NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SIMULATION-BASED VALIDATION OF NAVIGATION FILTER SOFTWARE FOR A SHALLOW WATER AUV NAVIGATION SYSTEM (SANS)

by

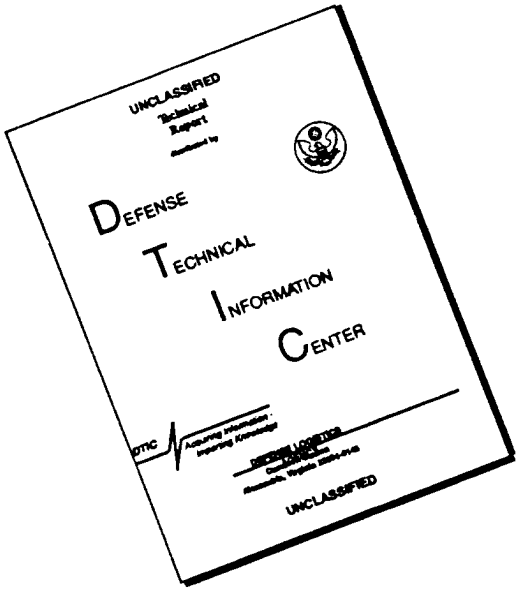
Ruediger Steven

March 1996

Thesis Advisor:	Robert B. McGhee
Second Reader:	Eric Bachmann

Approved for public release; distribution is unlimited.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
March 1996

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
SIMULATION-BASED VALIDATION OF NAVIGATION FILTER SOFTWARE FOR A SHALLOW WATER AUV NAVIGATION SYSTEM (SANS)

5. FUNDING NUMBERS

6. AUTHOR(S)
Steven, Ruediger

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Navigation filter software is currently being developed for an inertial navigation system without rotating gyros. This system shall replace the navigation system that is currently used in the Phoenix Autonomous Underwater Vehicle of the Naval Postgraduate School. The filter combines acceleration sensors, angular rate sensors, a water speed sensor, a magnetic compass and a GPS system. The harmonization of the sensors is performed by gain matrices. The filter code must be tested for correctness and evaluated, and optimal values for the gain matrices must be found. Both factors directly influence the accuracy of the computed positions, and thus the quality of AUV navigation.

In this thesis, the Kalman filter code is tested by experimentation with a simulation of a submarine. Two versions of the code are available, both written in LISP and C++. Test runs are performed in different simulated sea-states (water current), with and without noise added to the sensors, and with different values for the gain matrices.

Based on the experiments, the Kalman filter code seems to be correct and stable. Noise is the most important determinant of the filter performance. The results can be optimized by careful fine tuning of the gain matrices.

14. SUBJECT TERMS
Autonomous vehicle, Robotics, GPS/INS integration, navigation, navigation filter, kalman filter, kinematics, NPS AUV, LISP simulation

15. NUMBER OF PAGES
145

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

SIMULATION-BASED VALIDATION OF NAVIGATION FILTER SOFTWARE FOR A SHALLOW WATER AUV NAVIGATION SYSTEM (SANS)

Ruediger Steven
Lieutenant Commander, German Navy
Diplom-Kaufmann, Universitaet der Bundeswehr , 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1996

Author: Ruediger Steven

Approved By: Robert B McGhee , Thesis Advisor

Eric Bachmann, Second Reader

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Navigation filter software is currently being developed for an inertial navigation system without rotating gyros. This system shall replace the navigation system that is currently used in the Phoenix Autonomous Underwater Vehicle of the Naval Postgraduate School. The filter combines acceleration sensors, angular rate sensors, a water speed sensor, a magnetic compass and a GPS system. The harmonization of the sensors is performed by gain matrices. The filter code must be tested for correctness and evaluated, and optimal values for the gain matrices must be found. Both factors directly influence the accuracy of the computed positions, and thus the quality of AUV navigation.

In this thesis, the Kalman filter code is tested by experimentation with a simulation of a submarine. Two versions of the code are available, both written in LISP and C++. Test runs are performed in different simulated sea-states (water current), with and without noise added to the sensors, and with different values for the gain matrices.

Based on the experiments, the Kalman filter code seems to be correct and stable. Noise is the most important determinant of the filter performance. The results can be optimized by careful fine tuning of the gain matrices.

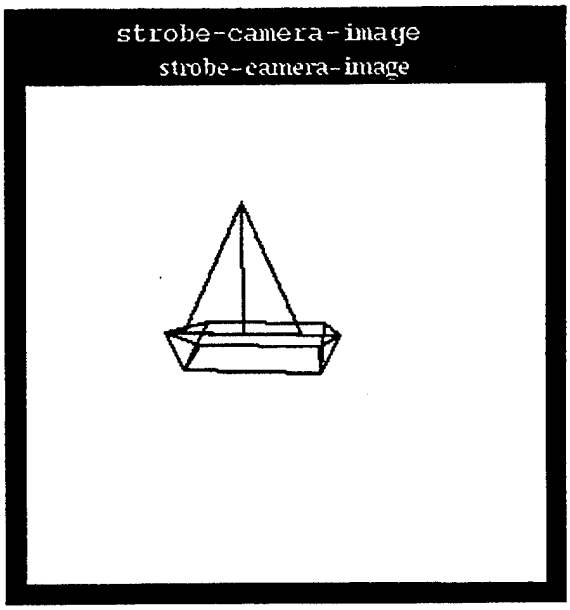
TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank Dr. Robert B. McGhee for his help and advice during this research work. His patience, enthusiasm, endurance and knowledge I find absolutely amazing. Although I will probably never reach his level of knowledge, I hope to achieve at least his level of patience.

I would also like to thank Eric Bachmann for his assistance in editing and correcting my work. He was always open to my questions and made himself available whenever I needed him. He makes the 80-hour-week a working standard.

This research was supported in part by Grant BCS - 9306252 from the National Science Foundation to the Naval Postgraduate School.



SEA-FEVER

I must go down to the seas again, to the lonely sea and the sky,
And all I ask is a tall ship and a star to steer her by,
And the wheel's kick and the wind's song and the white sail's shaking,
And a grey mist on the sea's face and a grey dawn breaking.

I must go down to the seas again, for the call of the running tide
Is a wild call and a clear call that may not be denied;
And all I ask is a windy day with the white clouds flying,
And the flung spray and the blown spume, and the seagulls crying.

I must go down to the seas again to the vagrant gypsy life,
To the gull's way and the whale's way where the wind's like a whetted knife;
And all I ask is a merry yarn from a laughing fellow-rover,
And quiet sleep and a sweet dream when the long trick's over.

John Masefield

I. INTRODUCTION

A. OVERVIEW

Inertial navigation hardware is commonly based on rotating gyros which stabilize a platform that holds acceleration sensors. There are some limiting factors to this approach: this setup is very expensive, because gyros and acceleration sensors must be of high precision, rotating gyros consume a not inconsiderable amount of electrical power, are prone to failure, and they may generate acoustic and structure-borne noise [Cox 94].

These factors are of importance in the philosophy of the Phoenix Autonomous Underwater Vehicle (AUV) of the Naval Postgraduate School [McGhee 95] for two reasons. First, one defined goal for the AUV is to provide a low cost general purpose platform, and second, the intended mission profile of the AUV is based on long-term independent operation in spite of the relatively small size of the vehicle which limits the available space for power supplies such as batteries.

The navigation filter developed by [McGhee 95] at the Naval Postgraduate School tries to solve the problems of cost and power consumption by eliminating rotating gyros and replacing them with acceleration and angular rate sensors. This also results in a smaller inertial module, leaving more space for payload. The problem is to accurately sense accelerations evoked by tilting the platform, e.g., in the surf or in the dive or surface processes of the vehicle, and filter out noise effectively enough to obtain accurate navigational information.

B. RESEARCH QUESTIONS

This thesis will examine the following research areas:

Software for the navigation filter for a low cost shallow water AUV navigation system (SANS), written in LISP by the author (adapted from [McGhee 96]) and in C++ by [Bachmann 95], will be evaluated by simulation. Both software versions should return identical results. This outcome will be accepted as a verification of the correctness of the code. Data for the simulation will be produced artificially by the LISP simulation and fed into the LISP and C++ filter code.

At the start of this work, the LISP simulation code was lacking pitch and roll control. The vehicle could only be moved in two dimensions. Adding pitch and roll control is important, because pitch and roll movements of the vehicle will disturb the accelerometers and angular-rate sensors.

One of the most important parts of the navigation filter is the selection of values for the gain matrices. In [Bachmann 95], it is shown that different combinations of gain values greatly influence the quality of the estimated position. This thesis will try to find suitable values for the gain matrices based on artificially generated data.

In the first version of the filter code, the computed apparent current was added to water speed and north and east velocities. The resulting velocities were integrated with north and east accelerations. This was suspected to lead to an unstable or underdamped behavior of the filter. This work examines this problem. In the new version of the code, the computed apparent current is added directly to the north and east water speed components after the integration of sensed accelerations.

C. ORGANIZATION OF CHAPTERS

Chapter II gives an overview of other proposals to navigate an AUV. Two general directions are described: navigation based on inertial techniques, and navigation based on acoustic techniques, the latter approach being chosen by most authors. A few propose means like video or chemical sensing, which are also briefly introduced.

Chapter III first describes problems encountered in the preceding work. Then the organization of the simulation is described. After this, the code and code changes are presented. Finally, the mission used for the tests in this thesis is introduced.

Chapter IV reports the results, and Chapter V summarizes this thesis, draws conclusions, and proposes areas of future work.

II. REVIEW OF PREVIOUS WORK

A. INTRODUCTION

AUV's are typically developed for tasks that require exact knowledge of position (mine hunting, environmental monitoring, underwater maintenance, etc.), making accurate navigation a prerequisite of a successful mission. Depending on the mission (clandestine or overt operation, operations near or far from the coast, in deep or shallow waters), a certain degree of navigational autonomy must be achieved. Available systems for these purposes range from GPS and inertial sensors to the processing of acoustic, geophysical and visual information [Scherbatyuk 95]. Bachmann and Gay [Bachmann 95] give a broad overview of previous work, especially of GPS. This chapter will give an overview of newer approaches to this problem or approaches not mentioned in the above thesis.

B. NAVIGATION SYSTEMS IN AUVs

1. Inertial and GPS Based Systems

[McGhee 95] describes a navigation system that is based on an inertial navigation unit for submerged navigation and differential GPS (DGPS) for surfaced position updates. Problems with this setup concerning time required to acquire GPS satellites and the influence of water covering the GPS antenna during fixing were examined in [Norton 94].

The system described in [McGhee 95] senses accelerations and angular rates with respective sensors and processes the data in a "nine-state" Kalman filter[1] resulting in an estimated position. To further enhance navigation accuracy, there is also a mechanical water

[1]Counting rate term bias estimates, which are subtracted from the output of the angular-rate sensors, the filter actually has twelve states.

speed sensor and a magnetic compass added to complement acceleration and angular rate data. The nine states can be divided into seven continuous-time states (three Euler angles, two horizontal velocities, two horizontal positions), and two discrete-time states (estimated east and north current), the later derived from the DGPS fixes. The problem with this approach is that DGPS fixes appear aperiodically, whenever the vehicle surfaces and is able to acquire a sufficient number of satellites. This makes it especially difficult to decide how much weight should be given to the updated position and calculated current. [McGhee 95]

An approach quite similar to the above described by [McGhee 95], although designed for long range (1000 km) and deep water (6000 m), is pursued by [McPhail 93]. This navigation system uses GPS for position updates when surfaced, and dead-reckoning when submerged. For attitude estimation, a tri-axis magnetic flux-gate sensor and a tri-axis accelerometer system is used. The author points out that magnetic disturbances in the earth's magnetic field and the perturbance of roll and pitch measurements with accelerometers by vehicle lateral acceleration leads to heading estimation errors of about 1 degree.

While [McGhee 95] uses Euler-angles for vehicle attitude definition, [McPhail 93] uses a direction cosine matrix. Euler-angles are ambiguous when the pitch attitude of the vehicle approaches 90 degrees. Furthermore, the computational handling of direction cosine matrices is more convenient compared to Euler angles.

[Cox 94] points to new developments in inertial navigation emphasizing accuracy, low power consumption, light weight, small size, and low acoustic and structure-borne noise. These goals are achieved with a solid-state Inertial Navigation Unit utilizing laser angular rate sensors. The performance of the unit is enhanced by adding an external velocity

meter coupled with a 19-state Kalman filter. The states are two level positions, two level velocities, three attitudes, three gyro biases, two level accelerometer biases, two level gravity deflection of the vertical, three log states, and two ocean current states.

Navigation is not the only important task of an INU, but also sensor stabilization, especially for devices like laser line scanners and side-scan sonars. Here, the motion of the vehicle must be sensed and compensated to correct quadratic phase error of the received signals and improve image resolution [Cox 94].

2. Acoustic-Based Systems

[Scherbatyuk 95] describes a navigation system that combines on-board sensors such as speed and heading sensors with a long-baseline (LBL) acoustic navigation[2] system and Kalman filtering for position corrections. It is, however, pointed out that LBL has problems such as limited range and noise. Sea bottom conditions and varying sea water densities can disturb signal propagation. This means, correctly received signals must be filtered out from the false ones.

[Atwood 95] have built and tested an AUV that is based on an LBL navigation system with an innovative fix-finding algorithm and commercially available hardware. They use a spherical navigation system, in which the vehicle actively interrogates acoustical transponders and calculates ranges from round trip transit times, resulting in a greater accuracy (about 1 m) compared to the hyperbolic method[3]. The acoustic transponders are

---

[2]A brief but concise overview of long-baseline, short-baseline and ultra-short baseline navigation is presented in [Austin 94]

[3] A hyperbolic navigation system is described in [Bellingham 92].

deployed in an area of about 1 km^2. In this system, the vehicle can use two operating modes, master mode or transponder mode. In the first mode, the vehicle triggers the acoustic transponders, which reply with a ping. The vehicle computer can then calculate distances and, applying acoustically measured depth, a position. In the second operating mode, a surface vessel triggers the vehicle, which in turn interrogates the transponders. Position can then be calculated in the surface vessel through an established GPS position and knowledge of the relative positions of the submerged vehicle and the transponders. This procedure is called the *fish solution* [Atwood 95]. It lets the operator on the ship monitor vehicle progress.

A new navigation algorithm in [Atwood 95] solves the problem of fading or destructive interference of the acoustic pings produced by the transponders. Especially in shallow waters there can be multipath effects; i.e., pings reflected from various surfaces or the sea bottom producing false transit times. The resulting error is typically greater than 10 m. To avoid this, fixes are calculated pairwise from every two transponders. The mean of all these fixes is calculated and the worst position is eliminated. This process is repeated until either the variance of the fixes is below a predefined threshold (e.g., 5 m), or, if the threshold is not reached with two remaining fixes, the current dead reckoning position is used as the fix. This procedure aims to imitate a human navigator who accepts an automatically computed position when it seems good, or rejects it and prefers his dead reckoning when it does not. The authors also conducted successful experiments with multiple vehicles, where it is important to synchronize the times at which each vehicle triggers its interrogations. One vehicle is designated master, the others are slaves. When the

master initiates the interrogation cycle with a ping, the other vehicles also receive the ping and wait a preset period of time before they start their interrogation cycle. The more vehicles there are, the fewer times each can update its position. With this procedure, the master can calculate the position of all the slaves.

[Carof 94] describes an acoustic system for long range navigation and guidance. He points out that long range acoustic navigation is limited by the slow speed of acoustic waves in the water. Passive-only acoustic systems normally need at least three external references. With less references available, it is necessary to use additional means such as vehicle motion measurement for a solution (Target Motion Analysis). For two available transponders, the author proposes a system that uses differential signal delay for ranging, plus Doppler between the received signals.

[Austin 94] chose ultra-short baseline navigation and spread spectrum signals for an ROV to overcome the long-range problems of acoustic navigation. Spread spectrum signals make it possible to lengthen the signal and to achieve a longer range without loss of accuracy. [Austin 94] especially refers to the benefits of signal encoding, which enhances reception and also makes it difficult to detect the signal without the knowledge of the code. A disadvantage, however, is the increased complexity of the signal which requires more complex processing, be it by software or by hardware.

[Vaganay 95] also use an ultra-short baseline system. They combine dead reckoning by an attitude and heading reference system (AHRS) or an inertial navigation unit, both based on gyros, with an electromagnetic log, a Doppler log and a depth cell with acoustic techniques to calculate an estimated position in a Kalman filter. One or two acoustic

beacons are deployed, floating in the ocean. The vehicle navigates autonomously as long as possible and steers back into the beacon range for position updates. As the beacons are floating freely, they are equipped with GPS receivers to provide the vehicle with their respective positions via acoustic modem. The authors adopt a three level software architecture similar to that described in [Byrnes 93] and used in the NPS Phoenix AUV [Healey 94].

[Rendas 94] combines long-baseline navigation with dead reckoning and calls it a *hybrid* system. The vehicle travels between deployed baseline arrays, each consisting, for example, of four transponders, and uses acoustic navigation when in range of an array. Outside the range, it uses a sonar/Doppler sensor and depth information for autonomous navigation. The distances between the arrays must be carefully planned, because the accuracy of navigation in the autonomous mode deteriorates with time, depending on the quality of the sensing systems. The transition from one mode to another takes place automatically. When the vehicle does not receive a sufficient number of range measurements from the transponders, it switches to autonomous mode and stops sending interrogation signals until it determines, based on estimated position and error prediction, that it is close enough to another baseline array.

The above system uses a variable dimension Kalman filter for both navigation modes. When there is no detectable acceleration, the filter assumes uniform motion and estimates position and linear velocity. This is called the *quiescent* model. When there is acceleration, the filter switches to a larger order (maneuvering model) and extends its state vector to include the accelerations.

[Smith 95] propose a solution where active transponders are impracticable, as, for example, in the vicinity of man-made underwater structures. This system uses several sonar sensors to track previously known features which are stored in an on-board map. Additionally, it tries to locate and classify further unknown features, measure their position and add them to the on-board map to provide a relative position to the underwater structure. The vehicle attitude information, necessary for three-dimensional operations, is supplied by two inclinometers for pitch and roll, and a compass. Sonar and inertial information are combined in an extended Kalman filter, resulting in a position estimate.

Using low-frequency sonar sensors, it is difficult to achieve exact navigation, because most solid objects will produce mirror-like returns. In the above work, to obtain more accurate data, three transducers are mounted closely spaced, co-axially on one pan-and-tilt unit to measure the time-of-flight of a sound pulse. Furthermore, a sonar model was developed that makes it possible to distinguish the form of the sonar target. The system tries to deduce whether it is scanning a plane, a sphere, a cylinder, an edge, a corner, or a point.

3. Other Systems

Another method proposed by [Scherbatyuk 95] is the use of relief data; i.e., the update of AUV position by comparing sensed terrain features or depth isolines with a digital map on board the AUV. This kind of navigation has been known to navigators for centuries, especially in connection with the analysis of ground samples taken from the sea floor by lead line. However, it can hardly be used for high precision navigation. Often, the sea bottom does not contain sufficiently distinguishable features. This may become a problem in the shallow water of a sandy coast.

Finally [Scherbatyuk 95] proposes a hybrid video and sonar based navigation system. The position can be updated using three different methods. First, position can be updated by measuring linear and angular velocities from real video images. Second, artificial or natural underwater features (like a pipeline, for example) can be used together with a side scan sonar for position fixing, or, third, markers can be deployed in the sea area at known positions to be used by video or sonar.

[Balasuriya 95] also proposes a vision based system. Pointing out acoustic shading and multipath effects, especially in underwater structures, as disadvantages of acoustic systems, he uses a vision based system to take advantage of the stability of speed of light in changing water densities. Furthermore, it is not necessary to add energy to the environment, and the information density of a video picture is very high as is the achievable resolution. However, in this work it is assumed that the target area is well lit and that the target consists of line features. Light patterns produced by the target and light intensity are used to control the behavior of the vehicle.

A very interesting but also very specialized system is introduced by [Consi 94]. It uses chemical sensing to localize a source of chemical discharge. Many animals use this technique to find food, mates and spawning grounds, or to let them avoid predators. For an AUV, one can imagine applications in scientific, environmental, commercial and defense-related fields. For the marine environment, factors like dispersion of fluids or the choice of a navigation algorithm are of primary concern.

C. SUMMARY

Many approaches to the problem of AUV navigation have been devised and new ones are still emerging. However, it seems that most of the above described proposals can only be used in very specialized applications. Most are also limited either by dependence on previously deployed external means or by some requirement to prepare or manipulate the target. The preferred method of most developers is the acoustic approach.

Although all developers seem to agree on the above stated basic design goals for AUV's, it appears that most described systems have a higher degree of complexity and dependence on external means than the system built by [McGhee 95]. Also, in all other systems, the degree of independence from outward references does not seem sufficient. Of course, an approach like that of NPS would not be suitable for missions under a closed ice field or missions that require deep dives, because the vehicle would not be able to surface periodically to get a GPS fix [Carof 94]. There is also some cost to surfacing. A vehicle on a 1000 km mission at an average depth of 5000 m that has to surface every 50 km for a navigation update uses about 20 % of its resources for this maneuver [Carof 94].

Acoustic navigation requires previously deployed acoustic beacons in the area of operation. Systems based on acoustic navigation do not fulfill the requirements for clandestine operations, because there will be a lot of "noise" in the water. The above mentioned encoding and use of the wide-band signals could be a solution to this problem.

It can be seen that high accuracy and other design goals for an inertial navigation system are achievable. But clearly, the cost increases rapidly with the degree of sophistication and the desired precision. From this point of view, the NPS Phoenix AUV,

described in [Healey 94], together with the navigation system explored in this thesis, seems to provide a very effective approach for achievement of clandestine missions in shallow water by a small AUV.

The rest of this thesis is concerned with verification of the correctness of the current NPS navigation filter [McGhee 95, Bachmann 95], and with experiments aimed at obtaining useful values for the filter gains.

III. DESCRIPTION OF THE SIMULATION

A. INTRODUCTION

The early sea-trials of the SANS small AUV navigation system described in [Bachmann 95] generally produced poor results. The estimated positions computed by the filter were very inconsistent. Several reasons for this are known already. For example, the filter gains were not yet optimized. One of the motivations for the sea-tests was "to obtain data for post-processing to be used in optimizing the Kalman filter gains" [Bachmann 95].

