uses a system + time delay model

$$\begin{bmatrix} \dot{x} \\ \dot{w} \\ \dot{q} \\ \dot{\theta} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} -2.857 & 0 & 0 & 0 & 0 \\ -1.156\times 5.714 & -1.749 & 0.085 & 0.016 & 0 \\ -10.401\times 5.714 & -5.246 & -7.017 & -0.584 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1.3 & 0 \end{bmatrix} \begin{bmatrix} x \\ w \\ q \\ \theta \\ z \end{bmatrix} + \begin{bmatrix} 1 \\ 1.156 \\ 10.401 \\ 0 \\ 0 \end{bmatrix} \delta s_d$$

(again, x is the dummy time delay variable)

```
kf.head.q[0][0]=0.2; kf.head.q[0][1]=0.1; kf.head.q[0][2]=0.1;
      kf.head.q[0][3]=0.1;
kf.head.q[1][0]=0.1; kf.head.q[1][1]=0.1; kf.head.q[1][2]=0.1;
      kf.head.q[1][3]=0.1;
kf.head.q[2][0]=0.1; kf.head.q[2][1]=0.1; kf.head.q[2][2]=0.1;
      kf.head.q[2][3]=0.1;
kf.head.q[3][0]=0.1; kf.head.q[3][1]=0.1; kf.head.q[3][2]=0.1;
      kf.head.q[3][3]=0.1;

kf.head.s_hat[0]=0.0; kf.head.s_hat[1]=0.0; kf.head.s_hat[2]=0.0;
      kf.head.s_hat[3]=s->psi;

for (i=0;i<4;i++) {
   for (j=0;j<4;j++) kf.head.p[i][j]=kf.head.delta[i][j];
   }

h[0][0]=1.0-0.2857; h[0][1]=0.0;        h[0][2]=0.0;
h[0][3]=0.0;         h[0][4]=0.0;
h[1][0]=-0.6605;     h[1][1]=1.0-0.1749; h[1][2]=-0.0085;
h[1][3]=0.0016;  h[1][4]=0.0;
h[2][0]=-5.943;      h[2][1]=-0.5246;    h[2][2]=1.0-0.7107;
h[2][3]=-0.0584; h[2][4]=0.0;
h[3][0]=0.0;         h[3][1]=0.0;           h[3][2]=0.1;
   h[3][3]=1.0;      h[3][4]=0.0;
h[4][0]=0.0;         h[4][1]=0.1;           h[4][2]=0.0;
   h[4][3]=-0.13;       h[4][4]=1.0;

f[0][0]=0.0;  f[0][1]=0.0;  f[0][2]=0.0;  f[0][3]=0.0;  f[0][4]=0.0;
f[1][0]=0.0;  f[1][1]=0.0;  f[1][2]=0.0;  f[1][3]=0.0;  f[1][4]=0.0;
f[2][0]=0.0;  f[2][1]=0.0;  f[2][2]=1.0;  f[2][3]=0.0;  f[2][4]=0.0;
f[3][0]=0.0;  f[3][1]=0.0;  f[3][2]=0.0;  f[3][3]=1.0;  f[3][4]=0.0;
f[4][0]=0.0;  f[4][1]=0.0;  f[4][2]=0.0;  f[4][3]=0.0;  f[4][4]=1.0;

delta[0][0]=0.1;  delta[0][1]=0.0;  delta[0][2]=0.0;  delta[0][3]=0.0;
   delta[0][4]=0.0;
delta[1][0]=0.0;  delta[1][1]=0.1;  delta[1][2]=0.0;  delta[1][3]=0.0;
   delta[1][4]=0.0;
delta[2][0]=0.0;    delta[2][1]=0.0;  delta[2][2]=0.5;  delta[2][3]=0.0;
   delta[2][4]=0.0;
delta[3][0]=0.0;  delta[3][1]=0.0;  delta[3][2]=0.0;  delta[3][3]=0.1;
   delta[3][4]=0.0;
delta[4][0]=0.0;  delta[4][1]=0.0;  delta[4][2]=0.0;  delta[4][3]=0.0;
   delta[4][4]=0.1;

q[0][0]=0.01;  q[0][1]=0.01;  q[0][2]=0.01;  q[0][3]=0.01;  q[0][4]=0.01;
q[1][0]=0.01;  q[1][1]=0.01;  q[1][2]=0.01;  q[1][3]=0.01;  q[1][4]=0.01;
q[2][0]=0.01;  q[2][1]=0.01;  q[2][2]=0.01;  q[2][3]=0.01;  q[2][4]=0.01;
q[3][0]=0.01;  q[3][1]=0.01;  q[3][2]=0.01;  q[3][3]=0.01;  q[3][4]=0.01;
q[4][0]=0.01;  q[4][1]=0.01;  q[4][2]=0.01;  q[4][3]=0.01;  q[4][4]=0.01;

kf.depth.s_hat[0]=0.0;        kf.depth.s_hat[1]=0.0;      kf.depth.s_hat[2]=0.0;
kf.depth.s_hat[3]=s->theta;   kf.depth.s_hat[4]=s->z_o;

for (i=0;i<5;i++) {
   for (j=0;j<5;j++) {
      kf.depth.h[i][j]=h[i][j];
      kf.depth.f[i][j]=f[i][j];
      kf.depth.delta[i][j]=delta[i][j];
      kf.depth.q[i][j]=q[i][j];
      kf.depth.p[i][j]=kf.depth.delta[i][j];
      }
```