Another reason was that the filter version shown in Figure 1 was suspected to produce instability or underdamping in ocean current estimation. This comes about because, in this version, the apparent current $(\dot{x}_c, \dot{y}_c)$ was added to the velocities through the water $(\dot{x}_w, \dot{y}_w)$. Then the difference of this sum and the estimated north and east velocities in earth coordinates $(\dot{x}_e, \dot{y}_e)$ was taken, multiplied by gain matrix K_3, and added to north and east accelerations in earth coordinates $(\ddot{x}_e, \ddot{y}_e)$ to correct bias errors in these signals. This approach submits the effects of current on velocity over ground to a first order time lag (inversely proportional to K_3), which could cause instability or underdamping. This thesis investigates this issue.

The latest version of the Kalman filter is shown in Figure 2. The main difference between the old and the new version is the handling of the apparent current $(\dot{x}_c, \dot{y}_c)$. In the new version, the apparent current is added directly to the north and east velocities relative to the water $(\dot{x}_w, \dot{y}_w)$. This version is the basis on which this thesis will explore the subject of the choice of values for the gain matrices.

To make the simulation more realistic, the simulated submarine should also be able to roll, pitch and dive. This was not implemented in the first version of the *perfect autopilot* [McGhee 96], and will be done in this thesis. Pitch and roll will also influence the acceleration and angular-rate sensors, and will disturb the position estimation of the Kalman filter.

Sea-trials, as described in [Bachmann 95], are an expensive and time consuming task. A simulation, on the other hand, offers a convenient way to test, verify and fine-tune code "on dry land". Another advantage is the opportunity to test and compare two versions of code, written by different authors, against each other. As one version is written in LISP, it would not be possible to test this code at sea, because LISP uses garbage collection for memory management and therefore is not useable for real-time systems with high sampling rates, as required by the SANS system.

B. OVERVIEW OF THE SIMULATION

The LISP code can be divided into two parts. In the first part, the vehicle is steered over a track, determined by a trajectory list, stored in the global variable **trajectory**. This trajectory list contains elements which are lists themselves, and each of these lists contains five elements with the following meaning. The first is the time in seconds. This determines the time at which the parameters in the list become active. The second element sets the speed of the vehicle, measured in feet per second. The third element sets the heading rate in radians per second. The fourth element determines the depth in feet the vehicle is supposed to dive to, and the fifth element sets the roll rate in radians per second. These data

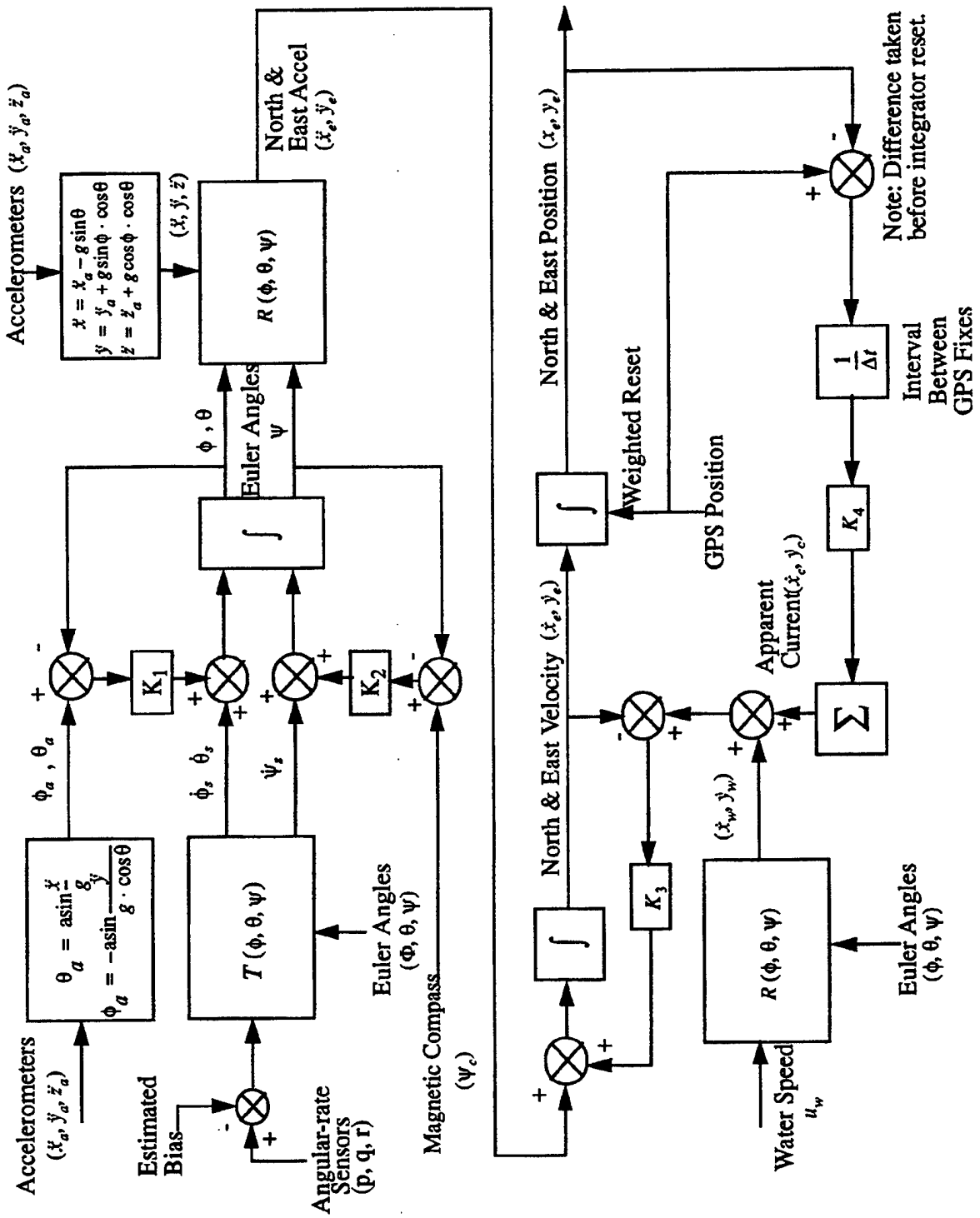


Figure 1: The original version of the navigation filter [Bachmann 95]

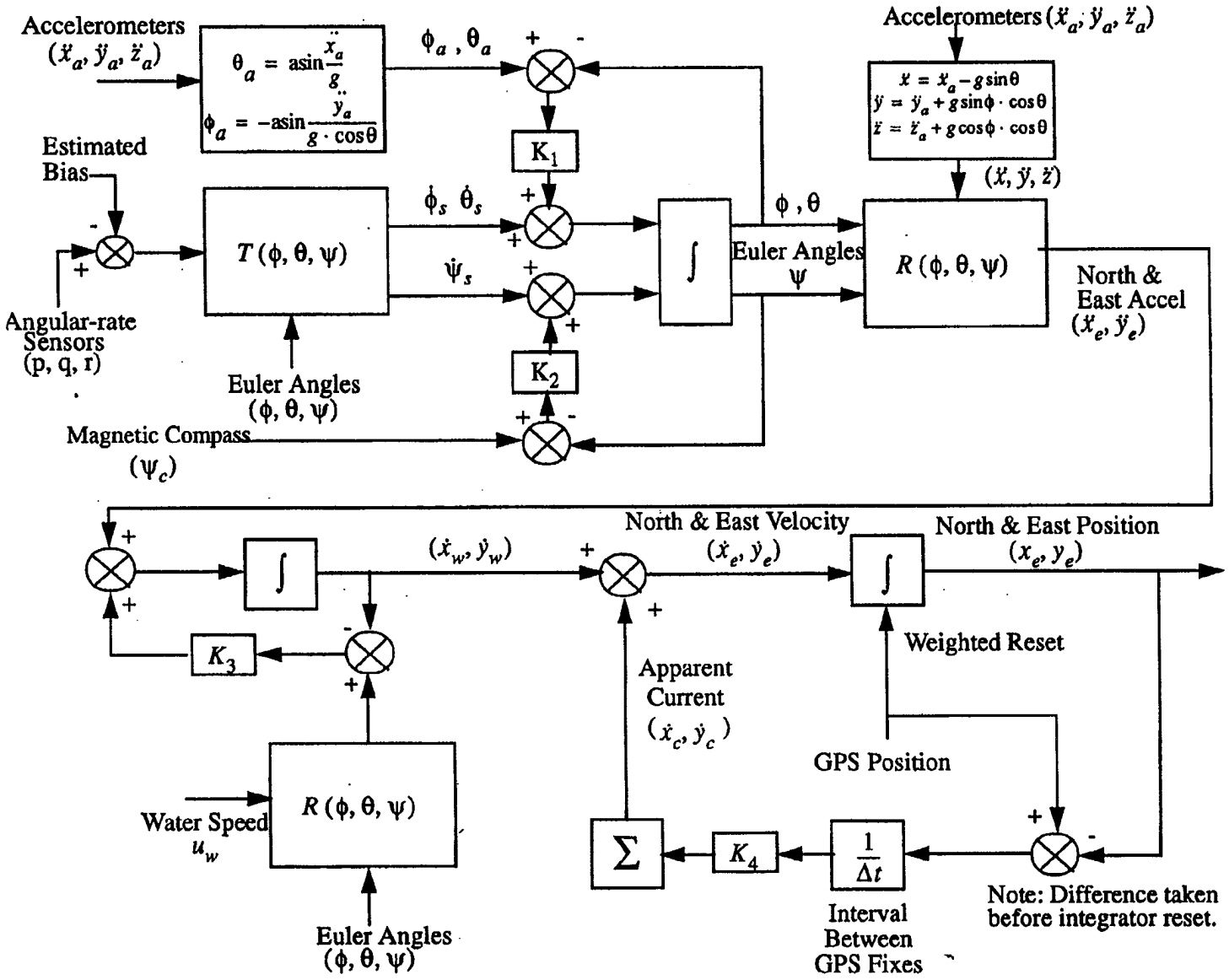


Figure 2: The new version of the navigation filter

are processed in the appropriate methods of the *perfect-autopilot* class and its base class, *rigid-body,* in the files *perfect-autopilot.cl* and *euler-angle-rigid-body.cl.* The updated position (posture) and rate (posture-rate, velocity) information is stored in slots of the *rigid-body* class; name, time and gain values specific to the *perfect autopilot* are stored in the slots of the *perfect-autopilot* class.

The coordinate system used is a three-dimensional Cartesian system with the origin at (0, 0, 0). Northings are measured along the x-axis, eastings along the y-axis, and depth is measured positive downward along the z-axis. All units are English units, distances are defined in feet, angles in radians, rates in feet per second or radians per second. Gravity is measured in feet per second squared.

While the simulated vehicle is traveling, the current time in seconds, three accelerometer readings $(\ddot{x}_a, \ddot{y}_a, \ddot{z}_a)$, three angular rates (p, q, r), heading, waterspeed, and position are recorded every tenth of a second in the global variable **mission-data**. This variable is a list of lists which includes a binary flag to indicate GPS fixes. This is necessary to synchronize position updates in both the LISP and the C++ filter code. Whenever the vehicle surfaces, GPS fixes are performed in an interval of one second. A GPS fix resets the estimated position calculated by the filter to the position recorded during the simulation run. It is also used for the calculation of the apparent current.

When **mission-data** is completed, the recorded accelerations and angular-rates can be fed into the LISP and C++ filter code, and the performance of the filter can be evaluated by comparing actual (simulated) and estimated positions.

C. DESCRIPTION OF CODE AND CODE CHANGES

1. The Perfect Autopilot

The *perfect autopilot* code was originally developed for aircraft simulation [McGhee 96]. The code is written in LISP and defined in the file *perfect-autopilot.cl*. It is used to steer a vehicle over a track defined in the global variable **trajectory**. The adjective *perfect* implies the following assumptions: first, there is no time delay in the steering process, as could be observed in the real world. For example, when the rudder of a boat is set to turn the boat, there will be a time delay between the deflection of the rudder and the beginning of the turn. The *perfect autopilot* changes the vehicle turn rate immediately when a new turn rate command is issued. The same is true for changes in pitch rate and roll rate. On the other hand, a new (longitudinal) speed command is followed with a first order timelag according to the following equation:

$$\dot{u} = K_l \left(u_{commanded} - u_{actual} \right) \tag{1}$$

where K_l is the longitudinal acceleration gain. Second, it is assumed that there is no sideslip and no angle of attack, contrary to the behavior of a submarine. The equations associated with these assumptions are presented in the following paragraphs.

In order to simulate the output of the acceleration sensors, the assumption of a flat, non-rotating earth is used. The following Newton-Euler equations are taken from [McGhee 69]:

$$\dot{u} = vr - wq + \frac{f_x}{m} - g\sin\theta \tag{2}$$

$$\dot{v} = wp - ur + \frac{f_y}{m} + g\cos\theta\sin\phi \tag{3}$$

$$\dot{w} = uq - vp + \frac{f_z}{m} + g\cos\theta\cos\phi \tag{4}$$

here u, v, w are velocities along the vehicle's x, y, and z axes, $\dot{u}$, $\dot{v}$, $\dot{w}$ the respective velocity growth rates, p, q, and r are angular rates, g is gravitational acceleration, and m is the mass of the vehicle. The quantities f_i are components of the vector $\vec{f}$ expressing applied forces in body coordinates. Thus the quotient $\frac{f_i}{m}$, $i \in \{x, y, z\}$, can be seen as the specific force "felt" at the accelerometers. Rearranging the equations (2) - (4) results in

$$\ddot{x}_a = \frac{f_x}{m} = \dot{u} - vr + wq + g\sin\theta \tag{5}$$

$$\ddot{y}_a = \frac{f_y}{m} = \dot{v} - wp + ur - g\cos\theta\sin\phi \tag{6}$$

$$\ddot{z}_a = \frac{f_z}{m} = \dot{w} - uq + vp - g\cos\theta\cos\phi \tag{7}$$

As the assumption is used that the model neglects sideslip and angle of attack,

$$v \equiv 0 \rightarrow \dot{v} = 0 \tag{8}$$

$$w \equiv 0 \rightarrow \dot{w} = 0 \tag{9}$$

so that

$$\ddot{x}_a = \dot{u} + g\sin\theta \tag{10}$$

$$\ddot{y}_a = up - g\cos\theta\sin\phi \tag{11}$$

$$\ddot{z}_a = -up - g\cos\theta\cos\phi \tag{12}$$

Equations (10) - (12) are encoded in the method *accelerometer-output* in the file *perfect-autopilot.cl*. They are different from the equations published in [Bachmann 95], which contained some typographical errors.

Roll and pitch control had to be added to the original code. Roll can be set as a rate in the fifth element of the current trajectory list. As can be seen in Appendix B, file *perfect-autopilot.cl*, the value is assigned directly to the fourth element of the *rigid body* velocity vector in the method *commanded-velocity*. Depth can be set as the fourth element of the current trajectory list. This value is passed to the method *desired-dive-angle*, listed in Appendix B. The dive angle is computed based on the difference between the commanded

depth and the current depth, stored in the *rigid body* posture vector, multiplied by a depth error gain. This is shown in equation (13):

$$\theta_{dive} = K_{dive} \left(z_{commanded} - z_{actual} \right) \tag{13}$$

Thus, the larger the difference in commanded and actual depth, the steeper will be the desired dive angle. The computed dive angle is passed to the method *dive-angle-rate*. The dive angle rate depends on the difference between the desired dive angle and the current dive angle, multiplied by a dive-angle-error-gain, as shown in equation (14):

$$\dot{\theta}_{dive} = -p = K_{rate} \left(\theta_{commanded} - \theta_{actual} \right) \tag{14}$$

The rate of change will thus be large when the difference is large, and small when the difference is small. The computed dive-angle-rate is passed to the fifth element of the velocity vector in the method *commanded-velocity*. Both above mentioned gain values must be fine-tuned carefully to ensure a smooth ride; i.e., to minimize depth over- or undershoot.

2. The Navigation Filter

The first part of the navigation filter, that is, the part in the upper left corner of Figure 2, converts the accelerations measured by the acceleration sensors into a vehicle attitude and integrates them with angular rates measured by the angular-rate sensors. The equations

$$\theta_a = asin \frac{\ddot{x}_a}{g} \tag{15}$$

$$\phi_a = -asin \frac{\ddot{y}_a}{g \cdot \cos\theta} \tag{16}$$

can be derived from equations (9) and (10), considering that $\dot{u}$, $\dot{v}$, $\dot{w}$ and u, v, w are not available in the filter of Figure 2, but that $\dot{u}$, $\dot{v}$, $\dot{w}$ must have an average value of 0 if the mean value of u, v, w is bounded [McGhee 96]. "Likewise, except for continuous turning, the components of angular rate, (p, q, r), must average to zero." [McGhee 96]. The third component of the attitude vector, ψ_c, is measured directly by a magnetic compass.

The purpose of the transformation matrix $T(\phi, \theta, \psi)$ is to convert the angular rates (p, q, r), which are measured in body coordinates, into Euler rates. The T-matrix is derived from the following equations published in [McGhee 93]:

$$\dot{\phi} = p + q\sin\phi\tan\theta + r\cos\phi\tan\theta \tag{17}$$

$$\dot{\theta} = q\cos\phi - r\sin\phi \tag{18}$$

$$\dot{\psi} = q\sin\phi\sec\theta + r\cos\phi\sec\theta \tag{19}$$

These equations are implemented in the form

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{bmatrix} \begin{bmatrix} P \\ Q \\ R \end{bmatrix} \tag{20}$$

The above matrix-multiplication is performed in the method *angle-derivatives* in the file *navigation-filter.cl* , and the T-matrix is computed in the function *body-rate-to-euler-rate-matrix* in the file *robot-kinematics.cl.*

A rotation matrix describes the orientation of a coordinate system attached to a body relative to a reference system [Craig 89]. The rotation matrices in the navigation filter make it possible to convert the sensed accelerations $(\ddot{x},\ \ddot{y},\ \ddot{z})$ and velocity u_w from body coordinates into earth coordinates. The function *rotation-matrix* is defined in the file *robot-kinematics.cl.*

To change the navigation filter into the old version shown in Figure 1, the methods *velocity-derivatives* and *update-list* had to be changed. In the last four lines of *velocity-derivatives* the estimated current had to be added to read

```
(list (+ (first linear-acceleration)
      (* k3 (- (+ (first water-relative-velocity)
                  (estimated-north-current filter))xdote)))
   (+ (second linear-acceleration)
      (* k3 (- (+ (second water-relative-velocity)
                  (estimated-east-current filter))ydote))))))
```

and in the last four lines of *update-list* the estimated current had to be removed to read

```
(list (* delta-t (estimated-north-velocity filter)))
(list (* delta-t (estimated-east-velocity filter))))))
```

In the C++ code the following lines in the file *postins.cpp* had to be changed, the changes being indicated in italics: In the first two lines, the current had to be added, and in the next two lines the originally added current had to be removed to read

```
//Subtract out previous velocity and apply statistical gain
waterSpeedCorrection[0] =
        Kthree1 * (waterSpeedCorrection[0] - velocities[0] + current[0]);
waterSpeedCorrection[1] =
        Kthree2 * (waterSpeedCorrection[1] - velocities[1] + current[1]);
```

and

```
//Integrate velocities to obtain posture
posture[0] += velocities[0] * deltaT;
posture[1] += velocities[1] * deltaT;
posture[2] += velocities[2] * deltaT;
```

3. The Rigid Body

The *rigid-body* class is the base class from which the simulated submarine is derived. It provides slots to store the posture, posture change rates, velocities, and velocity change rates as well as slots for the physical properties of a body like moments of inertia, forces, torques, mass, and coordinates for a wire frame model of the body. The associated methods provide for moves of the body and related updates of the above mentioned slots. The code is based on the assumptions of a flat earth that does not rotate, and on which current or wind have a constant linear velocity.

D. DESCRIPTION OF THE TEST PROCEDURE

The test mission is shown graphically in Appendix A, Figure A-1 to A-4 (track, depth profile and roll attitude). It is defined in the global variable **trajectory** in the file *perfect-autopilot.cl*. The vehicle is initially positioned at coordinate (0, 0), heading north. The speed for the whole mission is set to 3 feet per second (fps). This speed will be reached with a first order time lag. After one second the vehicle starts a right turn at a turn rate of 0.1 radians per second. The turn is finished after 11 seconds. During this first phase GPS fixes are performed at an interval of 1 second. After 11 seconds the vehicle commences a dive to a depth of 0.5 feet. At time 41 seconds, a right roll to an angle of .18 radians is initiated, and at 43 seconds the roll rate is reversed until the vehicle is straight and level again at 45 seconds. At 90 seconds the vehicle surfaces another 30 seconds for GPS fixes, and after that, at time 120 seconds, it dives again to 0.5 feet. At 150 seconds, a left turn at a rate of -0.1 radians per second is initiated, which is completed at 170 seconds. Between 190 and 210 seconds the vehicle surfaces again for GPS fixing, after which it dives to 0.5 feet. At 230 seconds, a left roll to -0.18 radians is initiated, the roll is reversed at 232 seconds to reestablish a straight and level attitude. At 240 seconds the vehicle starts a climb to the surface and takes GPS fixes until the next dive to 0.5 feet at 260 seconds. It surfaces for the last time at 290 seconds, and the mission is completed at 300 seconds.

Two types of test runs are performed. For the first type, the water current is set to 0.5 fps north and 1.0 fps east. This seems to be a realistic assumption for a calm sea state. For the second mission type the current is set to 0.5 fps north and 5.0 fps east. This can be

seen as challenge for a navigation system in a vehicle travelling at 3.0 fps, because there will be a lot of drift.

To add more realism to the simulation, additional tests were performed with noise added to the acceleration signals, angular-rate signals and the compass in the following way: A function *add-noise-to-mission-data* was added to the file *perfect-autopilot.cl.* This function postprocesses the global variable **mission-data** and adds noise to the values of $\ddot{x}_a$, $\ddot{y}_a$, $\ddot{z}_a$, p, q, r and ψ_c. The magnitude of the noise is set to a random number[4] in the range of about +/-1% of the full scale output of the accelerometers and +/-0.5% of the full scale output of the angular rate sensors. That is, +/-0.3 for the horizontal accelerometers (1% of 1g, or 32.2185 feet per sec^2), +/-0.6 for the vertical accelerometer (1% of 2g), and +/-0.005 for the angular rate sensors (0.5% of 1 radian per sec.). The compass was set to a precision of +/- 2 degrees.