End of **kalman_init()** function

Start of **kalman_filter()** function

End of **kalman_filter()** function

Start of **kalman_speed()** function

Filter inputs

```
        }
    }


void kalman_filter(void){
    void kalman_speed(void);
    void kalman_heading(void);
    void kalman_depth(void);

    kalman_speed();
    kalman_heading();
    kalman_depth();
    }


void kalman_speed(void){
    // The states estimated are u_dot & u ([0], [1] respectively)
    // Inputs are estimated speed rate and speed from sensor
    void mult_2_mat (double a[2][2], double b[2][2], double c[2][2]);
    int inv_2_mat(double matrix_to_invert[2][2]);

    double temp_mat[2][2], temp_mat1[2][2], temp_mat2[2][2];
    double s_hat_k[2], p_k[2][2], k_k[2][2], h_t[2][2], f_t[2][2];
    double x_k[2];
    int i,j,k;

    for(i=0;i<2;i++) {
        for(j=0;j<2;j++) {
            h_t[i][j]=kf.speed.h[j][i];
            f_t[i][j]=kf.speed.f[j][i];
            }
        }
    x_k[0]=sen.u_dot; x_k[1]=sen.speed;

    for(i=0;i<2;i++) {
        s_hat_k[i]=0.0;
        for(j=0;j<2;j++) s_hat_k[i]+=kf.speed.h[i][j]*kf.speed.s_hat[j];
        }

    mult_2_mat(kf.speed.h,kf.speed.p,temp_mat); mult_2_mat(temp_mat,h_t,p_k);
    for(i=0;i<2;i++) {
        for(j=0;j<2;j++) p_k[i][j]+=kf.speed.q[i][j];
        }

    mult_2_mat(kf.speed.f,p_k,temp_mat); mult_2_mat(temp_mat,f_t,temp_mat1);
    for(i=0;i<2;i++) {
        for(j=0;j<2;j++) temp_mat1[i][j]+=kf.speed.delta[i][j];
        }
    i=inv_2_mat(temp_mat1);
    if(i!=-1) {  // if i=-1 then the matrix was not inverted
        mult_2_mat(p_k,f_t,temp_mat); mult_2_mat(temp_mat,temp_mat1,k_k);

        for(i=0;i<2;i++) {
            temp_mat[i][0]=x_k[i];
            for(j=0;j<2;j++) temp_mat[i][0]-=kf.speed.f[i][j]*s_hat_k[j];
            }
        for(i=0;i<2;i++) {
            kf.speed.s_hat[i]=s_hat_k[i];
            for(j=0;j<2;j++) kf.speed.s_hat[i]+=k_k[i][j]*temp_mat[j][0];
            }

        mult_2_mat(k_k,kf.speed.f,temp_mat);
```

End of **kalman_speed()** function

Start of **kalman_heading()** function

Filter inputs

```
        for(i=0;i<2;i++) {
            for(j=0;j<2;j++) {
                if(i==j) temp_mat1[i][j]=1.0;
                else temp_mat1[i][j]=0.0;
                temp_mat1[i][j]-=temp_mat[i][j];
                }
            }
        mult_2_mat(temp_mat1,p_k,kf.speed.p);
        }
    }


void kalman_heading(void){
    // The states estimated are x,v,r & psi ([0], [1], [2] & [3] respectively)
    // Inputs are estimated heading rate and heading from compass
    void mult_4_mat (double a[4][4], double b[4][4], double c[4][4]);
    void inv_4_mat(double matrix_to_invert[4][4]);

    double temp_mat[4][4], temp_mat1[4][4], temp_mat2[4][4];
    double s_hat_k[4], p_k[4][4], k_k[4][4], h_t[4][4], f_t[4][4];
    double x_k[4];
    int i,j,k;

    for(i=0;i<4;i++) {
        for(j=0;j<4;j++) {
            h_t[i][j]=kf.head.h[j][i];
            f_t[i][j]=kf.head.f[j][i];
            }
        }
    x_k[0]=0.0;   x_k[1]=0.0; x_k[2]=sen.r; x_k[3]=sen.heading;

    for(i=0;i<4;i++) {
        s_hat_k[i]=0.0;
        for(j=0;j<4;j++) s_hat_k[i]+=kf.head.h[i][j]*kf.head.s_hat[j];
        }

    mult_4_mat(kf.head.h,kf.head.p,temp_mat); mult_4_mat(temp_mat,h_t,p_k);
    for(i=0;i<4;i++) {
        for(j=0;j<4;j++) p_k[i][j]+=kf.head.q[i][j];
        }

    mult_4_mat(kf.head.f,p_k,temp_mat); mult_4_mat(temp_mat,f_t,temp_mat1);
    for(i=0;i<4;i++) {
        for(j=0;j<4;j++) temp_mat1[i][j]+=kf.head.delta[i][j];
        }
    inv_4_mat(temp_mat1);
    mult_4_mat(p_k,f_t,temp_mat); mult_4_mat(temp_mat,temp_mat1,k_k);

    for(i=0;i<4;i++) {
        temp_mat[i][0]=x_k[i];
        for(j=0;j<4;j++) temp_mat[i][0]-=kf.head.f[i][j]*s_hat_k[j];
        }
    for(i=0;i<4;i++) {
        kf.head.s_hat[i]=s_hat_k[i];
        for(j=0;j<4;j++) kf.head.s_hat[i]+=k_k[i][j]*temp_mat[j][0];
        }

    mult_4_mat(k_k,kf.head.f,temp_mat);
    for(i=0;i<4;i++) {
        for(j=0;j<4;j++) {
            if(i==j) temp_mat1[i][j]=1.0;
            else temp_mat1[i][j]=0.0;
```

End of **kalman_heading()** function

Start of **kalman_depth()** function

Filter inputs

```
          temp_mat1[i][j]-=temp_mat[i][j];
          }
      }
   mult_4_mat(temp_mat1,p_k,kf.head.p);
   }


void kalman_depth(void){
   // The states estimated are x, w, q, theta & z ([0], [1], [2], [3] & [4]
   //     respectively)
   // Inputs are estimated pitch rate, pitch and depth
   void mult_5_mat (double a[5][5], double b[5][5], double c[5][5]);
   int inv_5_mat(double matrix_to_invert[5][5]);

   double temp_mat[5][5], temp_mat1[5][5], temp_mat2[5][5];
   double s_hat_k[5], p_k[5][5], k_k[5][5], h_t[5][5], f_t[5][5];
   double x_k[5];
   int i,j,k;

   for(i=0;i<5;i++) {
      for(j=0;j<5;j++) {
         h_t[i][j]=kf.depth.h[j][i];
         f_t[i][j]=kf.depth.f[j][i];
         }
      }
   x_k[0]=0.0; x_k[1]=0.0; x_k[2]=sen.q; x_k[3]=sen.pitch; x_k[4]=sen.depth;

   for(i=0;i<5;i++) {
      s_hat_k[i]=0.0;
      for(j=0;j<5;j++) s_hat_k[i]+=kf.depth.h[i][j]*kf.depth.s_hat[j];
      }

   mult_5_mat(kf.depth.h,kf.depth.p,temp_mat); mult_5_mat(temp_mat,h_t,p_k);
   for(i=0;i<5;i++) {
      for(j=0;j<5;j++) p_k[i][j]+=kf.depth.q[i][j];
      }

   mult_5_mat(kf.depth.f,p_k,temp_mat); mult_5_mat(temp_mat,f_t,temp_mat1);
   for(i=0;i<5;i++) {
      for(j=0;j<5;j++) temp_mat1[i][j]+=kf.depth.delta[i][j];
      }
   i=inv_5_mat(temp_mat1);
   if(i!=-1) {  // if i=-1 then the matrix was not inverted
      mult_5_mat(p_k,f_t,temp_mat); mult_5_mat(temp_mat,temp_mat1,k_k);

      for(i=0;i<5;i++) {
         temp_mat[i][0]=x_k[i];
         for(j=0;j<5;j++) temp_mat[i][0]-=kf.depth.f[i][j]*s_hat_k[j];
         }
      for(i=0;i<5;i++) {
         kf.depth.s_hat[i]=s_hat_k[i];
         for(j=0;j<5;j++) kf.depth.s_hat[i]+=k_k[i][j]*temp_mat[j][0];
         }

      mult_5_mat(k_k,kf.depth.f,temp_mat);
      for(i=0;i<5;i++) {
         for(j=0;j<5;j++) {
            if(i==j) temp_mat1[i][j]=1.0;
            else temp_mat1[i][j]=0.0;
            temp_mat1[i][j]-=temp_mat[i][j];
            }
         }
```