E. SUMMARY

Inconsistent results in earlier trials made it necessary to conduct an experimental evaluation both of the old and new filter versions. To make the simulation more realistic and to correct errors, several code changes had to be made in the files *perfect-autopilot.cl* and *navigation-filter.cl.* In an experimentation it is advantageous to have two versions of code in different programming languages. The results can be compared, and when they agree, one can have increased confidence that the desired algorithms have been correctly coded.

---

[4]More precisely, LISP generates pseudo random numbers. Every time the LISP environment is started it creates the same list of random numbers.

The general course of the simulation is to record data from a known test run and feed these data into the navigation filter. Real and estimated positions can be compared and evaluated.

The *rigid body* in the file *euler-angle-rigid-body.cl* is the base class from which the simulated submarine is derived. The *perfect autopilot* class in the file *perfect-autopilot.cl* steers the submarine over the test track. The navigation filter class in the file *navigation-filter.cl* estimates positions on an input of measured accelerations, angular rates, magnetic heading and speed through the water $(\ddot{x}_a,\ \ddot{y}_a,\ \ddot{z}_a,\ p,\ q,\ r,\ \psi_c,\ u_w)$, or resets positions using simulated GPS fixes respectively. The file *robot-kinematics.cl* provides a collection of functions for matrix and vector computations.

Finally, two test missions with different water current settings and several test runs with and without noise are described. The resulting trajectories are shown graphically in Appendix A.

IV. EXPERIMENTAL RESULTS

A. INTRODUCTION

This chapter describes results of the tests performed with different settings of perfect autopilot and navigation filter parameters. First, values had to be found to provide a smooth ride of the simulated submarine concerning the dive behavior and stabilization on commanded depths. Then, the performance of the navigation filter in both, LISP and C++, is compared and evaluated under diverse environment conditions. The main subject is to research the filter behavior with different settings of the gain matrices K_1 to K_4. The results are plotted graphically in Appendix A and discussed in Section III of this chapter.

The goals of the tests are to find out whether the filter code is correct, whether the original version of the navigation filter is unstable, and to get to conclusions about the filter response to different gain matrix settings. The question is also, whether there are optimal values for these matrices.

B. TEST RESULTS

After adding pitch and roll control in the *perfect autopilot,* a first consideration had to be given to the tuning of the gain values *dive-angle-error-gain* and *depth-error-gain.* A setting of .5 for *dive-angle-error-gain* and -.17 for *depth-error-gain* resulted in a smooth transition to the new depth without any over- or undershoot at a speed of 1 foot per second (fps). However, this proved inadequate for a speed of 3 fps due to a perceptible depth over- and undershoot. A setting of .8 and -.08 for the respective gain values rendered approximately the same, smooth result as in the 1 fps case.

The comparison of the filter performance in LISP and C++ showed only a hardly perceptible difference in the resulting track on the GNUplot (compare the C++ plots and the LISP plots in Appendix A). As this difference appears to be constant over all test runs, it can probably be attributed to floating point precision. A difference could be observed, however, between the new and the original version of the code. Comparing the width of the steps that occur when the system updates its position by GPS fix, the new version seems always to be a little more accurate than the old version (compare Figure A-5 to A-7).

Changing the values of the gain matrices showed the following filter behavior: Assigning the value 0.1 to K_3 instead of the preset value 0.5 in the new filter version seems to have a small, although perceptible effect (Figure A-8 and A-10). The old version, however, showed a performance that could be interpreted as underdamping, with consequent overshoot. The estimated position at GPS fix time was quite far off, and the plotted track, on the other plots always an almost straight line, showed some bending (Figure A-11 to A-14). At $K_3 = 0.7$, the effect is gone (Figure A-15 to A-18) . Changing the value of K_4 had no perceptible effect in both filter versions (Figure A-19 to A- 22).

Assigning different values to K_1 and K_2 did not result in perceptibly different results (Figure A-23 to A-26). It is interesting, however, that both versions of the filter showed the best performance at a setting of 0.0 for both K_1 and K_2 (Figure A-5 to A-7). Increasing this value to 0.1 led to a significantly worse performance (Figure A-27 to A-30). Adding noise to the signals, as described above in Section III, Subsection D, led to a profound

deterioration in performance[5] with K_1 and K_2 set to 0.0 (Figure A-31 and A-36). Setting K_1 and K_2 to 0.1 improved the performance visibly (Figure A-32 and A-37). Values beyond 0.1 showed deterioration again until the performance approximately stabilized for values between 0.5 and 1.0 (Figure A-33 to A-35 and Figure A-38 to A-40).

C. DISCUSSION OF THE TEST RESULTS

The tests showed a similar behavior of the navigation filters written in LISP and in C++. Both filter versions return acceptable results. The new version of the filter is better than the original version, especially when K_3 is set to a low value (compare Figure A-5 to A-7 and Figure A-8 to A-14). With K_3 set to a high value, 0.7, the performance is quite similar (Figure A-15 to A-18). This is, because "if k is a component of a diagonal gain matrix K, then the time constant of the corresponding closed loop state estimation filter is the reciprocal of k. That is, $\tau = \frac{1}{k}$" [McGhee 95]. With K_3 set to 0.1, current estimates are subject to a first order filtering effect with a time constant of 10 seconds, while $K_3 = 0.7$ corresponds to a time constant of only 1.4 seconds. So the shorter delay provides a better result. Low values of K_4 do not show any effect (Figure A-19 to A-22). This does not seem logical. The apparent current, based on the precise GPS system of the simulation, should also be very precise, and thus very important for the quality of the filter output. However, the submarine is surfaced for the first 11 seconds of the simulation and is performing GPS fixes once a second. The lack of effect of changes in K_4 can be attributed to the fact that, with no sensor noise, the filter is able to get a good current estimate in this period. With

[5]As it can be concluded at this point that the new filter version shows a better behavior than the original version, only the new version will undergo the additional tests.

a setting of 0.5 for K_4, for example, and no noise, half of the apparent current would be added as a correction in the first iteration of the filter (this is the meaning of the summer $\sum$ in Figure 2). In the second iteration, half of the remaining half would be added, and so on. Generally, after the first iteration, a fraction K_4 of the current is in the summer. After the second iteration, it is $K_4 + K_4 (1 - K_4)$, and so on. So even if K_4 is set to 0.1, a surface period of 11 seconds should be sufficient to compute a good current estimate.

Noise clearly proved to be the important factor of filter performance. First, without noise, for best results, K_1 and K_2 had to be set to 0.0, which means that the acceleration sensors had to be ignored completely. Any increase in K_1 and K_2 deteriorated the results (compare Figure A-5 to A-7 and Figure A-27 to A-30). In the absence of noise, the accelerometers seem to "misinterpret" attitude changes as a combination of acceleration and gravity changes. Allowing this input by setting K_1 to a value higher than 0.0 permits wrong data to enter the system and deteriorates the output in the absence of noise. However, when noise was added to the acceleration sensors, angular-rate sensors and the compass, a setting of 0.0 for K_1 and K_2 was not sufficient any more. A value of 0.1 gave the best result, decreasing the position error at the first GPS fix period from about 50 feet to about 20 feet, and at the second from about 30 feet to about 10 feet. A further increase of the value of K_1 and K_2 between 0.2 and 1.0 approximately stabilized the error at about 25 feet at the first GPS fix period, and about 10 feet at the second (Figure A-31 to A-40). Clearly, care must be taken when values for the gain matrices are selected.

The intensity of the actual current did not seem to have an influence on the performance of the navigation filter. This can be seen when comparing Figure A-8 and A-9,

with Figure A-11 and A-12. For this reason no further plots of the high current case are added to Appendix A. This result was not anticipated, because the environment (the sea state, for example) can be interpreted as a kind of noise, and a significant change in the strength of the current was expected to have some influence. However, the artificial environment of a simulation with the lack of errors in water speed sensing or GPS may be the cause for this outcome. Adding noise to these sensors should be subject of further research.

The test results show that the code for the navigation filter seems to be correct. Also, the original version of the filter does not seem to be unstable. It is only less accurate. Furthermore, there seem to exist optimal values for the gain matrices. However, these values can not be found in the artificial environment of a simulation. They depend on the noise characteristics, and these are more complex in the real world than can be generated in a simulation, unless a large study is conducted to accurately characterize measurement errors and environmental disturbances.

D. SUMMARY

Extensive experiments were performed to test the navigation filter code for correctness and stability, and to learn about the filter response to changes in gain matrix values. The correctness can be deduced from the similar behavior of the filter in both, the LISP and the C++ version of the code. The original version of the filter also proved to be stable within the range of the investigated parameters. However, the new version of the filter was more accurate than the original version.

Most emphasis was given to the examination of the influence of the gain matrix values on filter behavior. Noise played a major part in filter response. In the absence of noise, K_1 and K_2 had to be set to 0.0 to achieve the best results. With noise, the setting of the gain matrices is dependent on the noise level. Filter response to changes in K_3 and K_4 were minimal. This was attributed to the missing noise on the water speed sensor and the GPS. However, it seems clear that there are optimal values for the gain matrices. As the values for the best result in the case with added noise, 0.1 for K_1 and K_2, 0.5 for K_3, and 0.7 for K_4, are significantly different from the values used in [Bachmann 95], it can be concluded that only tests in the "real world" will reveal true optimal values for the real AUV navigation system.

V. SUMMARY AND CONCLUSIONS

A. CONCLUSIONS

The goals of this thesis were to complete the AUV simulation by adding roll and pitch control to the *perfect autopilot,* to verify the correctness of the navigation filter code and to evaluate it, to find suitable values for the gain matrices on the basis of artificially generated data, and to test the original version of the filter code for instability.

Both versions of the filter code in LISP and in C++ returned consistently good results within the range of the investigated parameters. For this reason, the filter code can be assumed to be a correct encoding of the desired algorithm. The original version of the code did not show instability; however, it proved to be less accurate than the new version.

The tests showed that there are optimal values for the gain matrices, but the values are dependent on the noise level added to the sensors. The noise characteristics of the "real world" for the sensors are unknown at present. However, the magnetic compass used in the sea trials described in [Bachmann 95], for example, is very undependable in the towfish environment. It can show fluctuations up to +/-10 degrees, is restricted in roll, and is also influenced by magnetic and electrical fields in the AUV. A "paddle wheel" water speed sensor must be carefully calibrated and can not be considered a precision instrument. The artificially generated noise in the range of +/-1% of the full scale output of the accelerometers, +/-0.5% of the full scale output of the angular rate sensors, and +/-2 degrees for the compass, is completely arbitrary as is the use of uniformly distributed and

independent errors. Sea state (wave action and water current) will also have a significant impact. Only tests in the "real world" will provide correct noise levels for which optimal gain values can be found.

B. RECOMMENDATIONS FOR FUTURE WORK

As noise is obviously a very important parameter to be considered when optimal gain values are desired, there is much room for research in this area. A first step would be the generation of noise for the water speed sensor and the GPS. It can be expected that the values used in this thesis for the respective gain matrices will come out not to be optimal ones, first, because they were chosen deliberately, and second, the results did not change much when they were changed. Further consideration should also be given to the GPS system. One tends to overestimate the precision and the reliability of this system, especially the differential GPS, and thus choose the gain for K_4 too high. But it is by no means certain that even the DGPS yields data accurate enough for the purposes of an AUV mission. The signals may be available only intermittently, for example due to shading by structures between the differential transmitter and the vehicle antenna or wave action leaving the antenna covered with water.

As it was discovered in the *perfect autopilot* that the appropriate gain values that govern the smooth transition to a commanded depth were dependent on the speed of the vehicle, a similar dependence can be suspected for the gain matrices of the navigation filter. This means that further research can be done in the area of a dynamic selection of gain matrix values. For example, gain values could be adjusted according to the speed of the vehicle, on the grounds of environmental changes like wave action, and they could be

changed based on experience the system deduces from the observed accuracy of previous GPS fixes. Of course, the more variables are introduced into the system, the more complex the system gets.

At the time of the completion of this work, new sea trials with a towfish [Bachmann 95] are being performed. The acceleration, angular-rate, water speed, and compass data recorded during these trials can be fed into the navigation filter and postprocessed. This will give further clues concerning the optimization of the values for the gain matrices. Much additional work of this sort needs to be accomplished before the performance limits of the SANS system concept can be fairly evaluated.