End of **kalman_depth**() function

Start of **mult_3_mat**() function

Note: the range of matrix multiplications and inversion routines that follow are due to run-time stack and memory errors experienced with the program; these represent work arounds. Ideally, the routines in NEW_MATH.C should be used.

End of **mult_3_mat**() function

Start of **inv_3_mat**() function

End of **inv_3_mat**() function

Start of **mult_2_mat**() function

End of **mult_2_mat**() function

Start of **inv_2_mat**() function

End of **inv_2_mat**() function

```
        mult_5_mat(temp_mat1,p_k,kf.depth.p);
        }
    }


void mult_3_mat(double a[3][3], double b[3][3], double c[3][3]) {
// Multiplies 3x3 matrices 'a' and 'b' to give 'c'
    int i,j,k;

    for(i=0;i<3;i++) {
        for(j=0;j<3;j++) {
            c[i][j]=0.0;
            for(k=0;k<3;k++) c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }

int inv_3_mat(double matrix_to_invert[3][3]) {
// Inverts a 3x3 matrix, returns -1 if not successful
    int invmat(int n, double a[MATRIX_SIZE][MATRIX_SIZE]);
    int i,j, return_value;
    double matrix_to_pass[MATRIX_SIZE][MATRIX_SIZE];

    for(i=0;i<3;i++) {
        for(j=0;j<3;j++) matrix_to_pass[i+1][j+1]=matrix_to_invert[i][j];
        }
    return_value=invmat(3, matrix_to_pass);
    for(i=0;i<3;i++) {
        for(j=0;j<3;j++) matrix_to_invert[i][j]=matrix_to_pass[i+1][j+1];
        }
    return(return_value);
    }


void mult_2_mat(double a[2][2], double b[2][2], double c[2][2]) {
// Multiplies 2x2 matrices 'a' and 'b' to give 'c'
    int i,j,k;

    for(i=0;i<2;i++) {
        for(j=0;j<2;j++) {
            c[i][j]=0.0;
            for(k=0;k<2;k++) c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }

int inv_2_mat(double matrix_to_invert[2][2]) {
// Inverts a 2x2 matrix, returns -1 if not successful
    int invmat(int n, double a[MATRIX_SIZE][MATRIX_SIZE]);
    int i,j, return_value;
    double matrix_to_pass[MATRIX_SIZE][MATRIX_SIZE];

    for(i=0;i<2;i++) {
        for(j=0;j<2;j++) matrix_to_pass[i+1][j+1]=matrix_to_invert[i][j];
        }
    return_value=invmat(2, matrix_to_pass);
    for(i=0;i<2;i++) {
        for(j=0;j<2;j++) matrix_to_invert[i][j]=matrix_to_pass[i+1][j+1];
        }
    return(return_value);
    }
```

Start of **mult_4_mat()** function

End of **mult_4_mat()** function

Start of **inv_4_mat()** function

End of **inv_4_mat()** function

Start of **mult_5_mat()** function

End of **mult_5_mat()** function

Start of **inv_5_mat()** function

```
void mult_4_mat(double a[4][4], double b[4][4], double c[4][4]) {
// Multiplies 4x4 matrices 'a' and 'b' to give 'c'
    int i,j,k;

    for(i=0;i<4;i++) {
        for(j=0;j<4;j++) {
            c[i][j]=0.0;
            for(k=0;k<4;k++) c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }

void inv_4_mat(double matrix_to_invert[4][4]) {
        // Inverts a 4x4 matrix
    void gaussj(float **a, int n, float **b, int m);
    float **matrix(int nrl, int nrh, int ncl, int nch);
    void free_matrix(float **m, int nrl, int nrh, int ncl, int nch);

    float **matrix_to_pass, **ans;
    int i,j;

    matrix_to_pass=matrix(1,4,1,4);
    ans=matrix(1,4,1,4);

    for(i=0;i<4;i++) {
        for(j=0;j<4;j++) {
            matrix_to_pass[i+1][j+1]=(float)matrix_to_invert[i][j];
            if(i==j) ans[i+1][j+1]=1.0;
            else ans[i+1][j+1]=0.0;
            }
        }
    gaussj(matrix_to_pass,4,ans,4);

    for(i=0;i<4;i++) {
        for(j=0;j<4;j++) matrix_to_invert[i][j]=(double)matrix_to_pass[i+1][j+1];
        }

    free_matrix(matrix_to_pass,1,4,1,4);
    free_matrix(ans,1,4,1,4);
    }


void mult_5_mat(double a[5][5], double b[5][5], double c[5][5]) {
// Multiplies 5x5 matrices 'a' and 'b' to give 'c'
    int i,j,k;

    for(i=0;i<5;i++) {
        for(j=0;j<5;j++) {
            c[i][j]=0.0;
            for(k=0;k<5;k++) c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }


int inv_5_mat(double matrix_to_invert[5][5]) {
// Inverts a 5x5 matrix, returns -1 if not successful
    int invmat(int n, double a[MATRIX_SIZE][MATRIX_SIZE]);
    int i,j, return_value;
    double matrix_to_pass[MATRIX_SIZE][MATRIX_SIZE];

    for(i=0;i<5;i++) {
```

End of **inv_5_mat**() function

```
    for(j=0;j<5;j++) matrix_to_pass[i+1][j+1]=matrix_to_invert[i][j];
    }
return_value=invmat(5, matrix_to_pass);
for(i=0;i<5;i++) {
    for(j=0;j<5;j++) matrix_to_invert[i][j]=matrix_to_pass[i+1][j+1];
    }
return(return_value);
}
```

23

TETHER.C

Start of **tether_init()** function

Set tether constants, e.g. drag coefficients, node spacing, etc.

Set initial tether velocities and tensions;
initial tangential velocity is equal to initial vehicle surge velocity

Set initial tether positions and angles;
tether starts as a straight-line streamer behind the vehicle

Set tether mass and added mass

End of **tether_init()** function

Start of **tether()** function;
only compile if using the tether model (see OPT.H)

```c
// Author:    R.K.Lea
// Date:      9 June 1997
// File:      TETHER.C
// Notes:     Simulates tether dynamics in the horizontal plane

#include <math.h>
#include <stdlib.h>

#include <sub.h>
#include <rov_ext.h>
#include <opt.h>

TETHER t;

void tether_init (STATE *s) {
    int i,j;
    const double mass_per_m = 6.5e-3;
    t.cable_length=20.0;
    t.space_step=t.cable_length/(float)(TETHER_POINTS-1);
    t.Cdn=1.0;
    t.Cdt=0.01;
    t.diam=2.5e-3;
    t.precision=1.0e-6;
    t.sl_length=0.0;

    for(i=TETHER_POINTS-1;i>=0;i--) {
        t.u_i[i]=s->u;       t.u_iplus1[i]=0.0;
        t.v_i[i]=0.0;    t.v_iplus1[i]=0.0;
        t.t_i[i]=0.0;
        if(i==TETHER_POINTS-1) {
            t.x_i[i]=s->x_o-0.97/2.0*cos(s->psi);
            t.y_i[i]=s->y_o-0.97/2.0*sin(s->psi);
            t.phi_i[i]=PIBY2-s->psi;
            }
        else {
            t.x_i[i]=t.x_i[i+1]-t.space_step*cos(s->psi);
            t.y_i[i]=t.y_i[i+1]-t.space_step*sin(s->psi);
            t.phi_i[i]=t.phi_i[i+1];
            }
        t.phi_iplus1[i]=t.phi_i[i];  t.phi_iminus1[i]=t.phi_i[i];
        t.x_iplus1[i]=t.x_i[i];        t.y_iplus1[i]=t.y_i[i];
        }
    t.m = mass_per_m;
    t.ma = 0.25*PI*s->density*SQR(t.diam);
    t.m1 = t.m+t.ma;
//  t.EI=210e9*PI*SQR(0.5e-4)*SQR(0.5e-4)/4.0*28.0;
    t.EI=0.01;
    }

void tether(STATE *s, double time_step) {
#if(TETHER_DYNAMICS)
    void mult_t_mat(double a[TETHER_POINTS][TETHER_POINTS],
            double b[TETHER_POINTS][TETHER_POINTS], double c[TETHER_POINTS][TETHER_POINTS],
            int skip);
    void mult_t_vect(double a[TETHER_POINTS][TETHER_POINTS], double b[TETHER_POINTS],
            double c[TETHER_POINTS], int skip);
    int inv_t_mat(int n, double a[TETHER_POINTS][TETHER_POINTS]);

    register int i,j,k;
```

Initialise tempory variables

$\overline{x}$ terms are x with numerical damping added;
'flange' is the amount of damping used i.e. none at the moment,
but the facility is here to add some if so desired

Find ϕ_j^{i+1}, i.e. ϕs for next time step

```
    double max_diff;
    double lambda_r=time_step/(2*t.space_step);
    double a[TETHER_POINTS], c[TETHER_POINTS], t_new[TETHER_POINTS];
    double temp1[TETHER_POINTS][TETHER_POINTS], tempv[TETHER_POINTS];
    double B[TETHER_POINTS][TETHER_POINTS], D[TETHER_POINTS][TETHER_POINTS],
           E[TETHER_POINTS][TETHER_POINTS], G[TETHER_POINTS][TETHER_POINTS],
           L_U[TETHER_POINTS][TETHER_POINTS], E_B[TETHER_POINTS][TETHER_POINTS];
    double u_bar[TETHER_POINTS], v_bar[TETHER_POINTS], phi_bar[TETHER_POINTS],
           phi_iminus1_bar[TETHER_POINTS];
    double veh_PHI=PIBY2-s->psi;
    double u_c=s->u;
    double v_c=s->v-0.5*s->r;
    double flange;

    for (i=0;i<TETHER_POINTS;i++) {
//      a[i]=0.0;
//      c[i]=0.0;
        tempv[i]=0.0;
//      u_bar[i]=0.0;
//      v_bar[i]=0.0;
//      phi_bar[i]=0.0;
        for (j=0;j<TETHER_POINTS;j++) {
//          temp1[i][j]=0.0;
            B[i][j]=0.0;
            D[i][j]=0.0;
            E[i][j]=0.0;
            G[i][j]=0.0;
            }
        }

    flange=0.0;
    u_bar[0]=t.u_i[0]+flange*(t.u_i[1]-2*t.u_i[2]+t.u_i[3]);
    v_bar[0]=t.v_i[0]+flange*(t.v_i[1]-2*t.v_i[2]+t.v_i[3]);
    phi_bar[0]=t.phi_i[0]+flange*(t.phi_i[1]-2*t.phi_i[2]+t.phi_i[3]);
    phi_iminus1_bar[0]=t.phi_iminus1[0]
           +flange*(t.phi_iminus1[1]-2*t.phi_iminus1[2]+t.phi_iminus1[3]);
    for (j=1;j<TETHER_POINTS-1;j++) {
        u_bar[j]=t.u_i[j]+flange*(t.u_i[j-1]-2*t.u_i[j]+t.u_i[j+1]);
        v_bar[j]=t.v_i[j]+flange*(t.v_i[j-1]-2*t.v_i[j]+t.v_i[j+1]);
        phi_bar[j]=t.phi_i[j]+flange*(t.phi_i[j-1]-2*t.phi_i[j]+t.phi_i[j+1]);
        phi_iminus1_bar[j]=t.phi_iminus1[j]
               +flange*(t.phi_iminus1[j-1]-2*t.phi_iminus1[j]+t.phi_iminus1[j+1]);
        }
    u_bar[TETHER_POINTS-1]=t.u_i[TETHER_POINTS-1]+flange*(t.u_i[TETHER_POINTS-1]
           -2*t.u_i[TETHER_POINTS-2]+t.u_i[TETHER_POINTS-3]);
    v_bar[TETHER_POINTS-1]=t.v_i[TETHER_POINTS-1]+flange*(t.v_i[TETHER_POINTS-1]
           -2*t.v_i[TETHER_POINTS-2]+t.v_i[TETHER_POINTS-3]);
    phi_bar[TETHER_POINTS-1]=t.phi_i[TETHER_POINTS-1]+flange*
           (t.phi_i[TETHER_POINTS-1]-2*t.phi_i[TETHER_POINTS-2]+t.phi_i[TETHER_POINTS-3]);
    phi_iminus1_bar[TETHER_POINTS-1]=t.phi_iminus1[TETHER_POINTS-1]+flange*
           (t.phi_iminus1[TETHER_POINTS-1]-2*t.phi_iminus1[TETHER_POINTS-2]
           +t.phi_iminus1[TETHER_POINTS-3]);


    t.phi_iplus1[0]=phi_bar[0]-lambda_r*(t.v_i[2]-4.0*t.v_i[1]+3.0*t.v_i[0]
                      +t.u_i[0]*(t.phi_i[2]-4.0*t.phi_i[1]+3.0*t.phi_i[0]));
    for (j=1;j<TETHER_POINTS-1;j++)
           t.phi_iplus1[j]=phi_bar[j]+lambda_r*(t.v_i[j+1]-t.v_i[j-1]
                              +t.u_i[j]*(t.phi_i[j+1]-t.phi_i[j-1]));
    t.phi_iplus1[TETHER_POINTS-1]=phi_bar[TETHER_POINTS-1]+lambda_r*
           (t.v_i[TETHER_POINTS-3]-4.0*t.v_i[TETHER_POINTS-2]+3.0*t.v_i[TETHER_POINTS-1]
           +t.u_i[TETHER_POINTS-1]*(t.phi_i[TETHER_POINTS-3]
```