APPENDIX A: PLOTS OF THE TEST MISSIONS

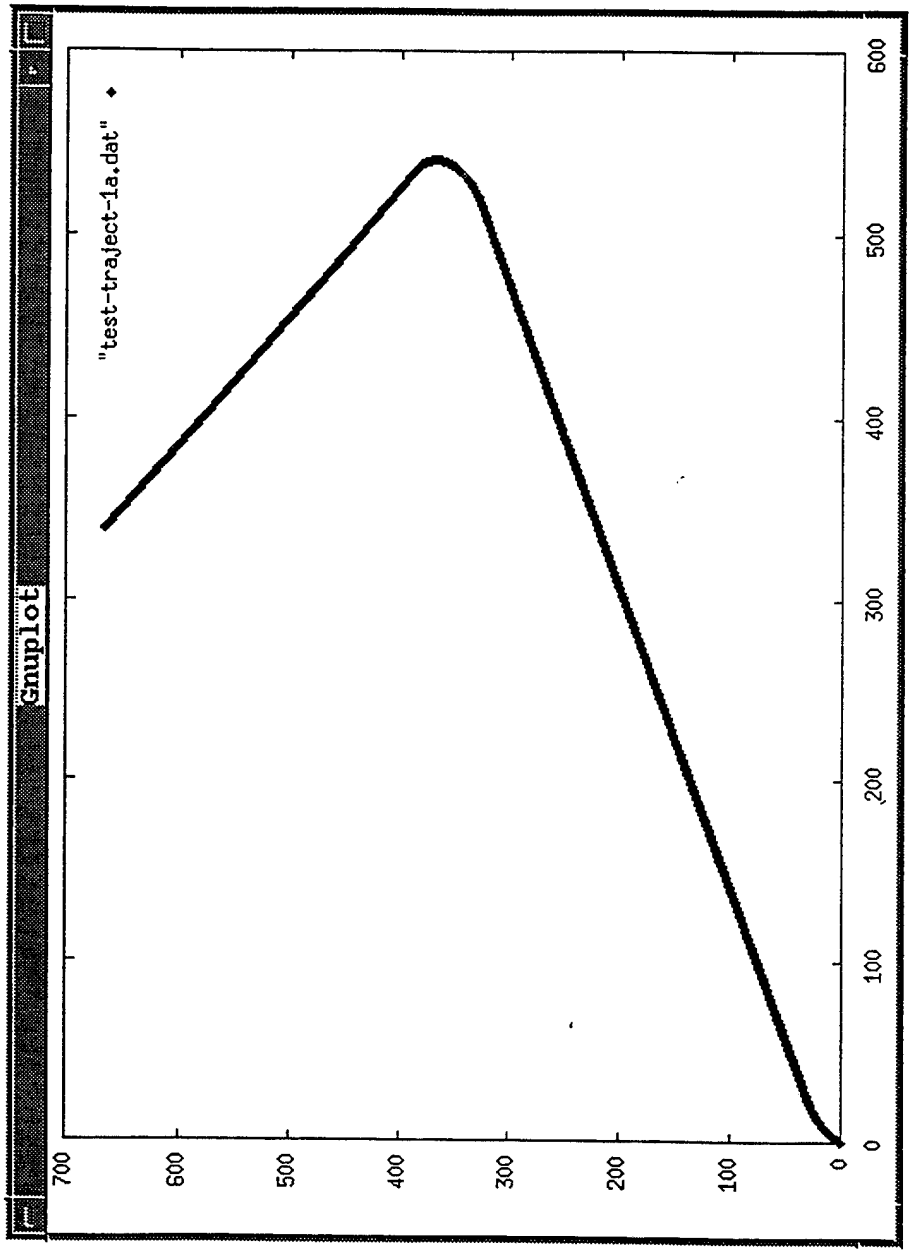


Figure A-1: Plot of test mission, current set to 0.5 north, 1.0 east

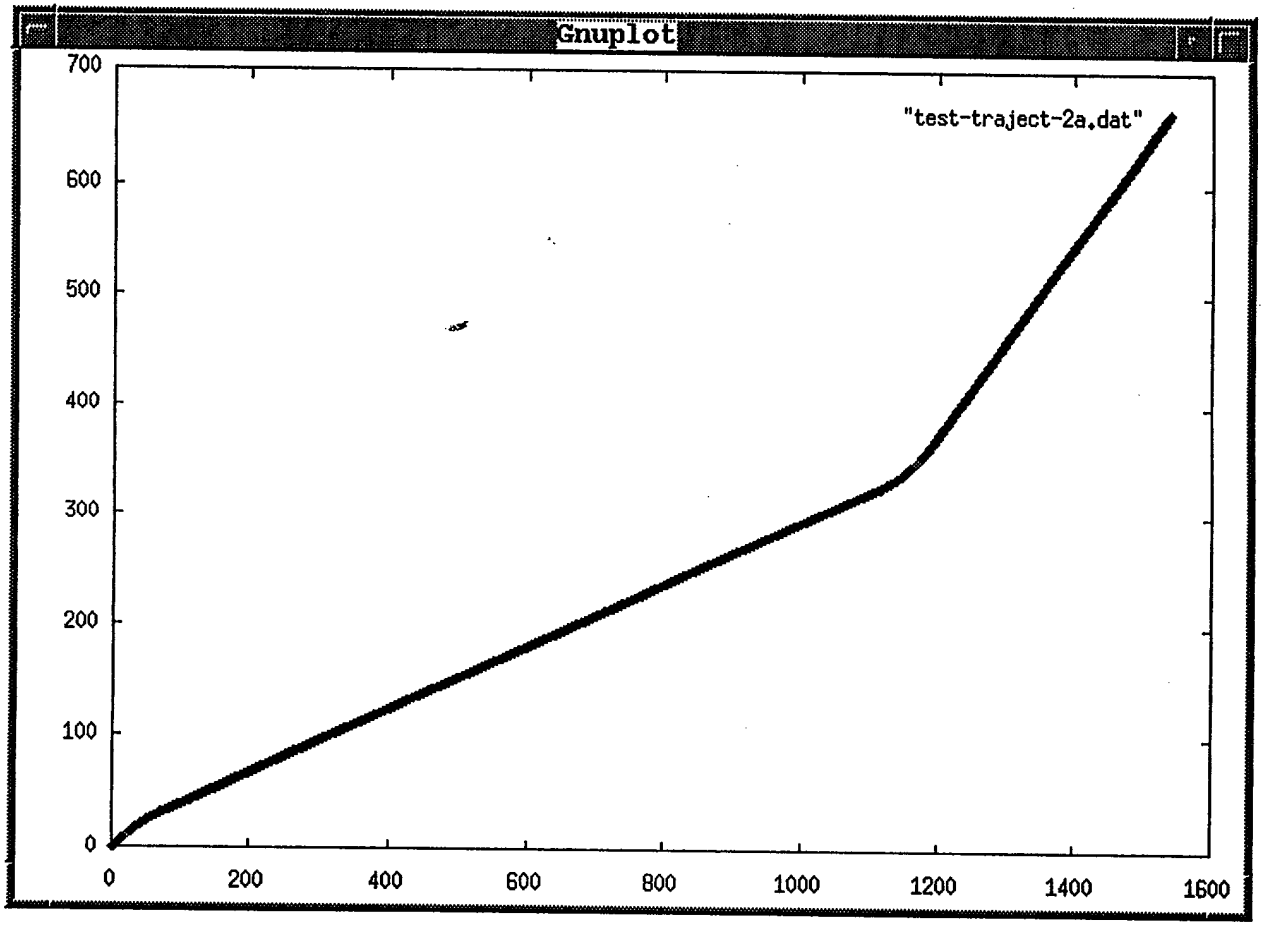


Figure A-2: Plot of test mission, current set to 0.5 north, 5.0 east

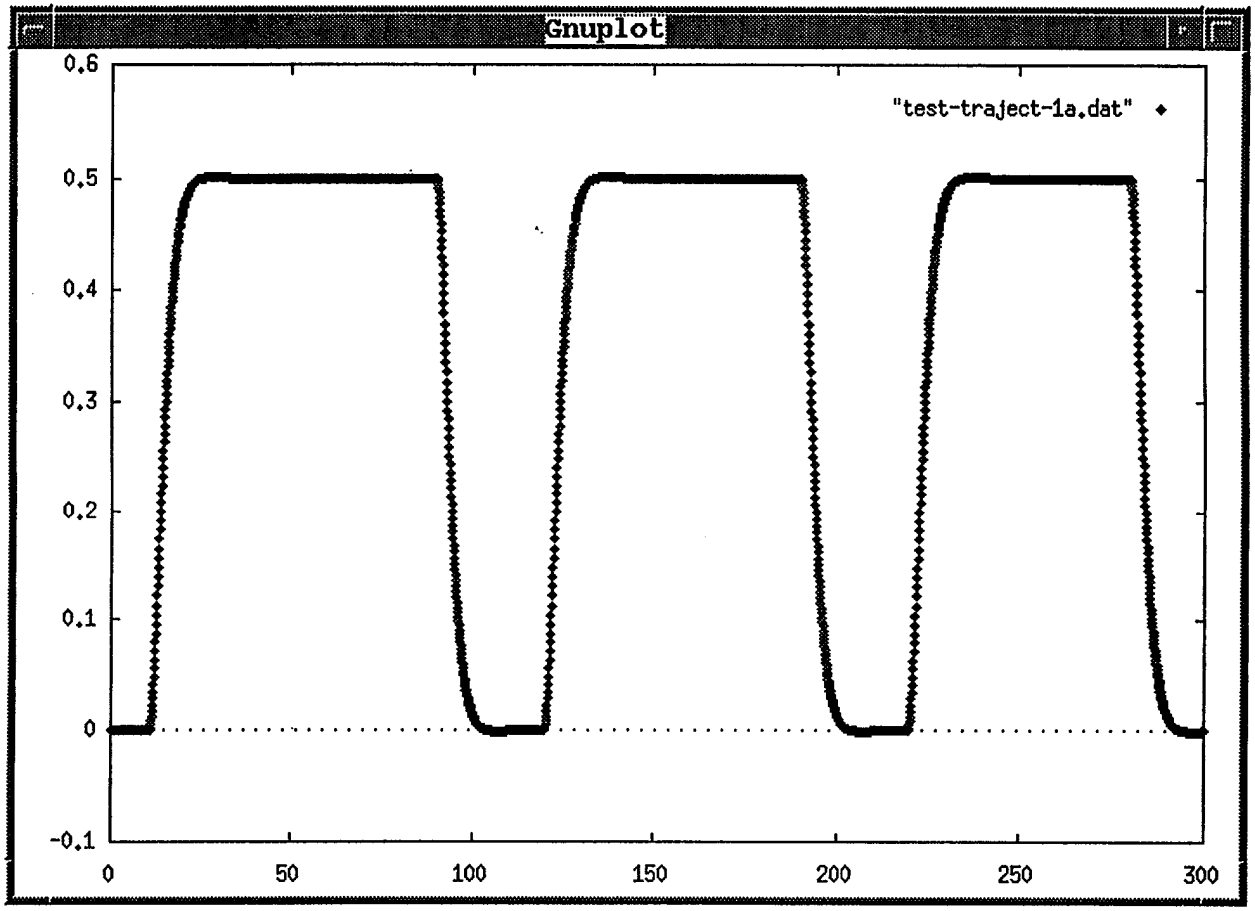


Figure A-3: Depth profile of test mission, depth plotted as positive number

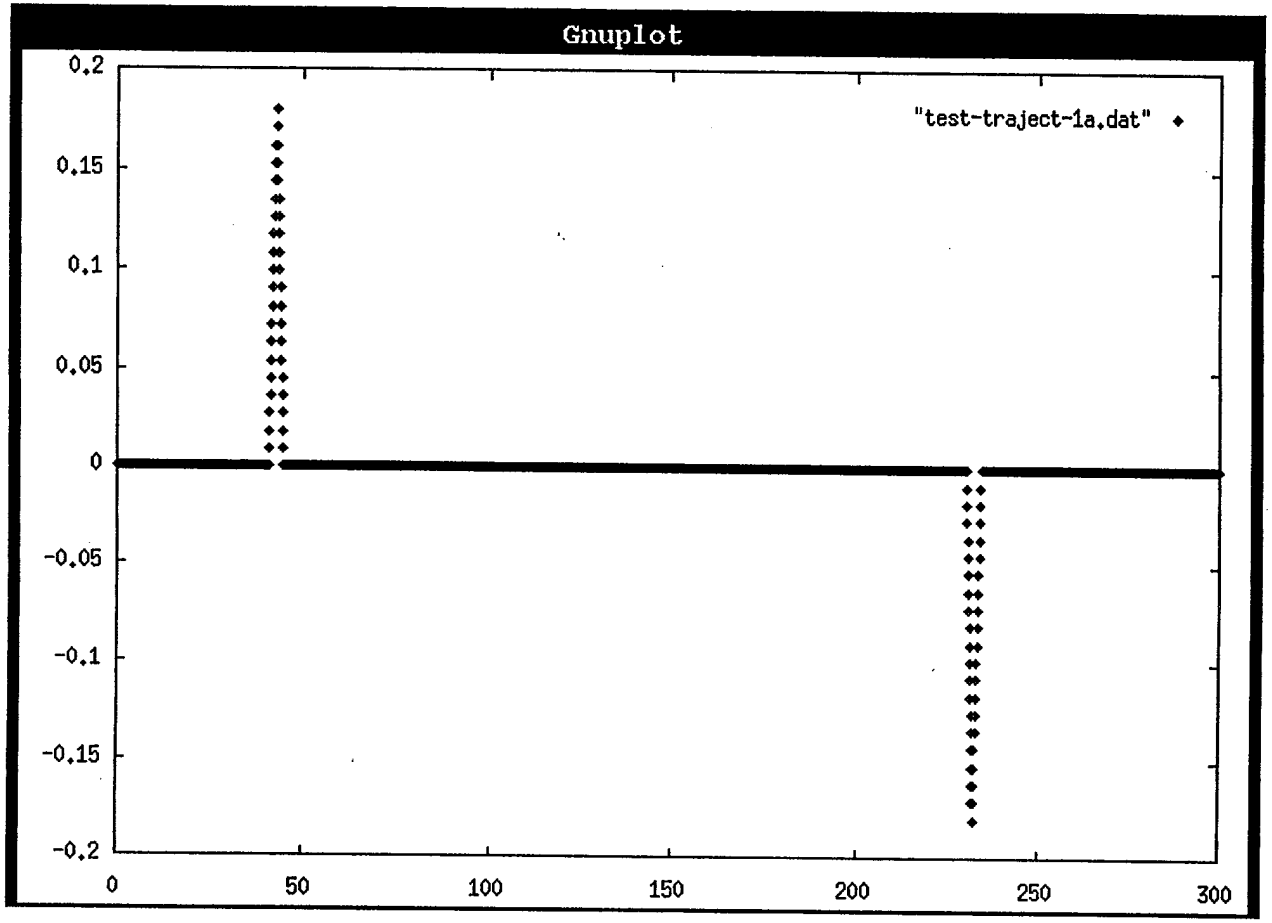


Figure A-4: Plot of the test mission, roll attitude, angle in radians, time in seconds

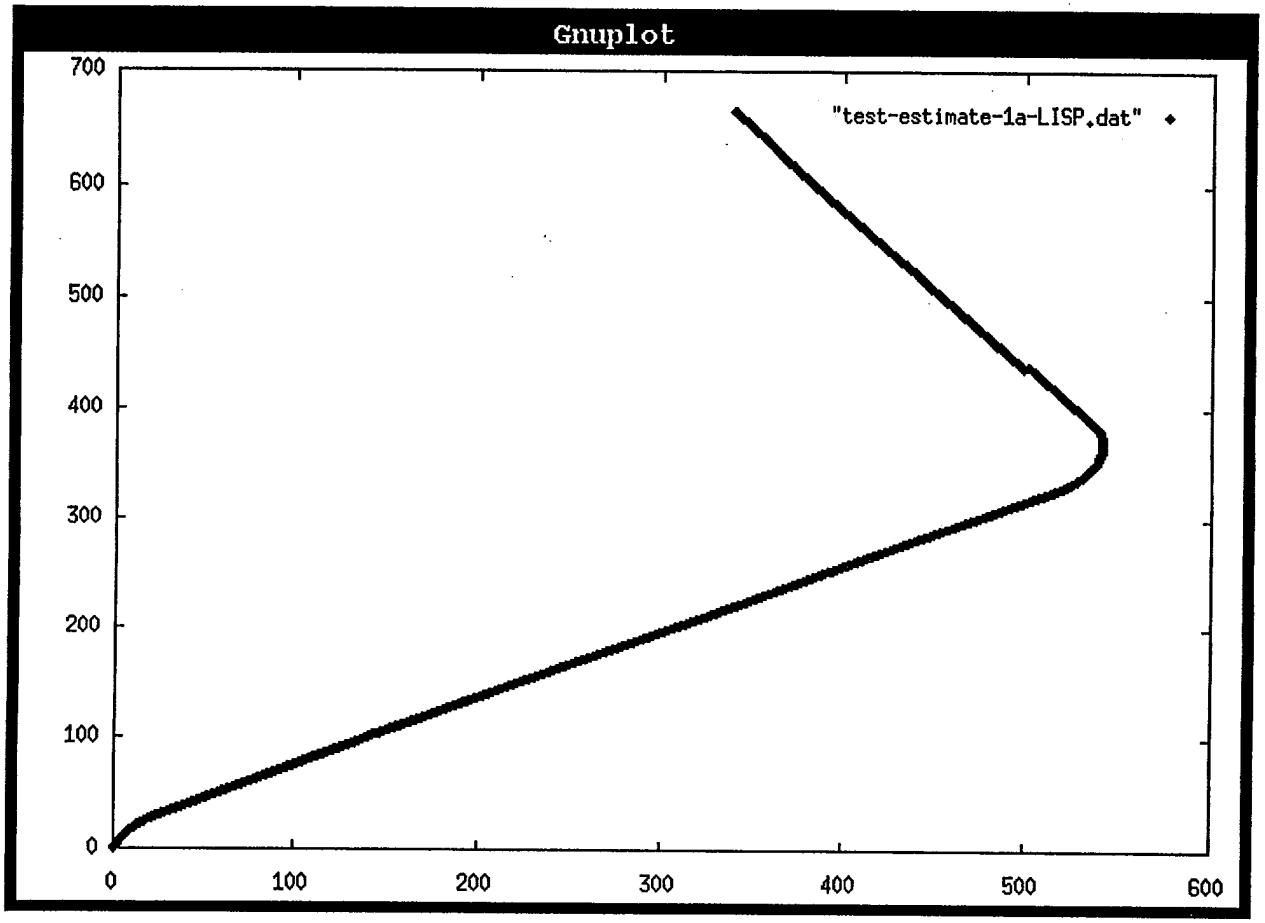


Figure A-5: New filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.5, K4: 0.7

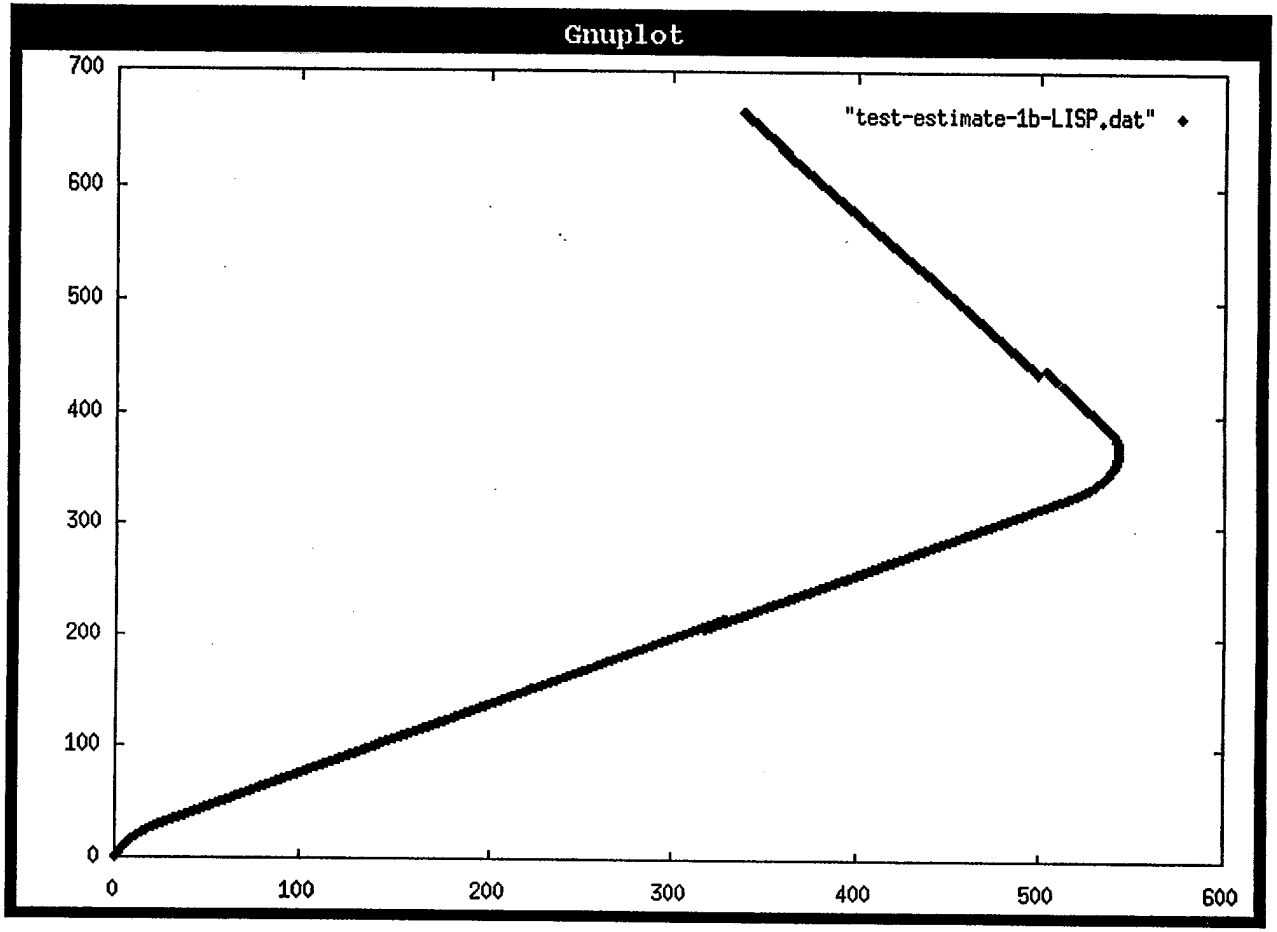


Figure A-6: Original filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.5; K4: 0.7

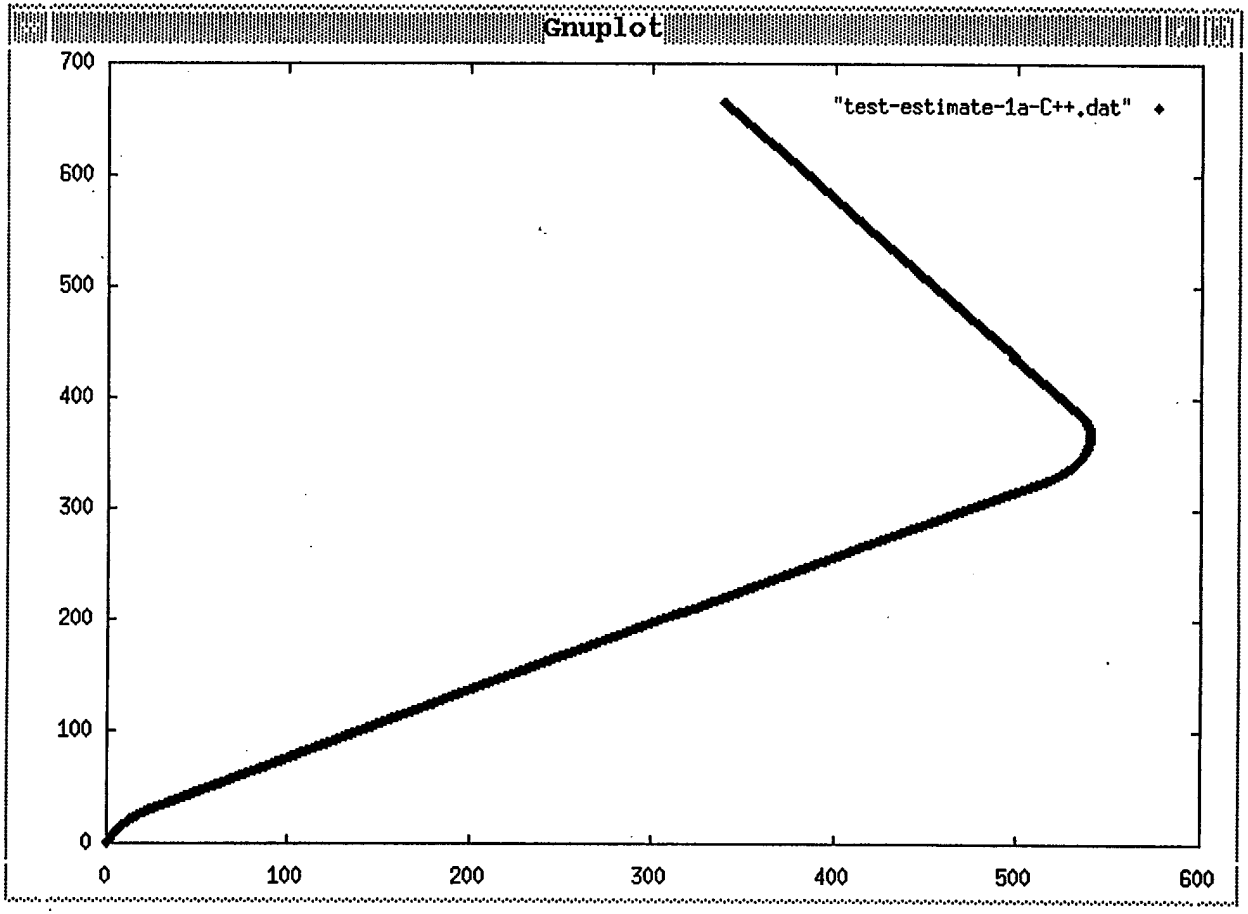


Figure A-7: New filter version, C++, K1: 0.0; K2: 0.0; K3: 0.5; K4: 0.7

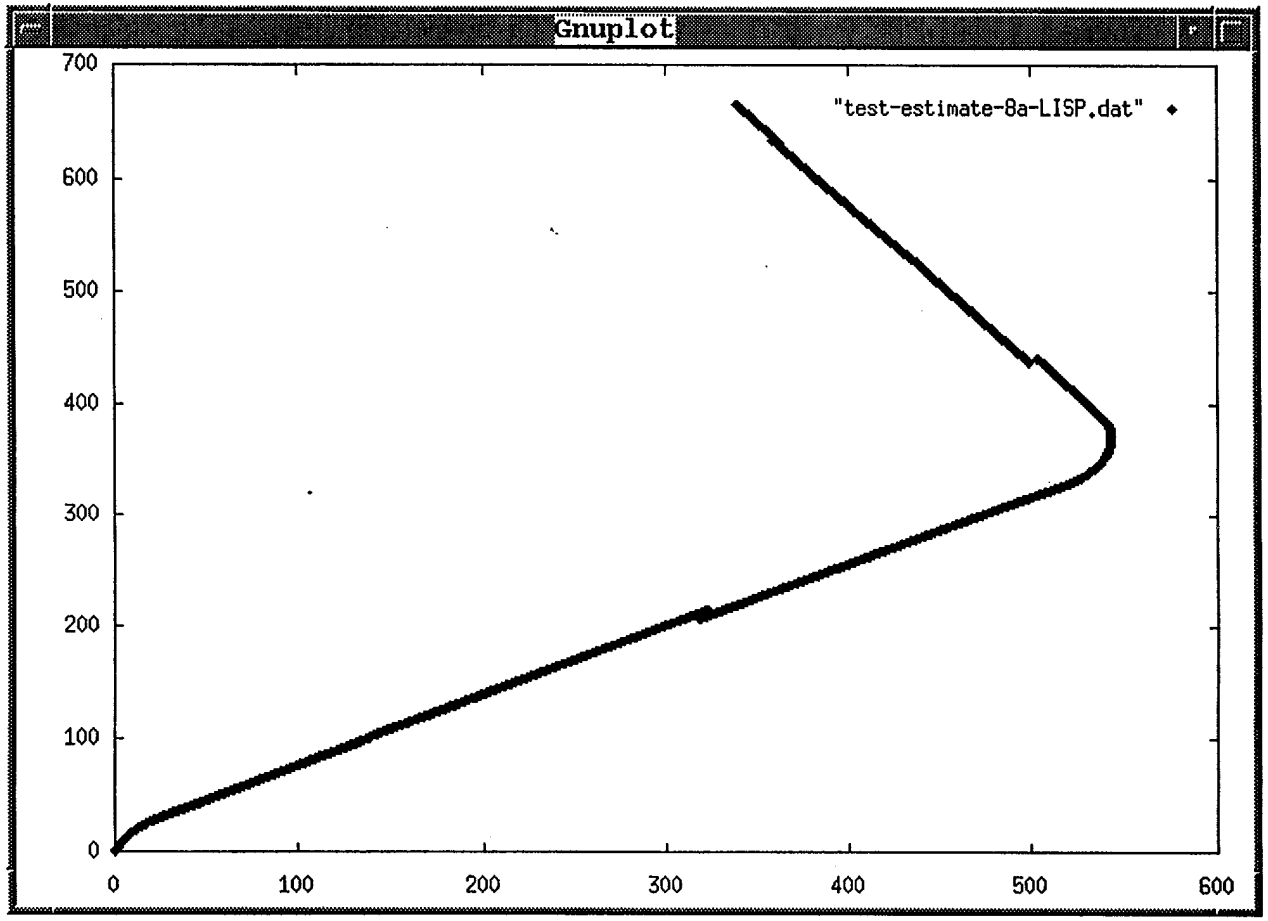


Figure A-8: New filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.1; K4: 0.7

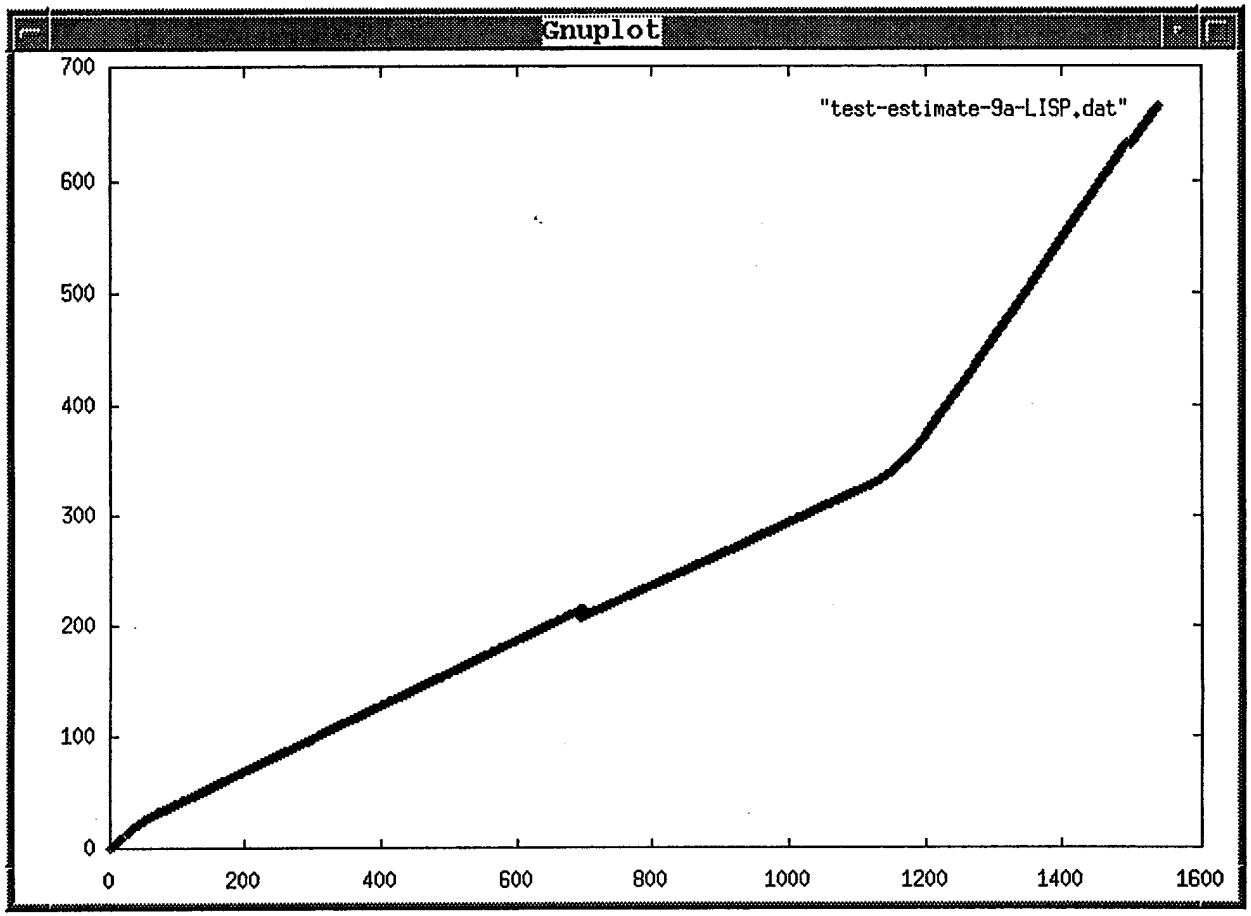


Figure A-9: New filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.1; K4: 0.7

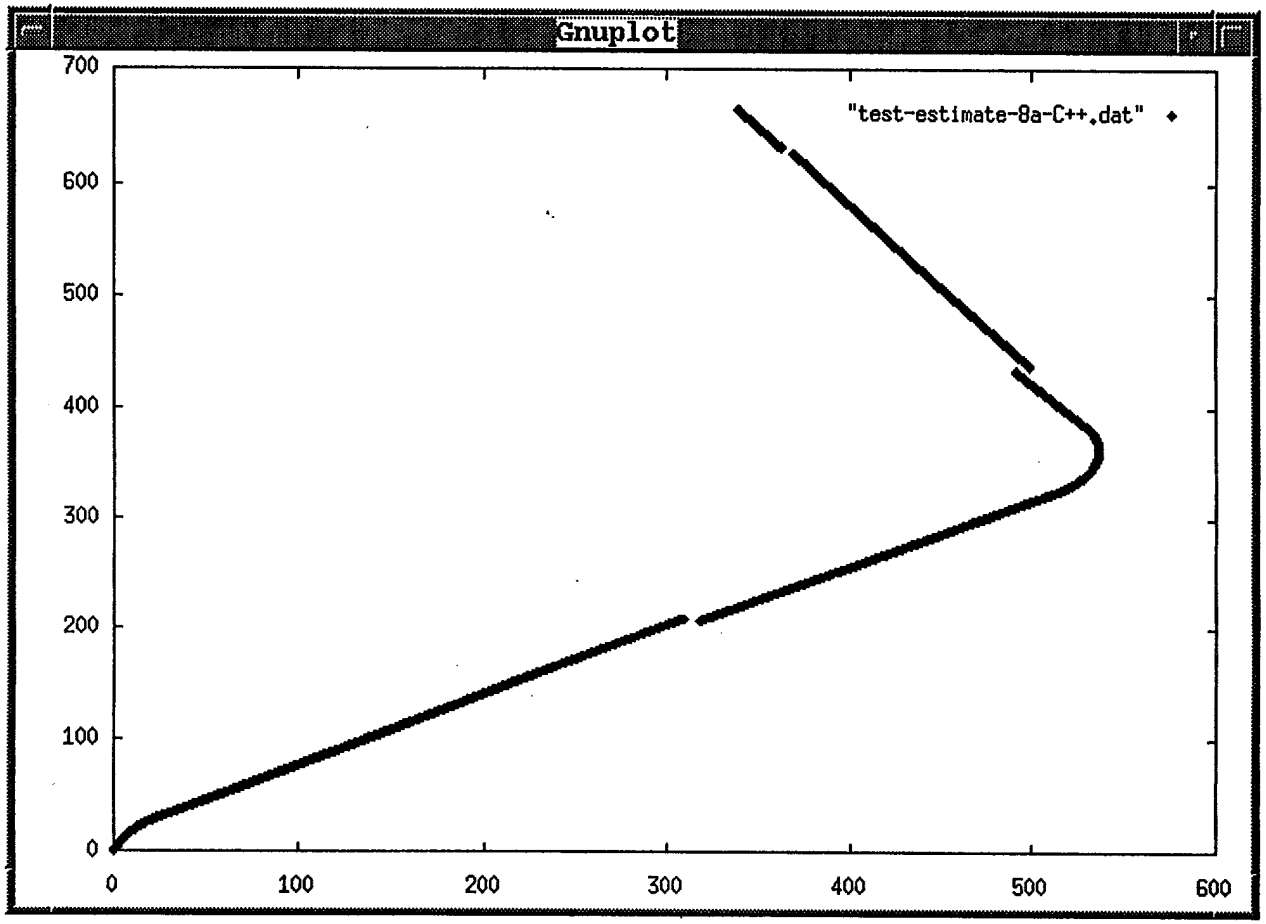


Figure A-10: New filter version, C++, K1: 0.0; K2: 0.0; K3: 0.1; K4: 0.7

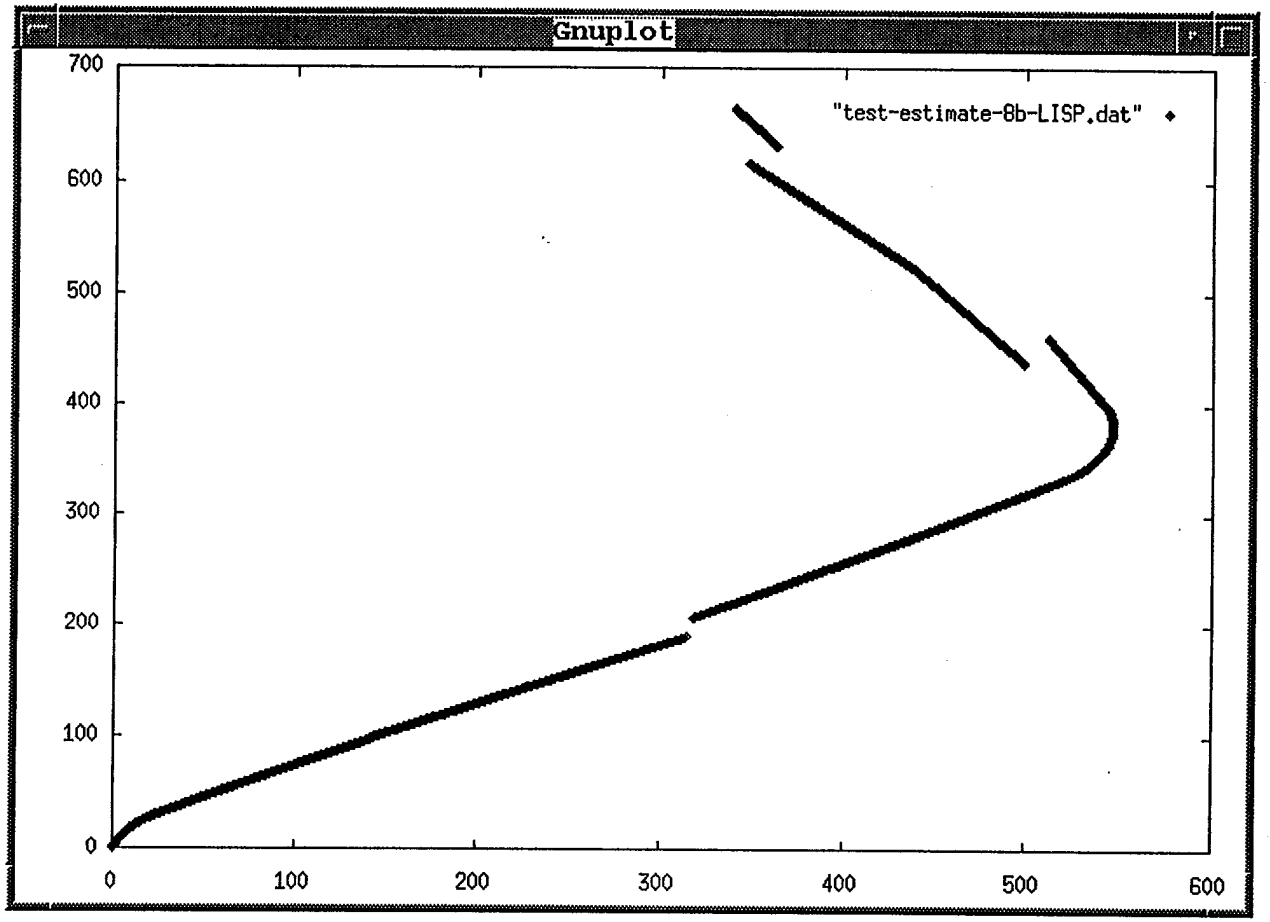


Figure A-11: Original filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.1; K4: 0.7

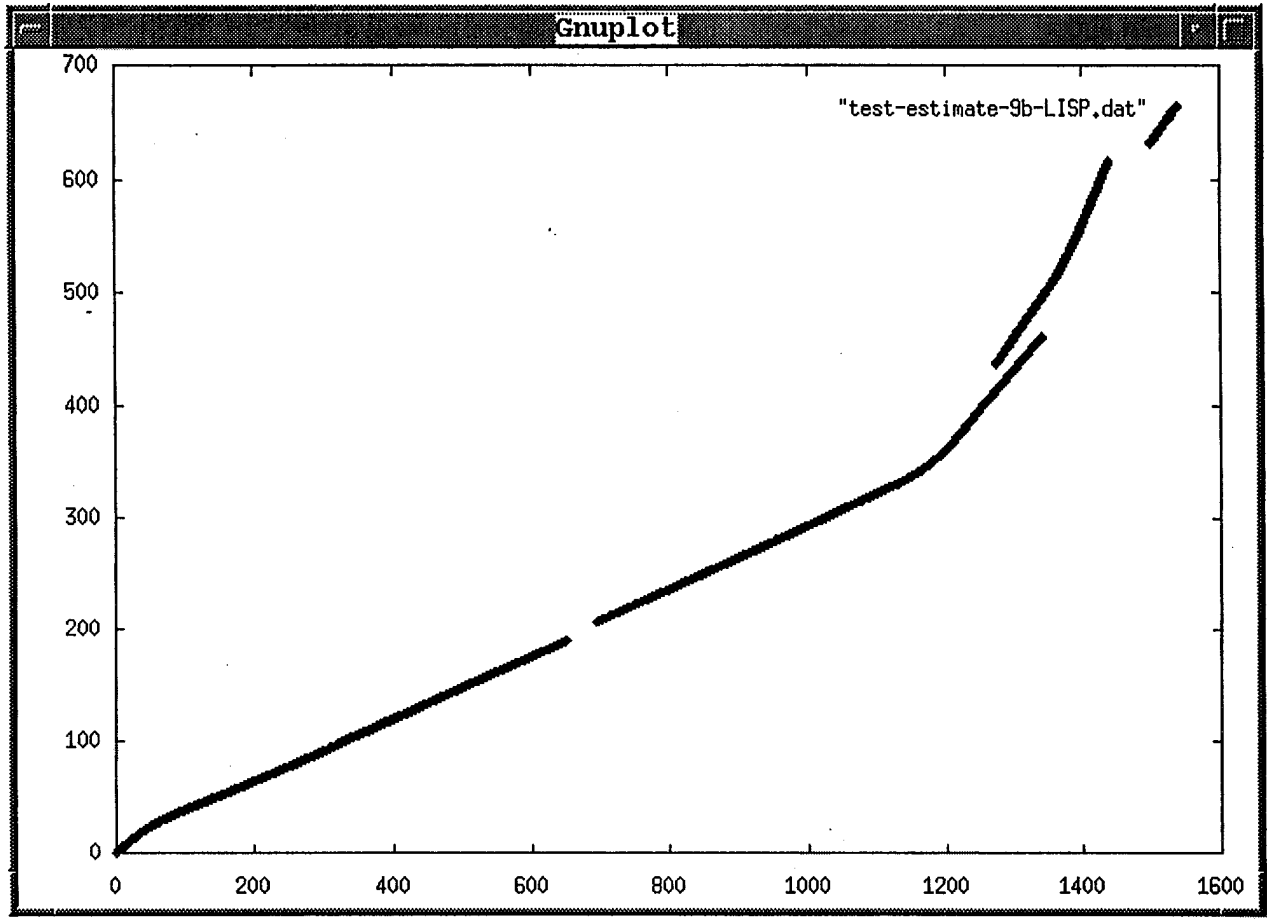


Figure A-12: Original filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.1; K4: 0.7

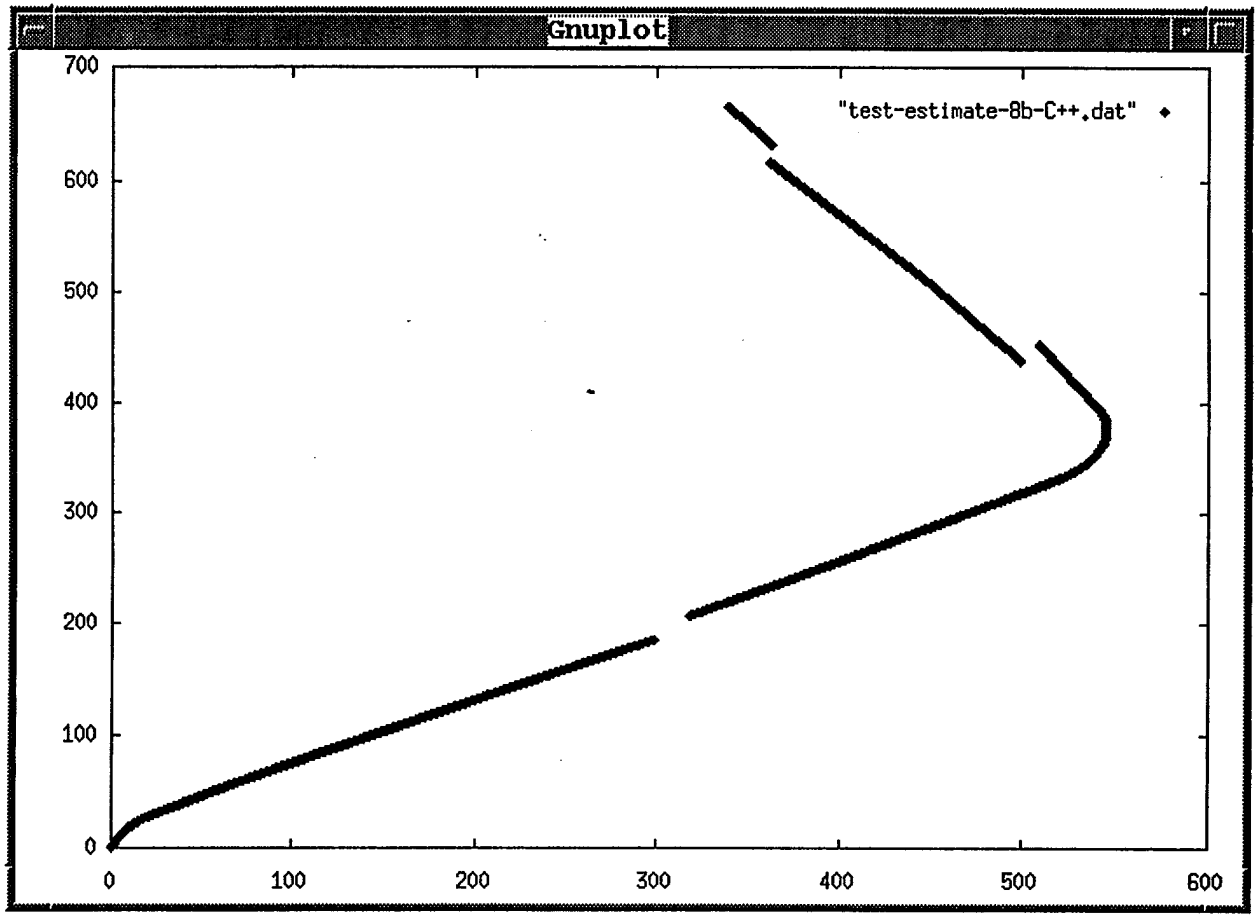


Figure A-13: Original filter version, C++, K1: 0.0; K2: 0.0; K3: 0.1; K4: 0.7

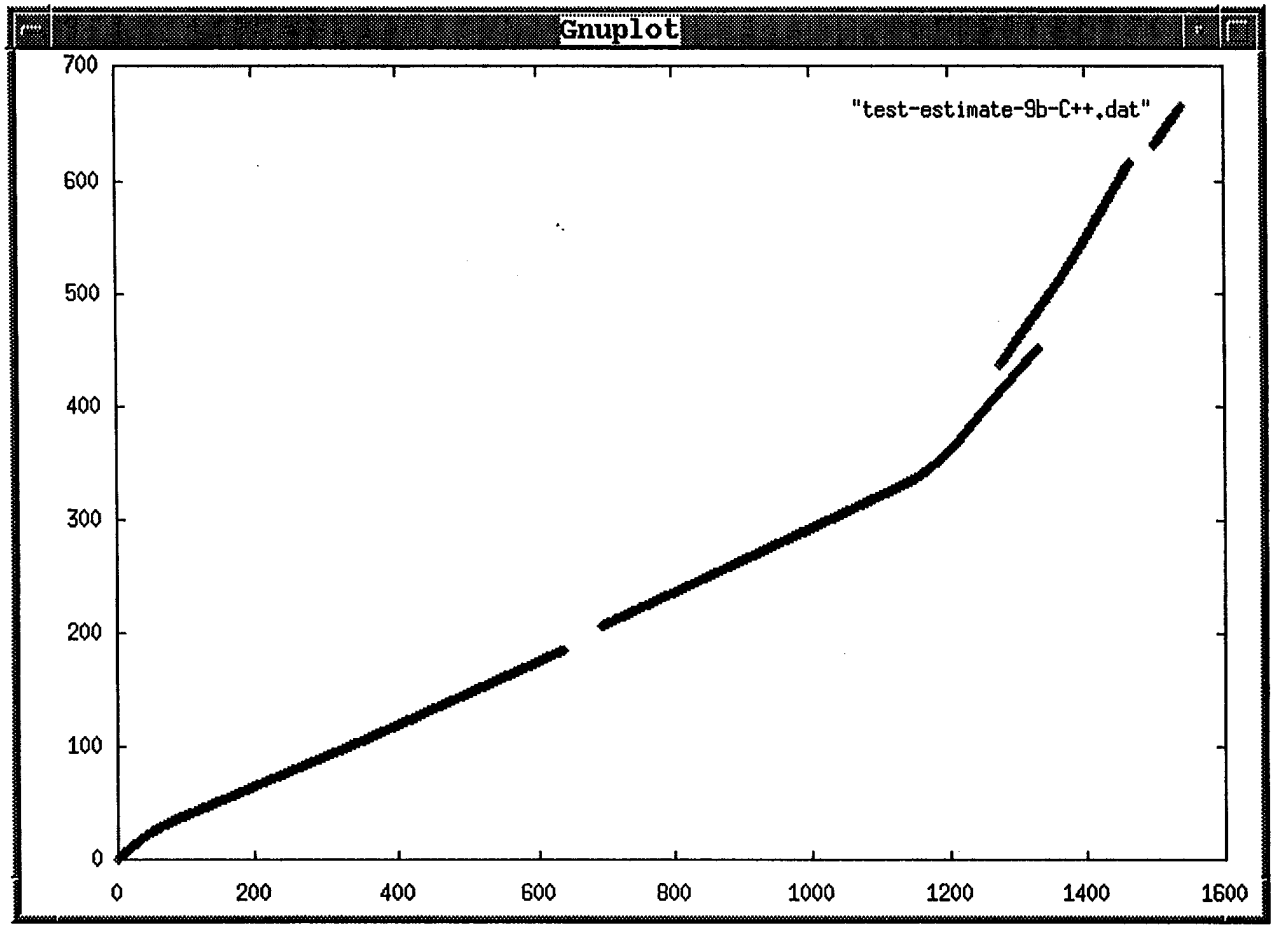


Figure A-14: Original filter version, C++, K1: 0.0; K2: 0.0; K3: 0.1; K4: 0.7

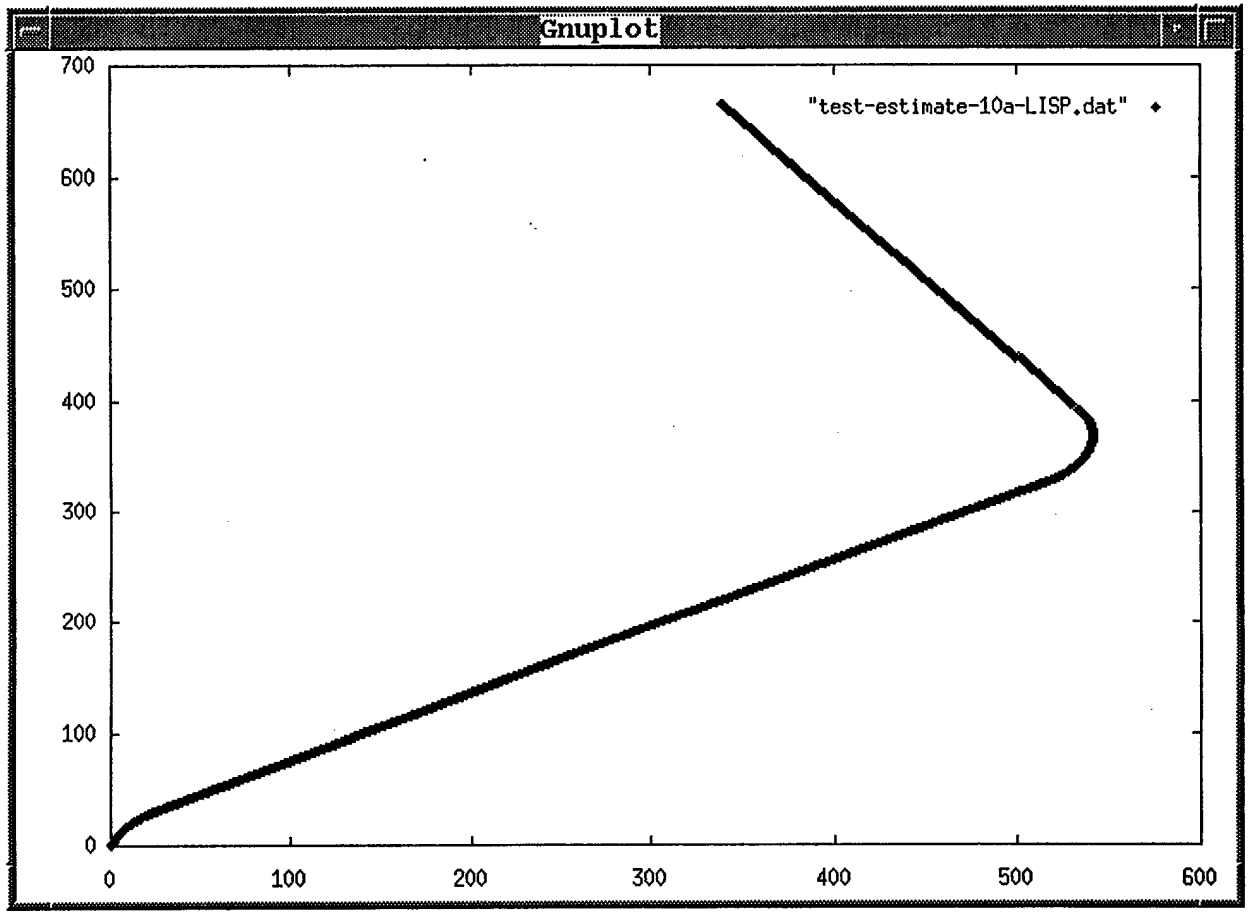


Figure A-15: New filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.7; K4: 0.7