$$
\boldsymbol{B} = \frac{\lambda_r}{m}\begin{bmatrix} -3 & 4 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}
$$

$$
\boldsymbol{E} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & -4 & 3 \end{bmatrix}
$$

$$
\boldsymbol{D} = \frac{\lambda_r}{m_1}\begin{bmatrix} -\phi_2^i + 4\phi_1^i - 3\phi_0^i & 0 & 0 & 0 & 0 \\ 0 & \phi_2^i - \phi_0^i & 0 & 0 & 0 \\ 0 & 0 & \phi_3^i - \phi_1^i & 0 & 0 \\ 0 & 0 & 0 & \phi_4^i - \phi_2^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}
$$

$$
\boldsymbol{G} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \phi_2^{i+1} - \phi_0^{i+1} & 0 & 0 & 0 \\ 0 & 0 & \phi_3^{+1i} - \phi_1^{i+1} & 0 & 0 \\ 0 & 0 & 0 & \phi_4^{i+1} - \phi_2^{i+1} & 0 \\ 0 & 0 & 0 & 0 & \phi_{n-3}^{i+1} - 4\phi_{n-2}^{i+1} + 3\phi_{n-1}^{i+1} \end{bmatrix}
$$

$$
\boldsymbol{a} = \begin{bmatrix} \overline{u}_0 \\ \overline{u}_1 \\ \overline{u}_2 \\ \overline{u}_3 \\ u_{connection\ point} \end{bmatrix}^i + \frac{1}{2}\begin{bmatrix} \phi_0^{i+1} - \overline{\phi}_0^i & 0 & 0 & 0 & 0 \\ 0 & \phi_1^{i+1} - \overline{\phi}_1^i & 0 & 0 & 0 \\ 0 & 0 & \phi_2^{i+1} - \overline{\phi}_2^i & 0 & 0 \\ 0 & 0 & 0 & \phi_3^{i+1} - \overline{\phi}_3^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_{n-1} \end{bmatrix}^i - \frac{\Delta t}{2m}\rho D\pi Cd_t\begin{bmatrix} u_0^i|u_0^i| \\ u_1^i|u_1^i| \\ u_2^i|u_2^i| \\ u_3^i|u_3^i| \\ 0 \end{bmatrix}
$$

If using the bending moments model,

$$
a_0 + = \frac{EI\lambda_r}{m(\Delta s)^2}\left(-\phi_2^i + 4\phi_1^i - 3\phi_0^i\right)\left(-\phi_3^i + 4\phi_2^i - 5\phi_1^i + 2\phi_0^i\right)
$$

$$
a_j + = \frac{EI\lambda_r}{m(\Delta s)^2}\left(\phi_{j+1}^i - \phi_{j-1}^i\right)\left(\phi_{j+1}^i - 2\phi_j^i + \phi_{j-1}^i\right)
$$

$$
n - 1 \geq j \geq 1
$$

$$
\boldsymbol{c} = \begin{bmatrix} \overline{v}_0 \\ \overline{v}_1 \\ \overline{v}_2 \\ \overline{v}_3 \\ v_{connection\ point} \end{bmatrix}^i - \frac{m}{m_1}\begin{bmatrix} \phi_0^{i+1} - \overline{\phi}_0^i & 0 & 0 & 0 & 0 \\ 0 & \phi_1^{i+1} - \overline{\phi}_1^i & 0 & 0 & 0 \\ 0 & 0 & \phi_2^{i+1} - \overline{\phi}_2^i & 0 & 0 \\ 0 & 0 & 0 & \phi_3^{i+1} - \overline{\phi}_3^i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_{n-1} \end{bmatrix}^i - \frac{\Delta t}{2m_1}\rho DCd_n\begin{bmatrix} v_0^i|v_0^i| \\ v_1^i|v_1^i| \\ v_2^i|v_2^i| \\ v_3^i|v_3^i| \\ 0 \end{bmatrix}
$$

If using the bending moments model,

$$
c_0 - = \frac{EI\lambda_r}{m_1(\Delta s)^2}\left(-3\phi_4^i + 14\phi_3^i - 24\phi_2^i + 18\phi_1^i - 5\phi_0^i\right)
$$

$$
c_j - = \frac{EI\lambda_r}{m_1(\Delta s)^2}\left(\phi_{j+2}^i - 2\phi_{j+1}^i + 2\phi_{j-1}^i - \phi_{j-2}^i\right)
$$