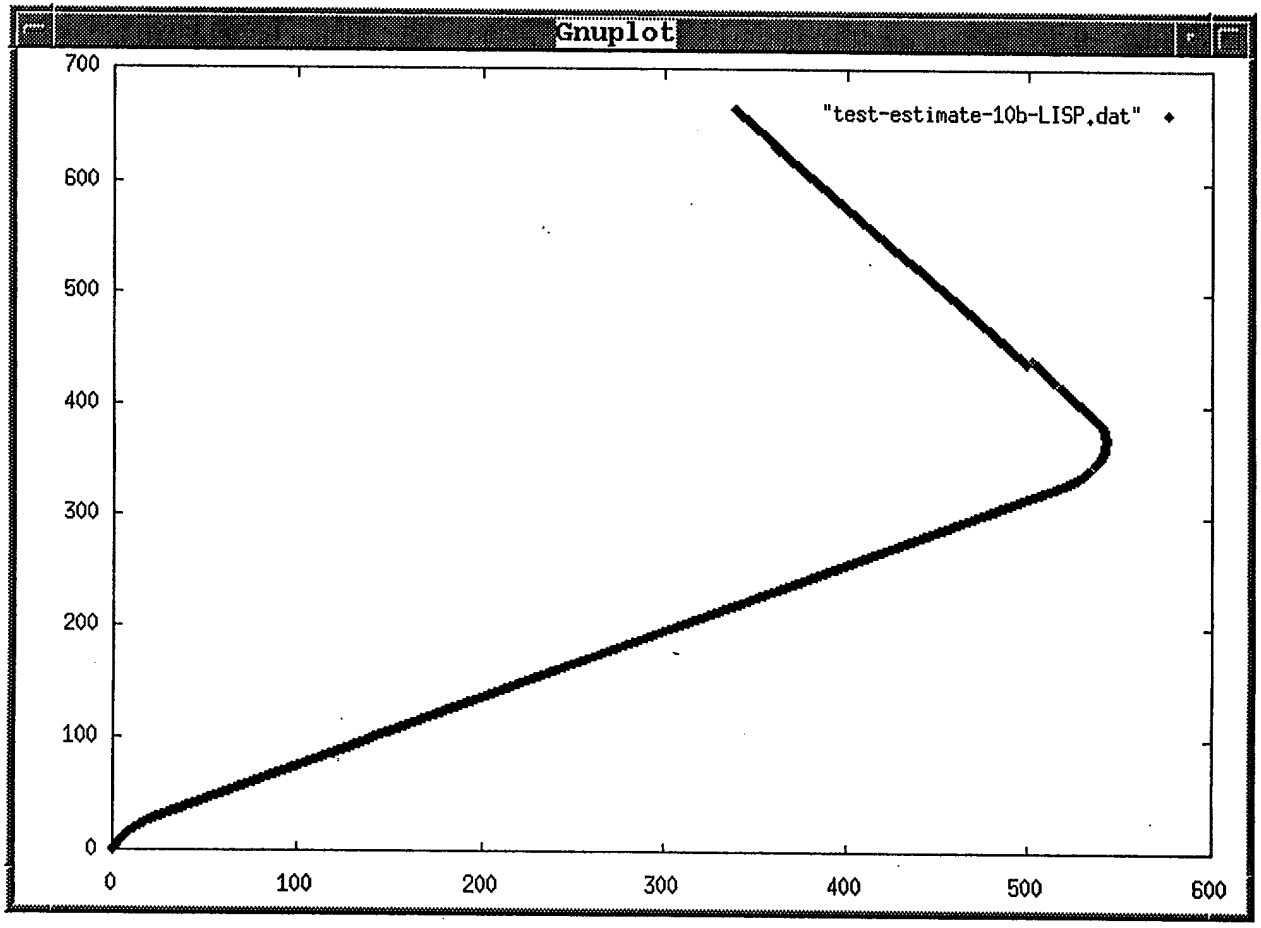


Figure A-16: Original filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.7; K4: 0.7

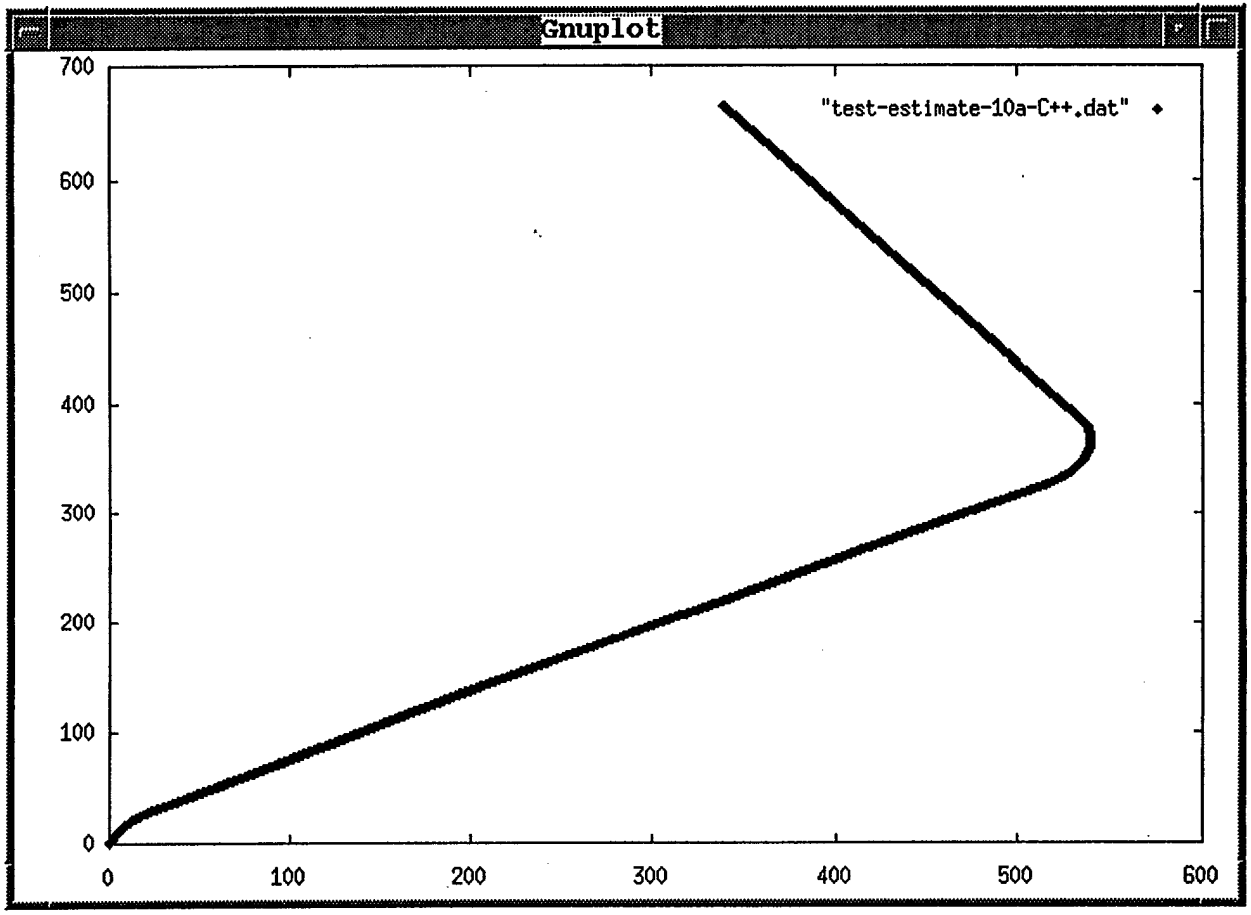


Figure A-17: New filter version, C++, K1: 0.0; K2: 0.0; K3: 0.7; K4: 0.7

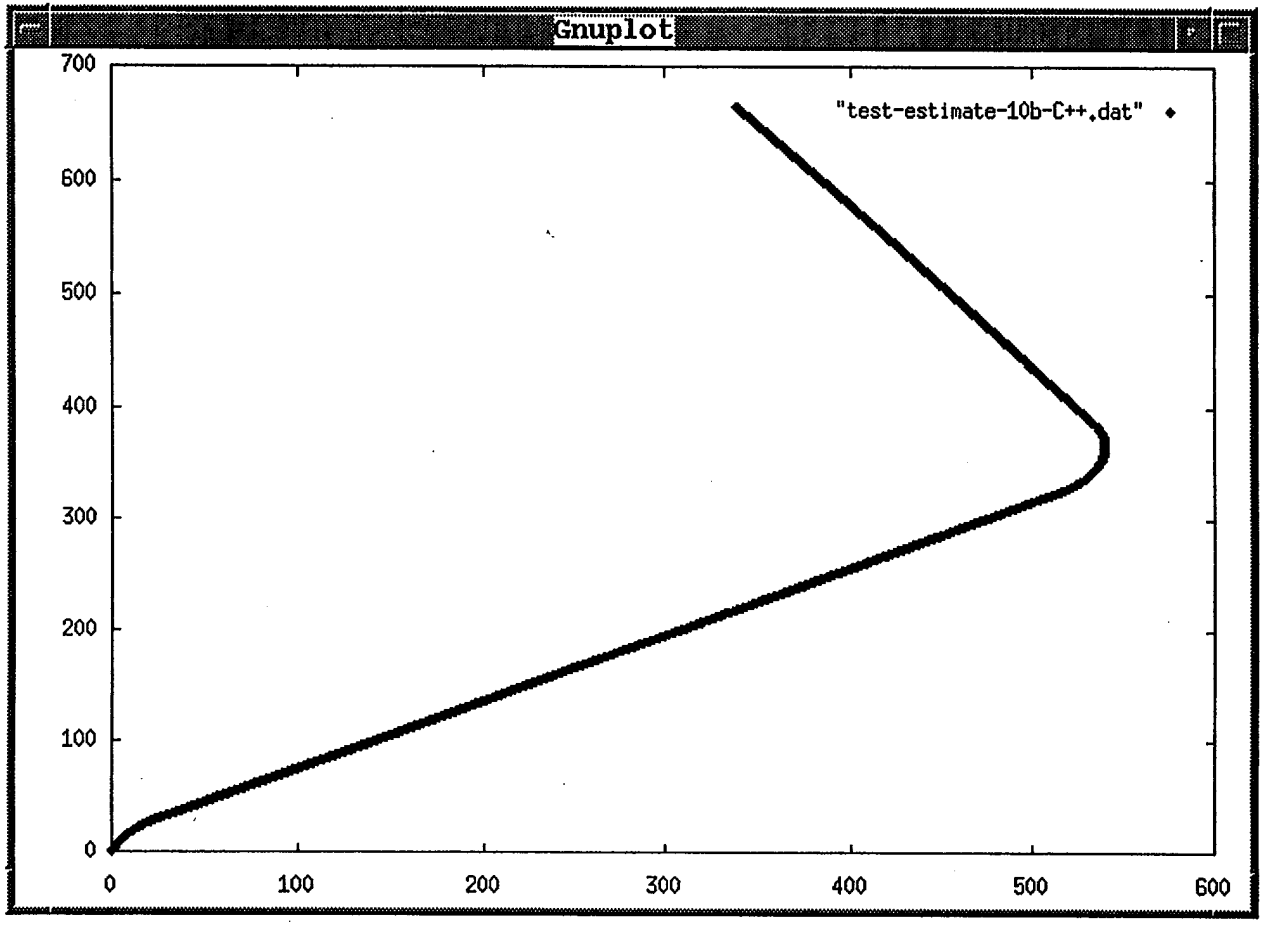


Figure A-18: Original filter version, C++, K1: 0.0; K2: 0.0; K3: 0.7; K4: 0.7

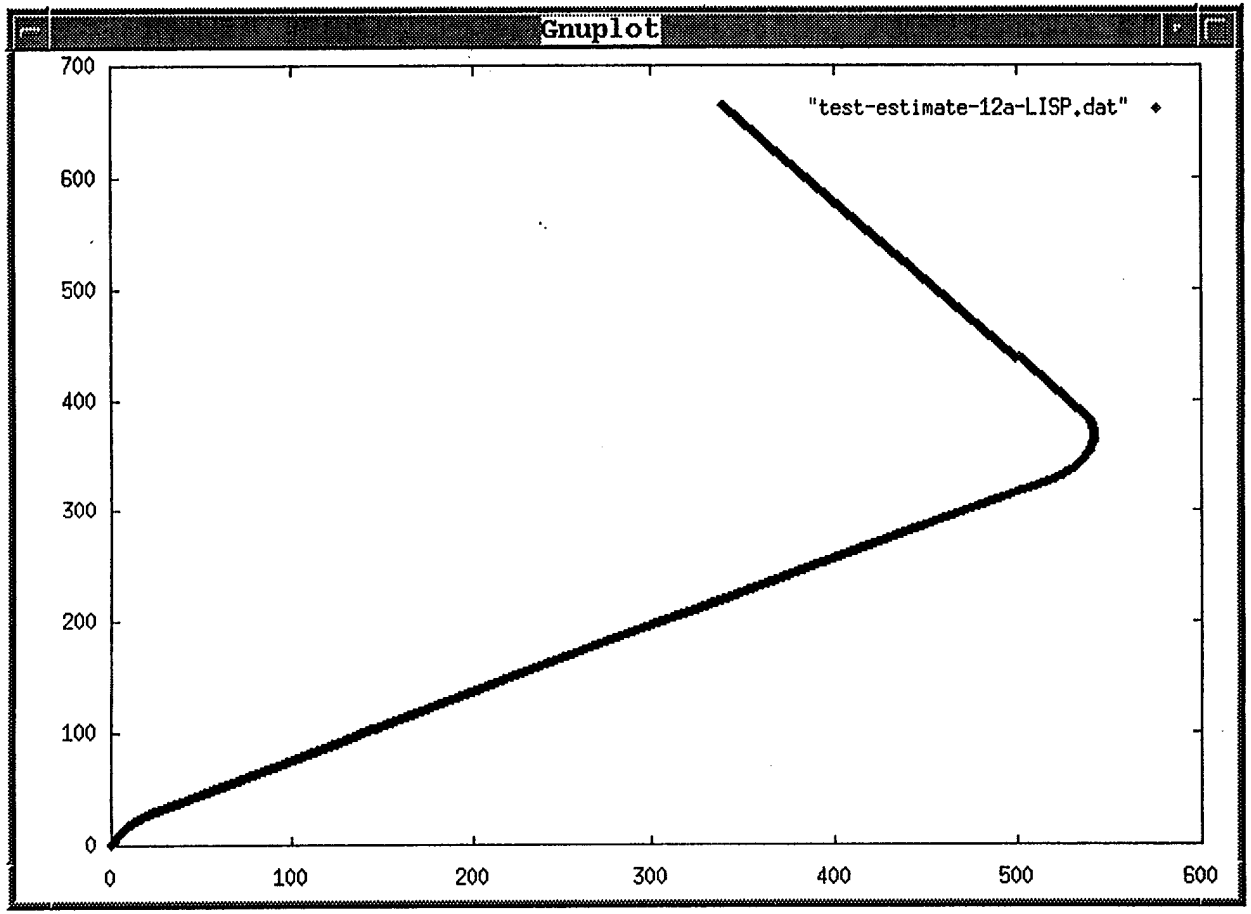


Figure A-19: New filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.5; K4: 0.3