$$
n - 1 \geq j \geq 1
$$

```
          -4.0*t.phi_i[TETHER_POINTS-2] +3.0*t.phi_i[TETHER_POINTS-1]));

    B[0][0]=-3.0*lambda_r/t.m; B[0][1]=4.0*lambda_r/t.m;
    B[0][2]=-1.0*lambda_r/t.m;
    for (j=1;j<TETHER_POINTS-1;j++) {
       B[j][j-1]=-1.0*lambda_r/t.m;
       B[j][j+1]=1.0*lambda_r/t.m;
       }

    for (j=1;j<TETHER_POINTS-1;j++) {
       E[j][j-1]=-1.0;
       E[j][j+1]=1.0;
       }
    E[TETHER_POINTS-1][TETHER_POINTS-3]=1.0;
    E[TETHER_POINTS-1][TETHER_POINTS-2]=-4.0;
    E[TETHER_POINTS-1][TETHER_POINTS-1]=3.0;

    D[0][0]=-lambda_r/t.m1*(t.phi_i[2]-4.0*t.phi_i[1]+3.0*t.phi_i[0]);
    for (j=1;j<TETHER_POINTS-1;j++)
          D[j][j]=lambda_r/t.m1*(t.phi_i[j+1]-t.phi_i[j-1]);




    for (j=1;j<TETHER_POINTS-1;j++) G[j][j]=t.phi_iplus1[j+1]-t.phi_iplus1[j-1];
    G[TETHER_POINTS-1][TETHER_POINTS-1]=t.phi_iplus1[TETHER_POINTS-3]
          -4.0*t.phi_iplus1[TETHER_POINTS-2]+3.0*t.phi_iplus1[TETHER_POINTS-1];




    for (j=0;j<TETHER_POINTS-1;j++) {
       a[j]=u_bar[j]+0.5*(t.phi_iplus1[j]-phi_iminus1_bar[j])*t.v_i[j]
             -time_step/(2.0*t.m)*s->density*t.diam*PI*t.Cdt*t.u_i[j]
             *fabs(t.u_i[j]);
#if(BENDING==TRUE)
       if(j==0) a[j]+= t.EI*lambda_r/(t.m*SQR(t.space_step))
             *(-t.phi_i[2]+4.0*t.phi_i[1]-3.0*t.phi_i[0])
             *(-t.phi_i[3]+4.0*t.phi_i[2]-5.0*t.phi_i[1]+2.0*t.phi_i[0]);
       else a[j]+= t.EI*lambda_r/(t.m*SQR(t.space_step))
             *(t.phi_i[j+1]-t.phi_i[j-1])*(t.phi_i[j+1]-2.0*t.phi_i[j] +t.phi_i[j-1]);
#endif
       }
    a[TETHER_POINTS-1]=u_c*cos(t.phi_iplus1[TETHER_POINTS-1]-veh_PHI)
                      -v_c*sin(t.phi_iplus1[TETHER_POINTS-1]-veh_PHI);

    for (j=0;j<TETHER_POINTS-1;j++) {
       c[j]=v_bar[j]-t.m/t.m1*0.5*(t.phi_iplus1[j]-phi_iminus1_bar[j])*t.u_i[j]
             -time_step/(2.0*t.m1)*s->density*t.diam*t.Cdn*t.v_i[j]
             *fabs(t.v_i[j]);
#if(BENDING==TRUE)
       if(j<=1) c[j]-= t.EI*lambda_r/(t.m1*SQR(t.space_step))
             *(-3.0*t.phi_i[j+4]+14.0*t.phi_i[j+3]-24.0*t.phi_i[j+2]+18.0
             *t.phi_i[j+1]-5.0*t.phi_i[j]);
       else if(j==TETHER_POINTS-2) c[j]-= t.EI*lambda_r/(t.m1*SQR(t.space_step))
             *(3.0*t.phi_i[j-4]-14.0*t.phi_i[j-3]+24.0*t.phi_i[j-2]
             -18.0*t.phi_i[j-1]+5.0*t.phi_i[j]);
       else c[j]-= t.EI*lambda_r/(t.m1*SQR(t.space_step))
             *(t.phi_i[j+2]-2.0*t.phi_i[j+1]+2.0*t.phi_i[j-1]-t.phi_i[j-2]);
#endif
       }
    c[TETHER_POINTS-1]=-u_c*sin(t.phi_iplus1[TETHER_POINTS-1]-veh_PHI)
```

We want to solve $\boldsymbol{E}(\boldsymbol{a}+\boldsymbol{Bt}) = \boldsymbol{G}(\boldsymbol{c}+\boldsymbol{Dt}) \Rightarrow \boldsymbol{Ea}+\boldsymbol{EBt} = \boldsymbol{Gc}+\boldsymbol{GDt}$
so find $(\boldsymbol{EB}-\boldsymbol{GD})^{-1}$ — note that $\boldsymbol{G}$ and $\boldsymbol{D}$ are diagonal

Find $(\boldsymbol{Gc}-\boldsymbol{Ea})$ here — note again that $\boldsymbol{G}$ is diagonal and $\boldsymbol{E}$ is pseudo-diagonal

Now $\boldsymbol{t} = (\boldsymbol{EB}-\boldsymbol{GD})^{-1}(\boldsymbol{Gc}-\boldsymbol{Ea})$

Find tangential velocities for next time step:
$\boldsymbol{u}^{i+1} = \boldsymbol{a}+\boldsymbol{Bt}$

Find normal velocities for next time step:
$\boldsymbol{v}^{i+1} = \boldsymbol{c}+\boldsymbol{Dt}$

Find new node positions given current velocities

Update new velocities, positions, etc.

End of **tether()** function

Start of **mult_t_mat()** function

This and the following routines are used to perform matrix manipulation for the tether matrices and vectors; this should be handled by the routines in NEW_MATH.C instead, but again there were problems with stack and memory sizes

End of **mult_t_mat()** function

Start of **mult_t_vect()** function

```
                    -v_c*cos(t.phi_iplus1[TETHER_POINTS-1]-veh_PHI);

    mult_t_mat(E, B, temp1, 0);
    for (j=1;j<TETHER_POINTS-2;j++) temp1[j][j]-=G[j][j]*D[j][j];
    inv_t_mat(TETHER_POINTS-1, temp1);

    for (j=1;j<TETHER_POINTS-1;j++) tempv[j]=G[j][j]*c[j]+a[j-1]-a[j+1];
    tempv[TETHER_POINTS-1]=G[TETHER_POINTS-1][TETHER_POINTS-1]*c[TETHER_POINTS-1]
        -a[TETHER_POINTS-3]+4*a[TETHER_POINTS-2]-3*a[TETHER_POINTS-1];

    mult_t_vect(temp1, tempv, t.t_i, 1);
    t.t_i[0]=0.0;

    mult_t_vect(B, t.t_i, t.u_iplus1, 0);
    for (j=0;j<TETHER_POINTS;j++) t.u_iplus1[j]+=a[j];

    mult_t_vect(D, t.t_i, t.v_iplus1, 0);
    for (j=0;j<TETHER_POINTS;j++) t.v_iplus1[j]+=c[j];

    for (j=0;j<TETHER_POINTS;j++){
        t.x_iplus1[j]=t.x_i[j]
            +time_step*(t.u_i[j]*sin(t.phi_i[j])+t.v_i[j]*cos(t.phi_i[j]));
        t.y_iplus1[j]=t.y_i[j]
            +time_step*(t.u_i[j]*cos(t.phi_i[j])-t.v_i[j]*sin(t.phi_i[j]));
        }

    for (j=0;j<TETHER_POINTS;j++){
        t.x_i[j]=t.x_iplus1[j];
        t.y_i[j]=t.y_iplus1[j];
        t.u_i[j]=t.u_iplus1[j];
        t.v_i[j]=t.v_iplus1[j];
        t.phi_iminus1[j]=t.phi_i[j];
        t.phi_i[j]=t.phi_iplus1[j];
        }
    }


void mult_t_mat(double a[TETHER_POINTS][TETHER_POINTS],
        double b[TETHER_POINTS][TETHER_POINTS], double c[TETHER_POINTS][TETHER_POINTS],
        int skip) {
// Multiplies two TETHER_POINTS x TETHER_POINTS matrices 'a' and 'b' to give 'c'
// It misses out the first 'skip' rows and columns
    register  int i,j,k;

    for(i=skip;i<TETHER_POINTS;i++) {
        for(j=skip;j<TETHER_POINTS;j++) {
            c[i][j]=0.0;
            for(k=skip;k<TETHER_POINTS;k++) c[i][j]+=a[i][k]*b[k][j];
            }
        }
#endif
    }

#if(TETHER_DYNAMICS)
void mult_t_vect(double a[TETHER_POINTS][TETHER_POINTS], double b[TETHER_POINTS],
        double c[TETHER_POINTS], int skip) {
// Multiplies matrix 'a' by vector 'b' to give 'c'.
    register  int i,k;

    for(i=skip;i<TETHER_POINTS;i++) {
        c[i]=0.0;
        for(k=skip;k<TETHER_POINTS;k++) c[i]+=a[i][k]*b[k];
```

End of **mult_t_vect()** function

Start of **inv_t_vect()** function

End of **inv_t_vect()** function

Start of **t_gauss()** function

```
        }
    }

int inv_t_mat(int n, double a[TETHER_POINTS][TETHER_POINTS]) {
    int t_gauss(int n, int m, double a[TETHER_POINTS][2*TETHER_POINTS]);
    double b[TETHER_POINTS][2*TETHER_POINTS];     /* work space matrix */
    register int i, j;                            /* loop counters */
    for( i = 1; i <= n; i++) {
        for( j = 1; j <= n; j++) {
            b[i][j] = a[i][j];
            b[i][n+j] = 0.0;
            if( i == j )  b[i][n+j] = 1.0;
            }
        }
    i = t_gauss(n,n,b);
    if(i==-1) return(-1);
    for(i=1;i<=n;i++) for( j = 1; j <= n; j++) a[i][j] = b[i][j+n];
    return(0);
    }


int t_gauss(int n, int m, double a[TETHER_POINTS][2*TETHER_POINTS]) {
    double u, x;
    int kk, in, ie;
    register int i,j,k;
    if(n>1) {
        for( k = 1; k < n; k++) {
            u = fabs(a[k][k]);
            kk = k + 1;
            in = k;
            for( i = kk; i <= n; i++) {
                if( fabs(a[i][k]) > u) {
                    u = fabs(a[i][k]);
                    in = i;
                    }
                }
            if( k != in ) {
                for( j = k; j <= n+m; j++) {
                    x = a[k][j];
                    a[k][j] = a[in][j];
                    a[in][j] = x;
                    }
                }
            if( u < PRECISION ) return(-1);
            for( i = kk; i <= n; i++) {
                for( j = kk; j <= n+m; j++ ) {
                     if( a[k][k] != 0.0 ) a[i][j] += -a[i][k]*a[k][j] / a[k][k];
                     else return(-1);
                     }
                 }
            } /* end for k */
        if( fabs(a[n][n]) < PRECISION ) return(-1);
        for( k = 1; k <= m; k++) {
            a[n][n+k] = a[n][n+k] / a[n][n];
            for( ie = 1; ie < n; ie++) {
                i = n - ie;
                in = i + 1;
                for( j = in; j <= n; j++) a[i][n+k] += -a[j][n+k]*a[i][j];
                a[i][n+k] = a[i][n+k] / a[i][i];
                }
            }
        return(0);    /* solution */
```

End of **t_gauss**() function

```
        }
    else  {
        if( fabs(a[1][1]) < PRECISION) return(-1);
        for(j = 1; j <= m; j++) a[1][n+j] = a[1][n+j] / a[1][1];
        return(0);
        }
    }
#endif
```

24

NEW_MATH.C

Read the book to find out more about these functions!

Start of **gaussj()** function

```
// Author:    R.K.Lea
// Date:      4 July 1997
// File:      NEW_MATH.C
// Notes:     New math routines for matrix inversion
//            Taken from W.H.Press, B.P.Flannery, S.A.Teukolsky & W.T.Vettering,
//            Numerical Recipes in C - The Art of Scientific Computing.
//            Cambridge University Press, Cambridge, UK, 1988.

#include <math.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

#include <sim.h>
#include <sub.h>

#define SWAP(a,b) {float temp=(a);(a)=(b);(b)=temp;}

void gaussj(float **a, int n, float **b, int m){
   /*
   Linear equation solution by Gauss-Jordan elimination.  a[1..n][1..n] is the input
   matrix.  b[1..m][1..m] is the input containing the m right-hand side vectors.  On
   output, a is replaced by its matrix inverse, and b is replaced by the corresponding
   set of solution vectors.
   */

   int *indxc, *indxr, *ipiv;
   int i, icol, irow, j, k, l, ll, *ivector(int nl, int nh);
   float big, dum, pivinv;
   void nrerror(char error_text[]), free_ivector(int *v, int nl, int nh);

   indxc=ivector(1,n);
   indxr=ivector(1,n);
   ipiv=ivector(1,n);
   for (j=1;j<=n;j++) ipiv[j]=0;
   for (i=1;i<=n;i++) {
      big=0.0;
      for(j=1;j<=n;j++) if(ipiv[j] !=1) for(k=1;k<=n;k++) {
         if(ipiv[k]==0) {
            if(fabs(a[j][k])>=big) {
               big=fabs(a[j][k]);
               irow=j;
               icol=k;
               }
            } else if (ipiv[k]>1) nrerror("GAUSSJ: Singular Matrix-1");
         }
      ++(ipiv[icol]);
      if(irow!=icol){
         for (l=1;l<=n;l++) SWAP(a[irow][l],a[icol][l])
         for (l=1;l<=m;l++) SWAP(b[irow][l],b[icol][l])
         }
      indxr[i]=irow;
      indxc[i]=icol;
      if(a[icol][icol]==0.0) nrerror("GAUSSJ: Singular Matrix-2");
      pivinv=1.0/a[icol][icol];
      a[icol][icol]=1.0;
      for (l=1;l<=n;l++) a[icol][l]*=pivinv;
      for (l=1;l<=m;l++) b[icol][l]*=pivinv;
      for(ll=1;ll<=n;ll++) if(ll!=icol) {
```