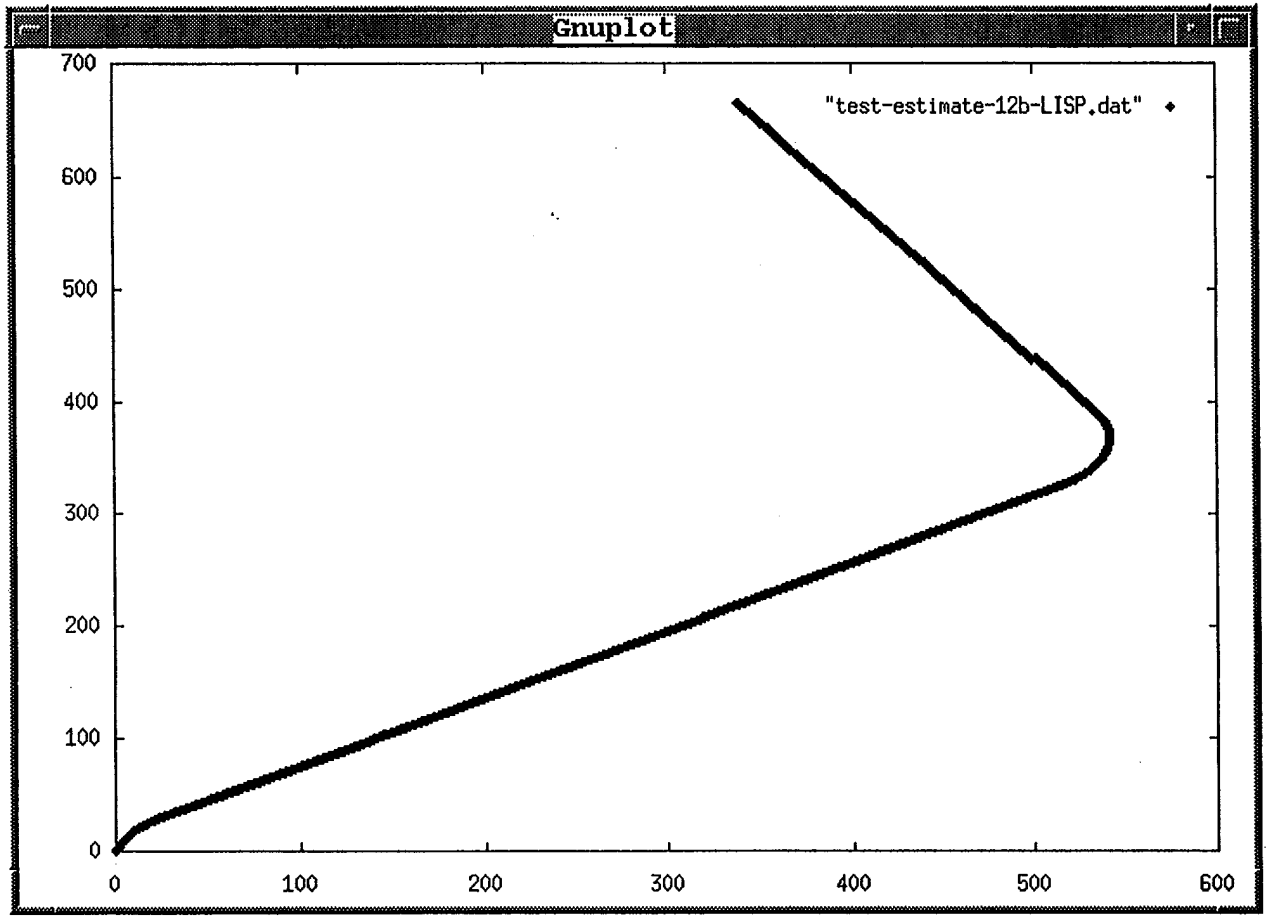


Figure A-20: Original filter version, LISP, K1: 0.0; K2: 0.0; K3: 0.5; K4: 0.3

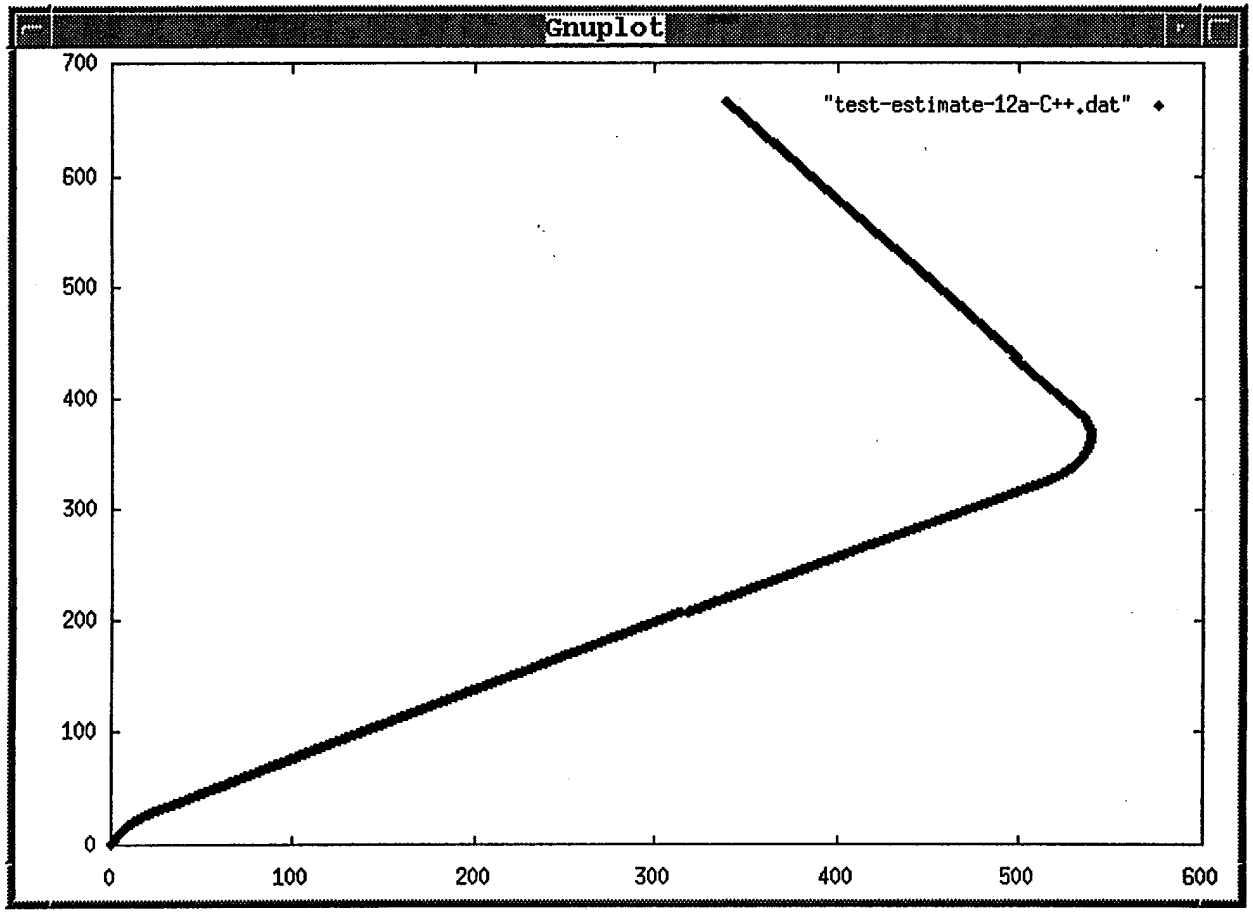


Figure A-21: New filter version, C++, K1: 0.0; K2: 0.0; K3: 0.5; K4: 0.3

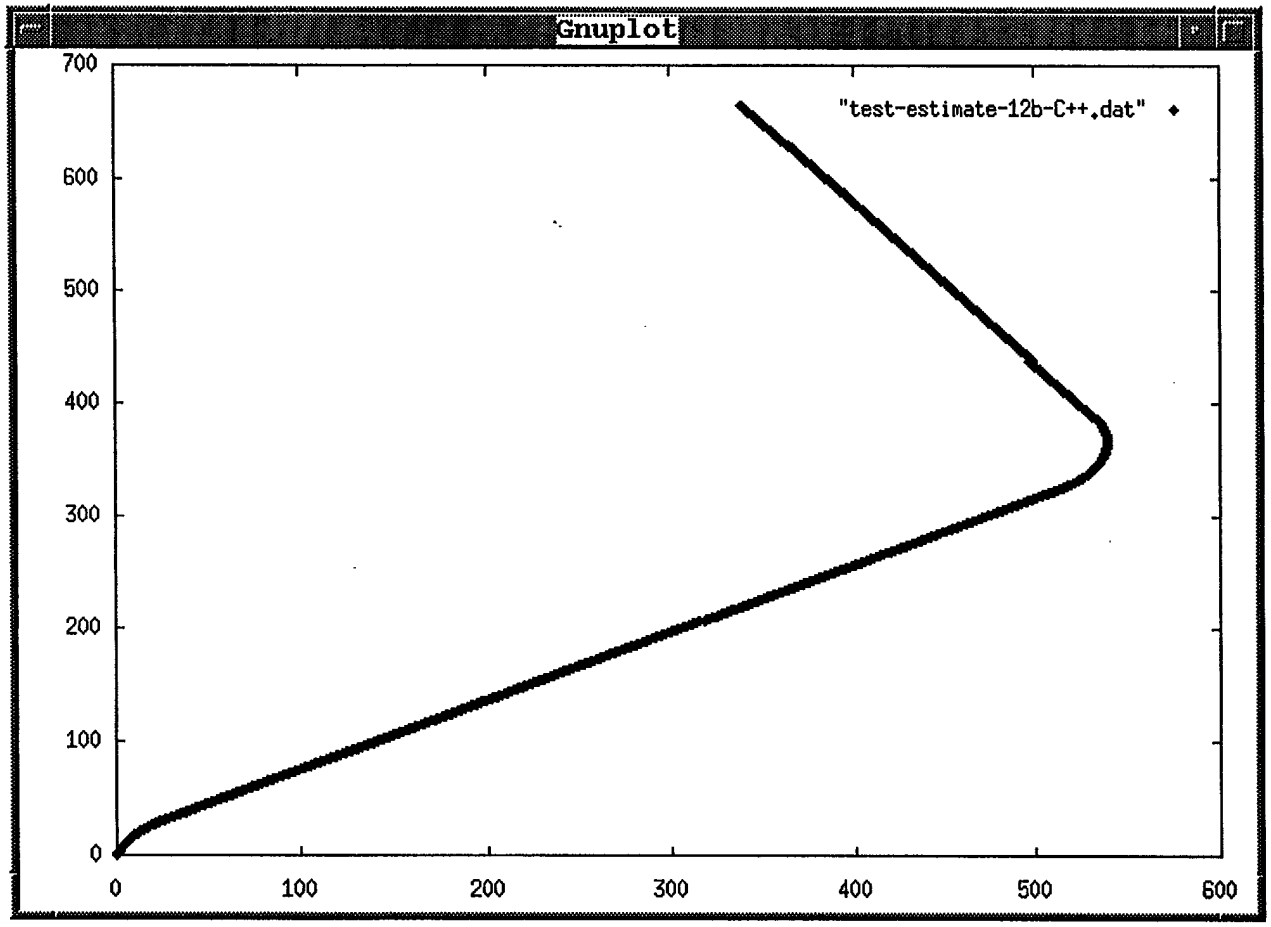


Figure A-22: Original filter version, C++, K1: 0.0; K2: 0.0; K3: 0.5; K4: 0.3

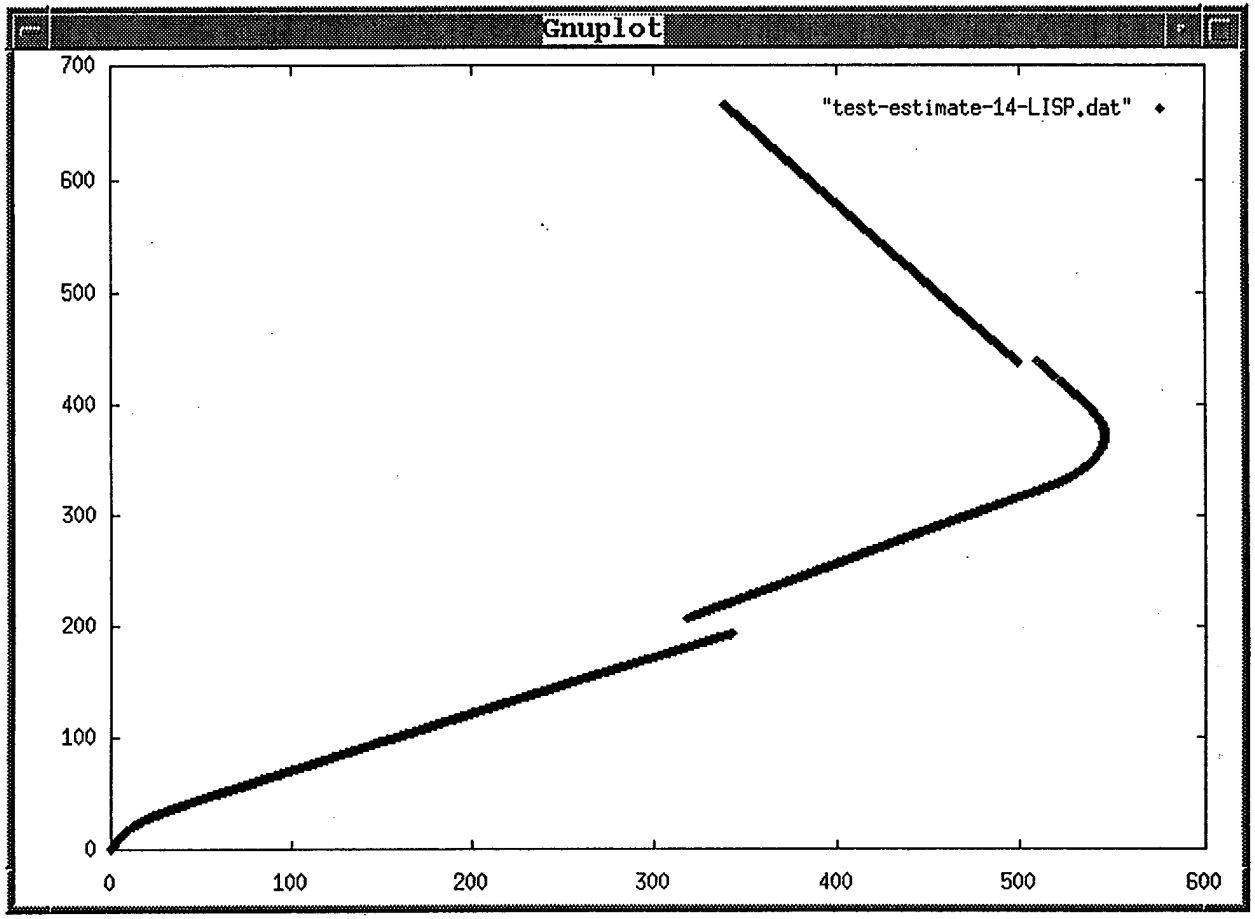


Figure A-23: New filter version, LISP, K1: 0.4; K2: 0.8; K3: 0.5; K4: 0.7

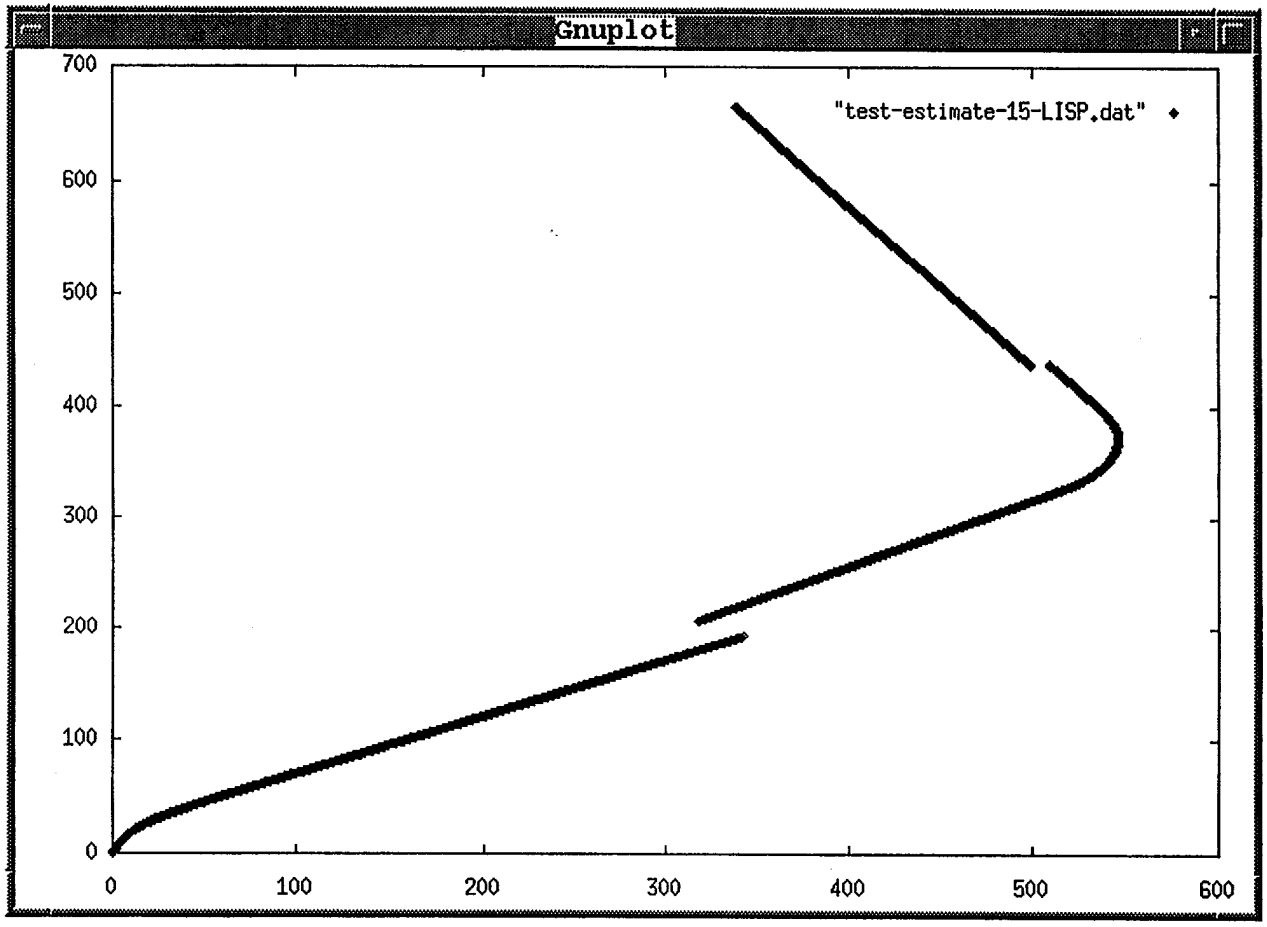


Figure A-24: New filter version, LISP, K1: 0.4; K2: 0.1; K3: 0.5; K4: 0.7

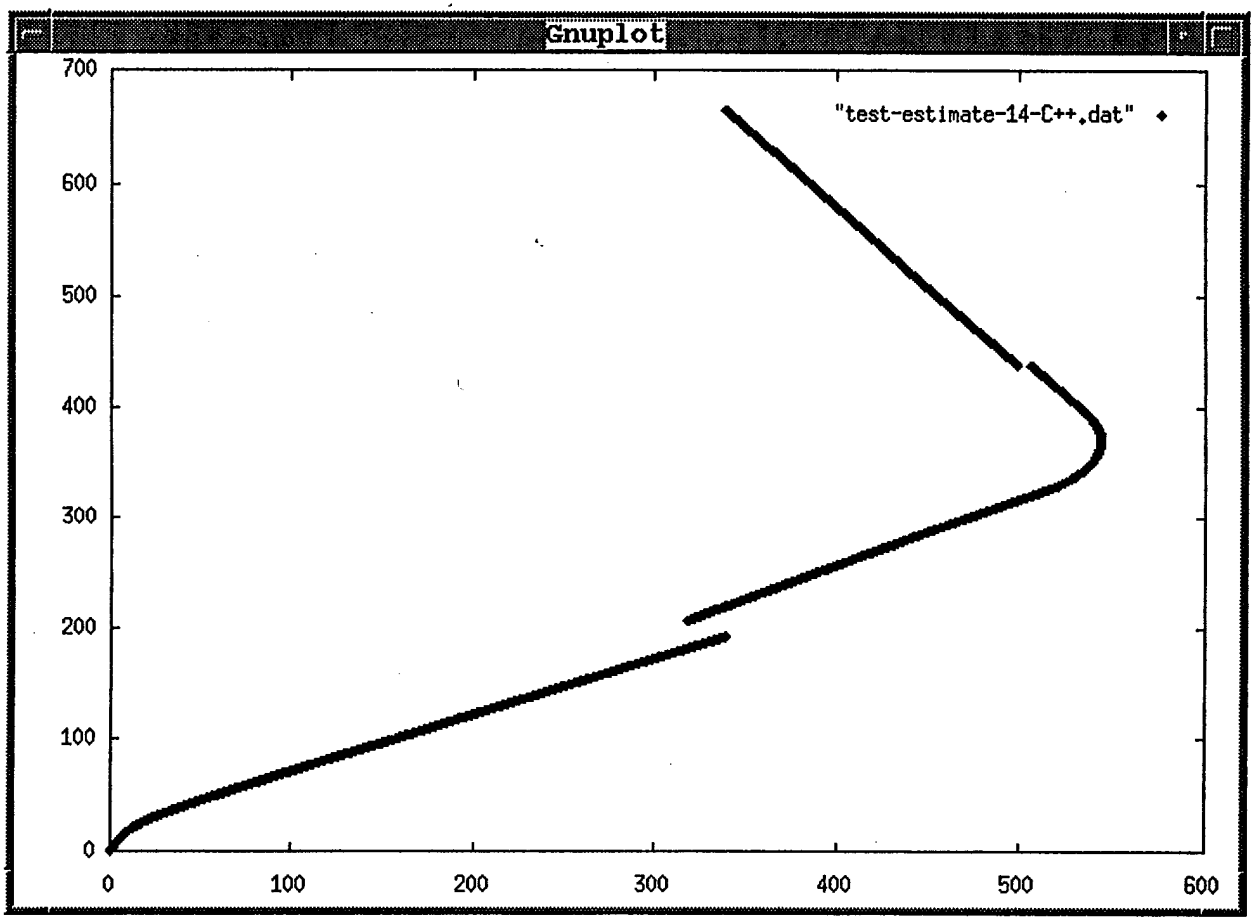


Figure A-25: New filter version, C++, K1: 0.4; K2: 0.8; K3: 0.5; K4: 0.7

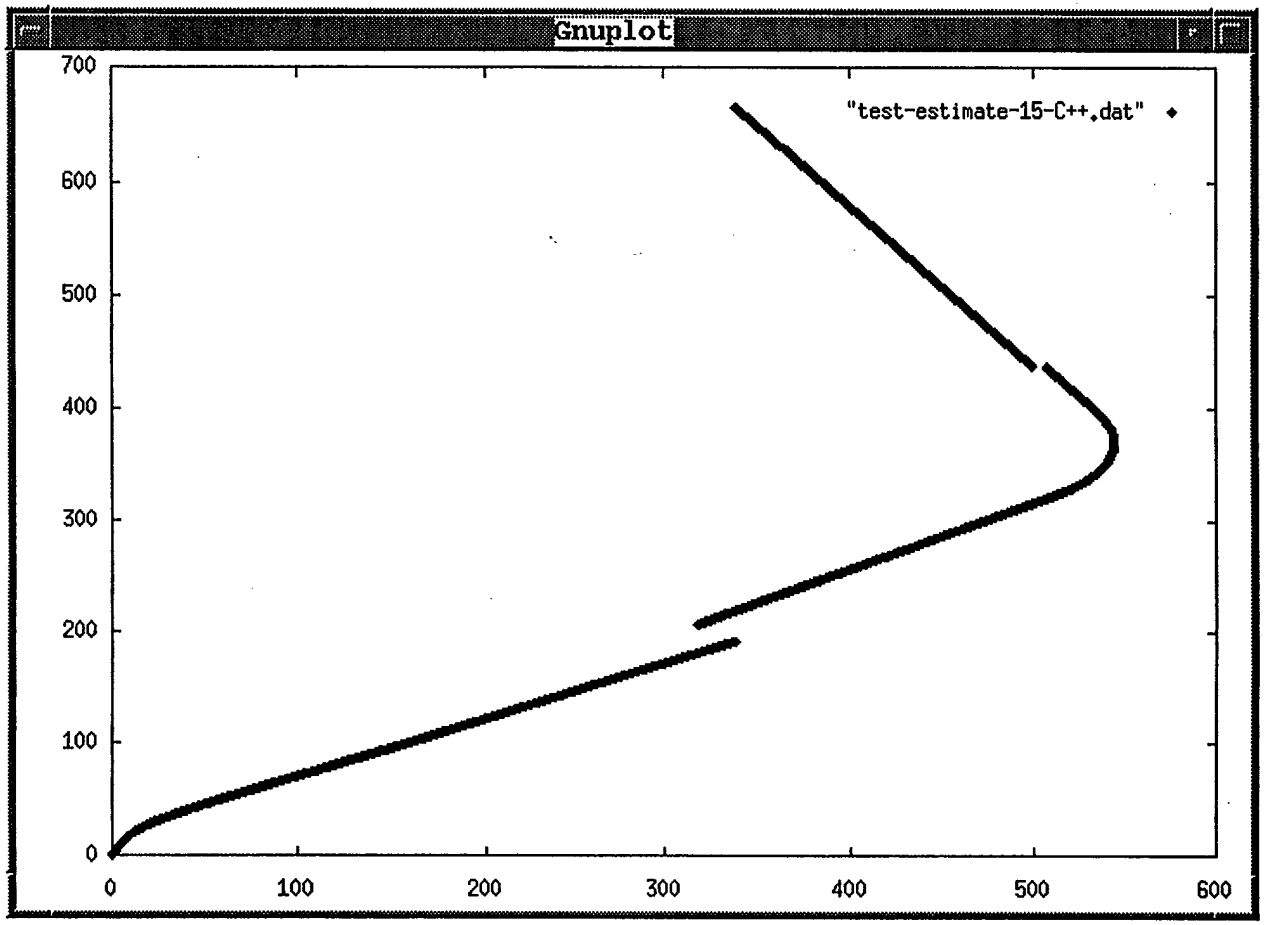