End of **gaussj()** function

Start of **nrerror()** function

End of **nrerror()** function

Start of **ivector()** function

End of **ivector()** function

Start of **free_ ivector()** function

End of **free_ ivector()** function

Start of **matrix()** function

End of **matrix()** function

Start of **free_matrix()** function

End of **free_matrix()** function

Start of **submatrix()** function

```
            dum=a[ll][icol];
            a[ll][icol]=0.0;
            for (l=1;l<=n;l++) a[ll][l]-=a[icol][l]*dum;
            for (l=1;l<=m;l++) b[ll][l]-=b[icol][l]*dum;
            }
        }
    for(l=n;l>=1;l--) {
        if(indxr[l]!=indxc[l]) SWAP(a[k][indxr[l]],a[k][indxc[l]]);
        }
    free_ivector(ipiv,1,n);
    free_ivector(indxr,1,n);
    free_ivector(indxc,1,n);
    }

void nrerror(char error_text[]) {
    printf("Numerical Recipes run-time error...\n");
    printf("%s\n",error_text);
    printf("...now exiting to system...\n");
    exit(1);
    }

int *ivector(int nl, int nh){
    // Allocate an int vector with subscript range v[nl..nh]
    int *v;
    v=(int *)malloc((unsigned)(nh-nl+1)*sizeof(int));
    if(!v) nrerror("allocation failure in ivector()");
    return v-nl;
    }

void free_ivector(int *v, int nl, int nh){
    // Free an int vector allocated by ivector()
    free((char*)(v+nl));
    }

float **matrix(int nrl, int nrh, int ncl, int nch) {
    // Allocate a float matrix with subscript range m[nrl..nrh][ncl..nch]
    int i;
    float **m;
    m=(float **)malloc((unsigned)(nrh-nrl+1)*sizeof(float*));
    if(!m)nrerror("allocation failure 1 in matrix()");
    m-=nrl;
    for(i=nrl;i<=nrh;i++) {
        m[i]=(float *)malloc((unsigned)(nch-ncl+1)*sizeof(float));
        if (!m[i])nrerror("allocation failure 2 in matrix()");
        m[i]-=ncl;
        }
    return m;
    }

void free_matrix(float **m, int nrl, int nrh, int ncl, int nch) {
    // Frees a float matrix allocated by matrix()
    int i;
    for(i=nrh;i>=nrl;i--) free((char*)(m[i]+ncl));
    free((char*)(m+nrl));
    }

float **submatrix(float **a, int oldrl, int oldrh, int oldcl, int oldch,
        int newrl, int newcl){
    // Point a submatrix [newrl..][newcl..] to a[oldrl..oldrh][oldcl..oldch]
    int i,j;
    float **m;
    m=(float **)malloc((unsigned)(oldrh-oldrl+1)*sizeof(float*));
```

End of **submatrix()** function

Start of **free_submatrix()** function

End of **free_submatrix()** function

Start of **big_gauss()** function

```
    if(!m)nrerror("allocation failure in submatrix()");
    m-=newrl;
    for(i=oldrl,j=newrl;i<=oldrh;i++,j++) m[j]=a[i]+oldcl-newcl;
    return m;
    }

void free_submatrix(float **b, int nrl, int nrh, int ncl, int nch) {
    // Free a submatrix allocated by submatrix()
    free((char*)(b+nrl));
    }


// The following routines are modified from the functions found in the program
// for inverting the mass matrix; these are used mostly for the STR autopilot

int big_gauss(int n, int m, double a[13][2*13]) {
    double u, x;   /* temp variables */
    int k, kk, in, ie, i, j;      /* loop counters etc... */

    if( n > 1 ) {
        for( k = 1; k < n; k++) {
            u = fabs(a[k][k]);
            kk = k + 1;
            in = k;

            /* search for index in of maximum pivot value */
            for( i = kk; i <= n; i++) {
                if( fabs(a[i][k]) > u) {
                    u = fabs(a[i][k]);
                    in = i;
                    }
                } /* end for i */

            if( k != in ) {
                for( j = k; j <= n+m; j++) {  /* interchange rows k and index in */
                    x = a[k][j];
                    a[k][j] = a[in][j];
                    a[in][j] = x;
                    }
                }

            if( u < PRECISION ) {  /* check if pivot too small */
                return(-1);  /* matrix is singular */
                }

            for( i = kk; i <= n; i++) {  /* forward elimination step */
                for( j = kk; j <= n+m; j++ ) {
                     if( a[k][k] != 0.0 ) a[i][j] += -a[i][k]*a[k][j] / a[k][k];
                     else return(-1);  /* division by zero */
                     }
                 }
            } /* end for k */

        if( fabs(a[n][n]) < PRECISION ) return(-1);  /* division by zero */

        for( k = 1; k <= m; k++) {  /* back substitution */
            a[n][n+k] = a[n][n+k] / a[n][n];
            for( ie = 1; ie < n; ie++) {
                i = n - ie;
                in = i + 1;
                for( j = in; j <= n; j++) a[i][n+k] += -a[j][n+k]*a[i][j];
                a[i][n+k] = a[i][n+k] / a[i][i];
```

End of **big_gauss**() function

Start of **inv_big_mat**() function

End of **inv_big_mat**() function

```
            }
        }
        return(0);    /* solution */
    }
else {  /* n > 1 */
    if( fabs(a[1][1]) < PRECISION) return(-1); /* division by zero */

    for(j = 1; j <= m; j++) a[1][n+j] = a[1][n+j] / a[1][1];
    return(0);
    }
}


int inv_big_mat(int n, double a[13][13]) {
    int big_gauss(int n, int m, double a[13][2*13]);

    double b[13][2*13];     /* work space matrix */
    int i, j;

    for(i=1; i<=n; i++) for(j=1; j<=n; j++) {
        b[i][j] = a[i][j];
        b[i][n+j] = 0.0;
        if(i==j)  b[i][n+j] = 1.0;
        }
    i = big_gauss(n,n,b);     /* Compute matrix inverse by Gaussian Elimination */
    if(i==-1) return(-1);
    for(i=1; i<=n; i++) for(j=1; j<=n; j++) a[i][j] = b[i][j+n];
    return(0);
    }
```