Figure A-26: New filter version, C++, K1: 0.4; K2: 0.1; K3: 0.5; K4: 0.7

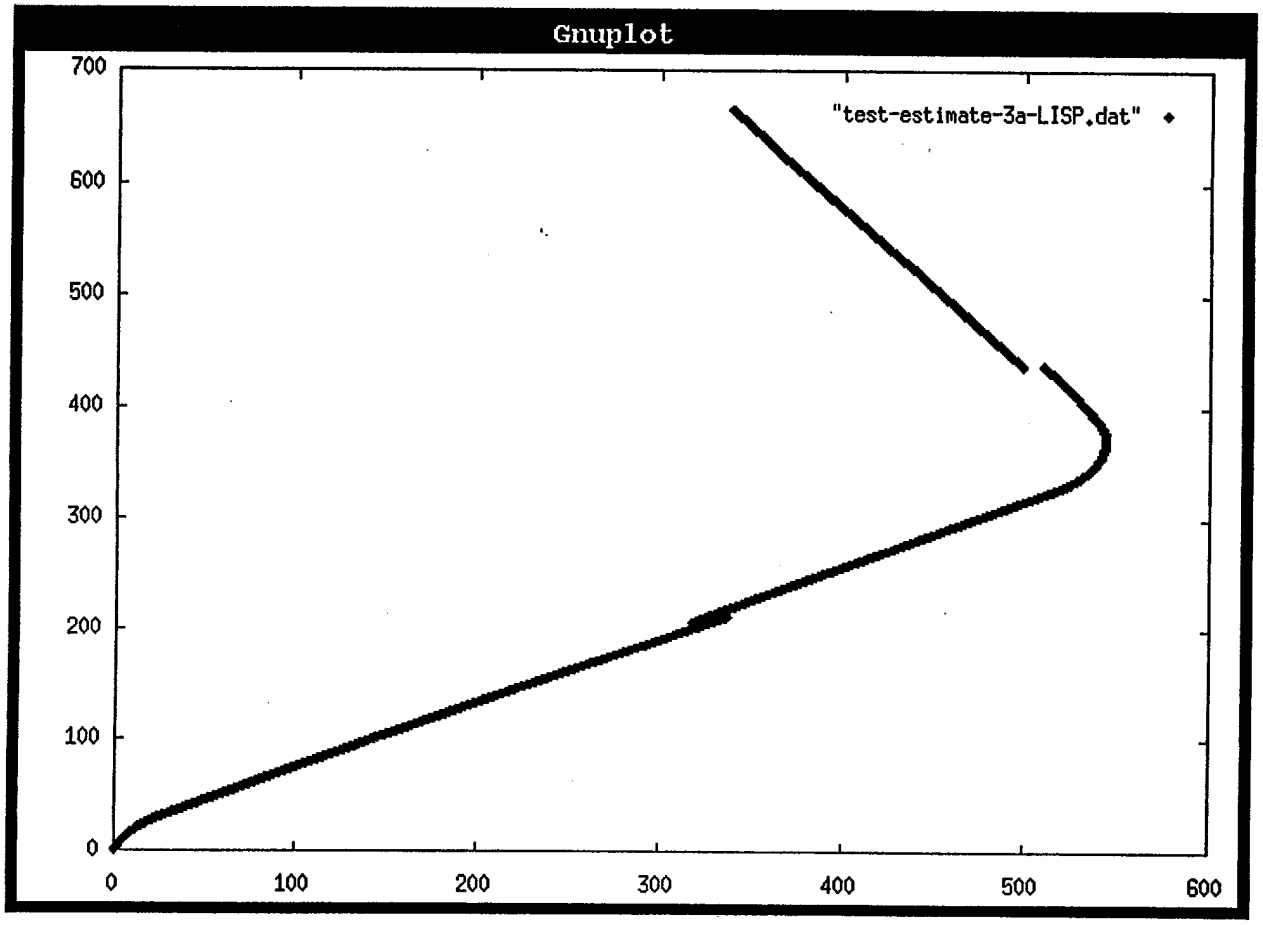


Figure A-27: New filter version, LISP, K1: 0.1; K2: 0.1; K3: 0.5; K4: 0.7

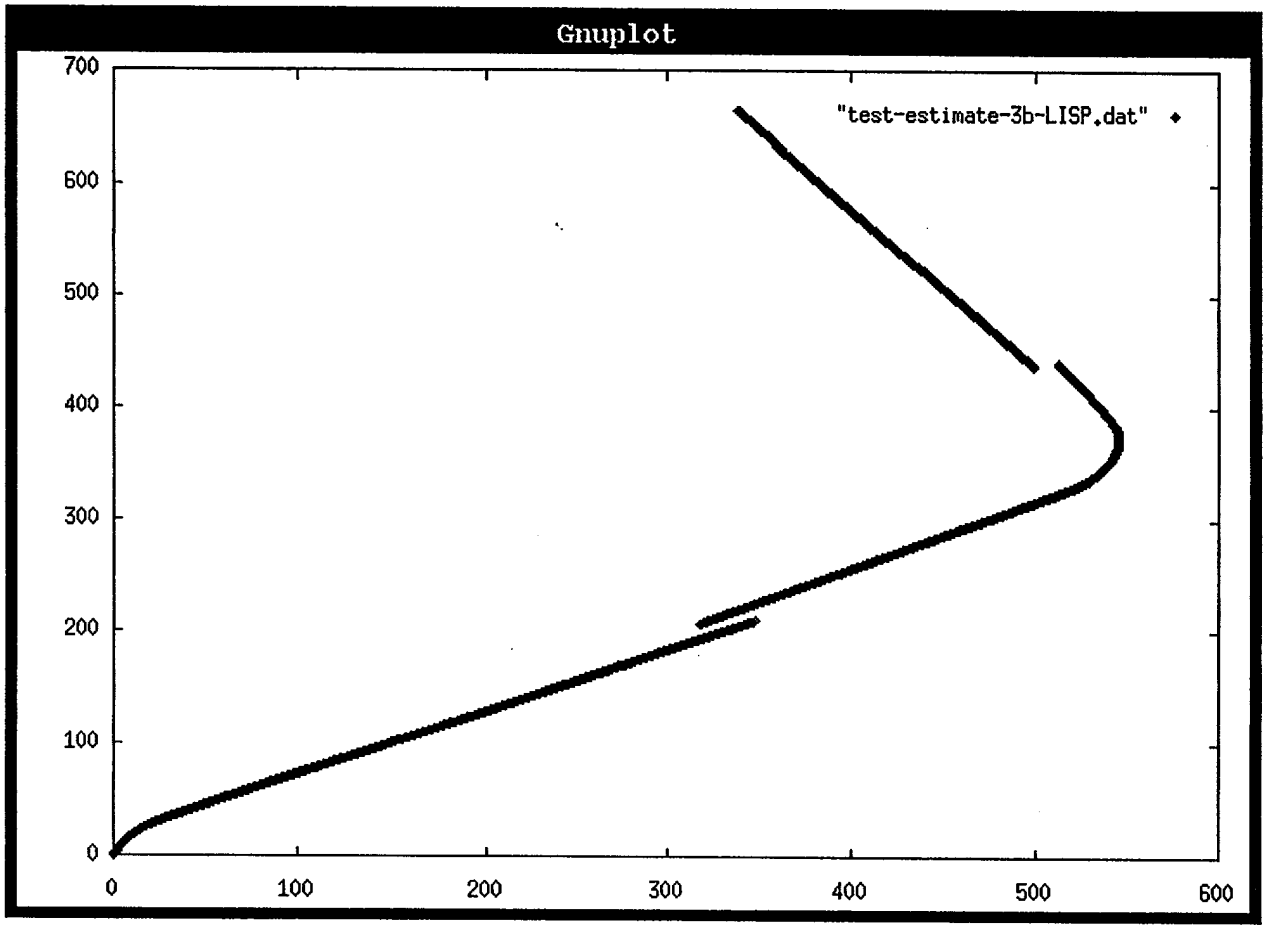


Figure A-28: Original filter version, LISP, K1: 0.1; K2: 0.1; K3: 0.5; K4: 0.7

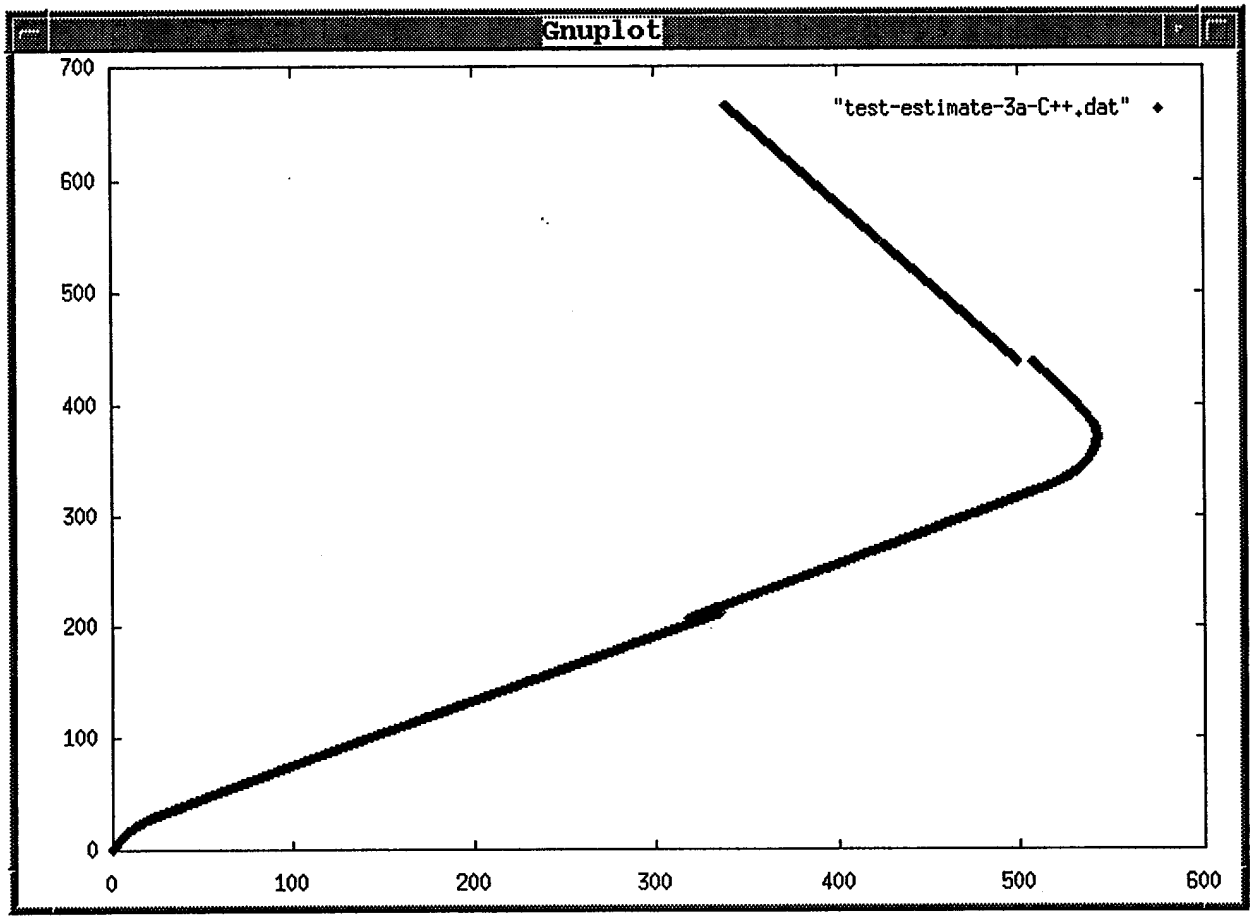


Figure A-29: New filter version, C++, K1: 0.1; K2: 0.1; K3: 0.5; K4: 0.7

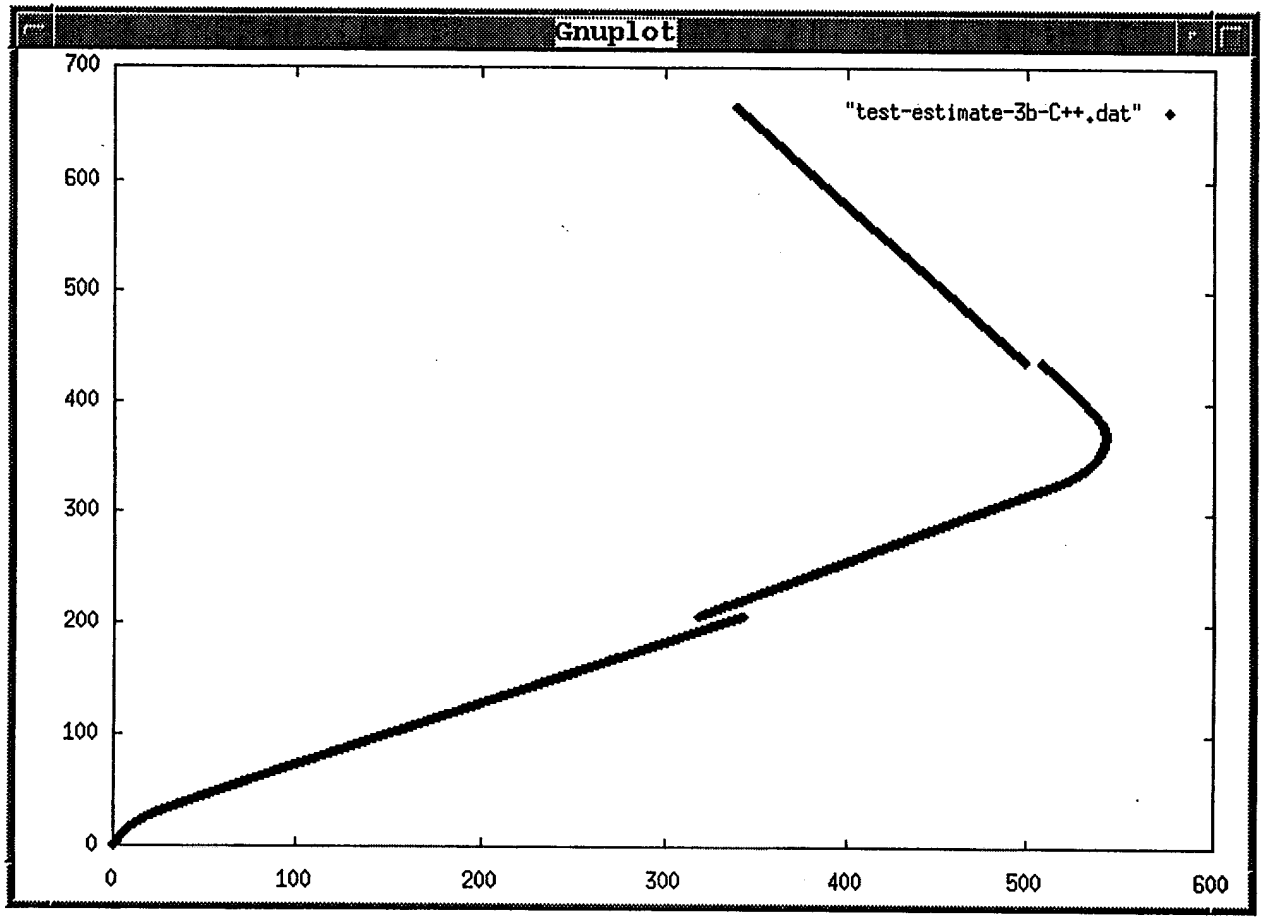


Figure A-30: Original filter version, C++, K1: 0.1; K2: 0.1; K3: 0.5; K4: 0.7

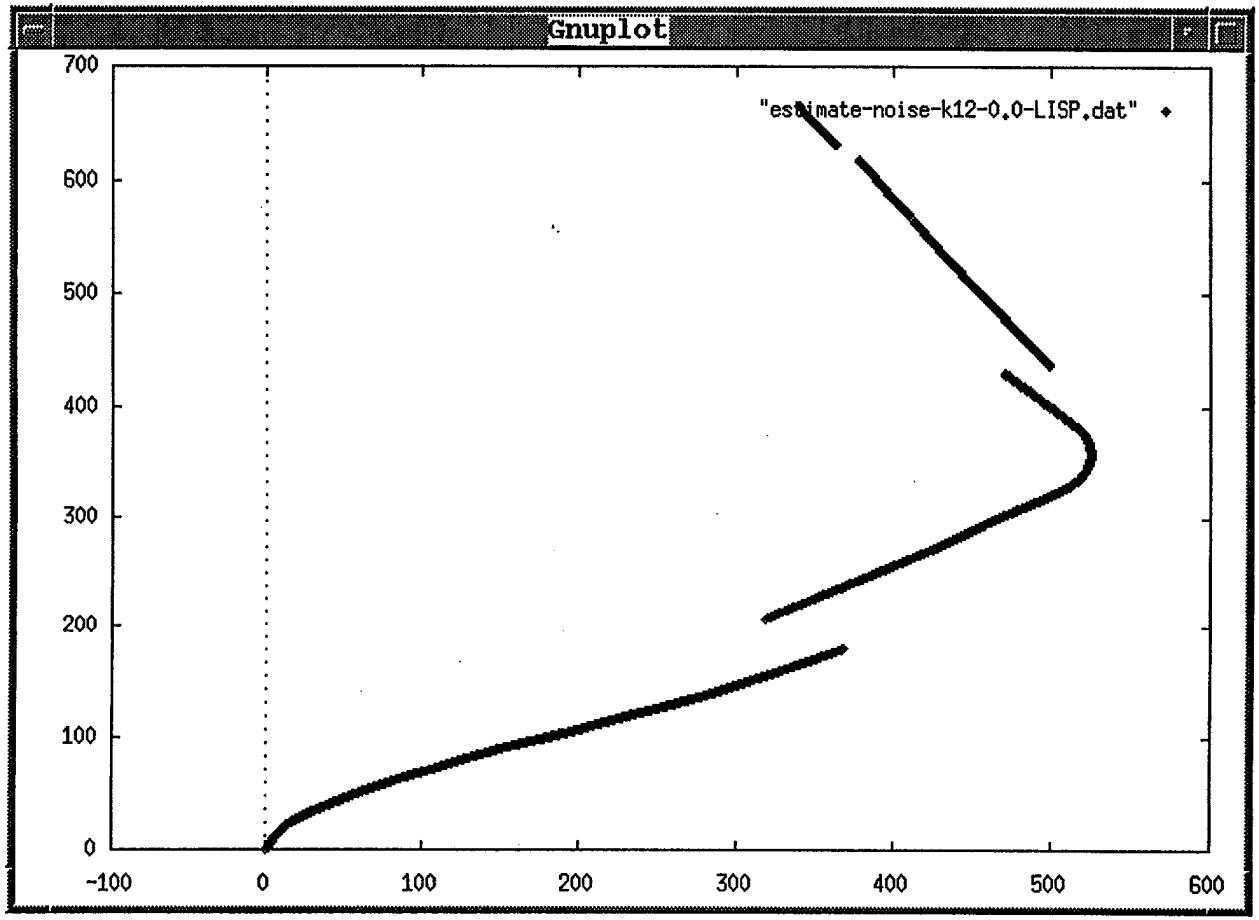


Figure A-31: New filter version, added noise, LISP, K1: 0.0; K2: 0.0; K3: 0.5; K4: 0.7

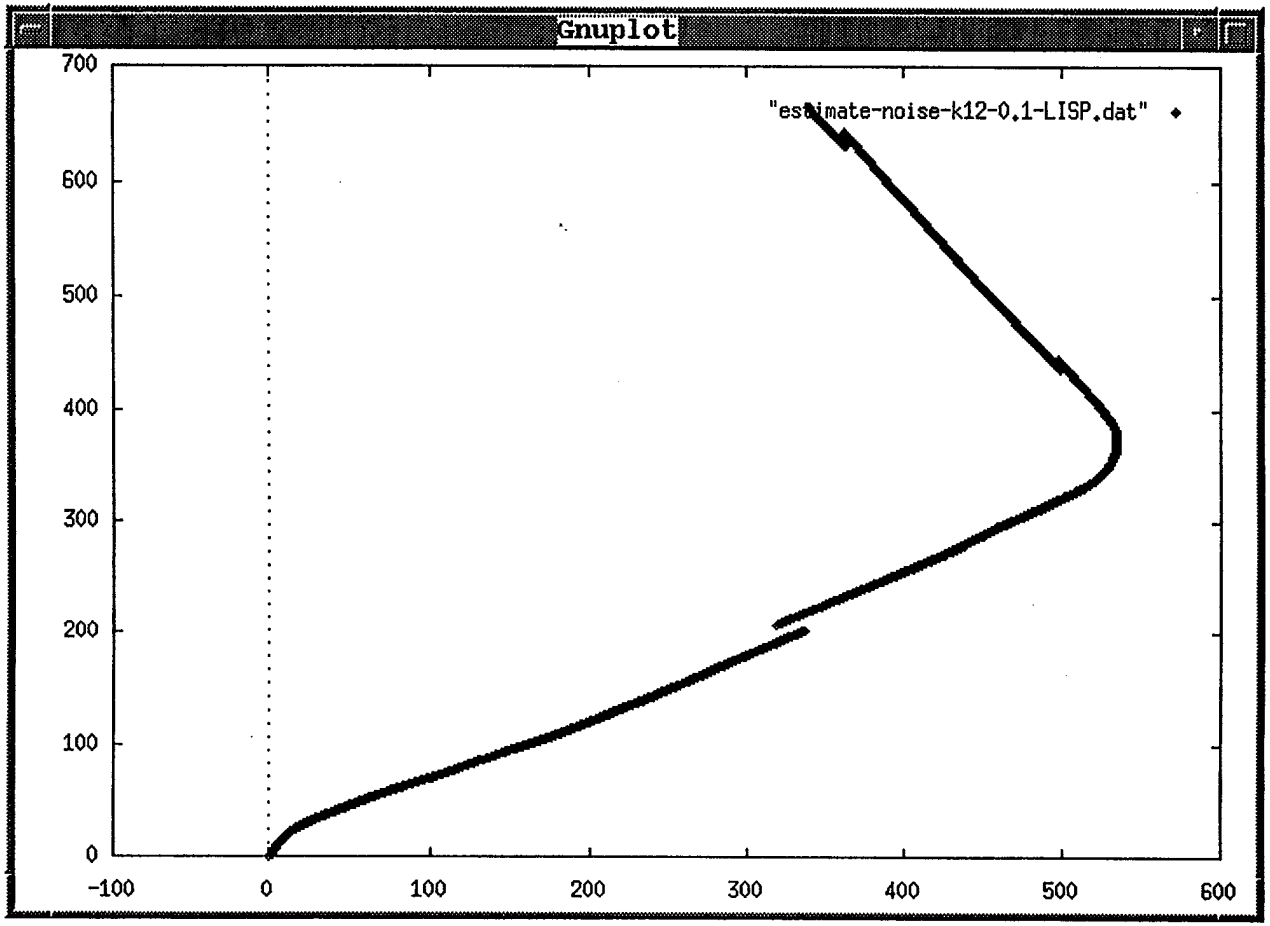


Figure A-32: New filter version, added noise, LISP, K1: 0.1; K2: 0.1; K3: 0.5; K4: 0.7

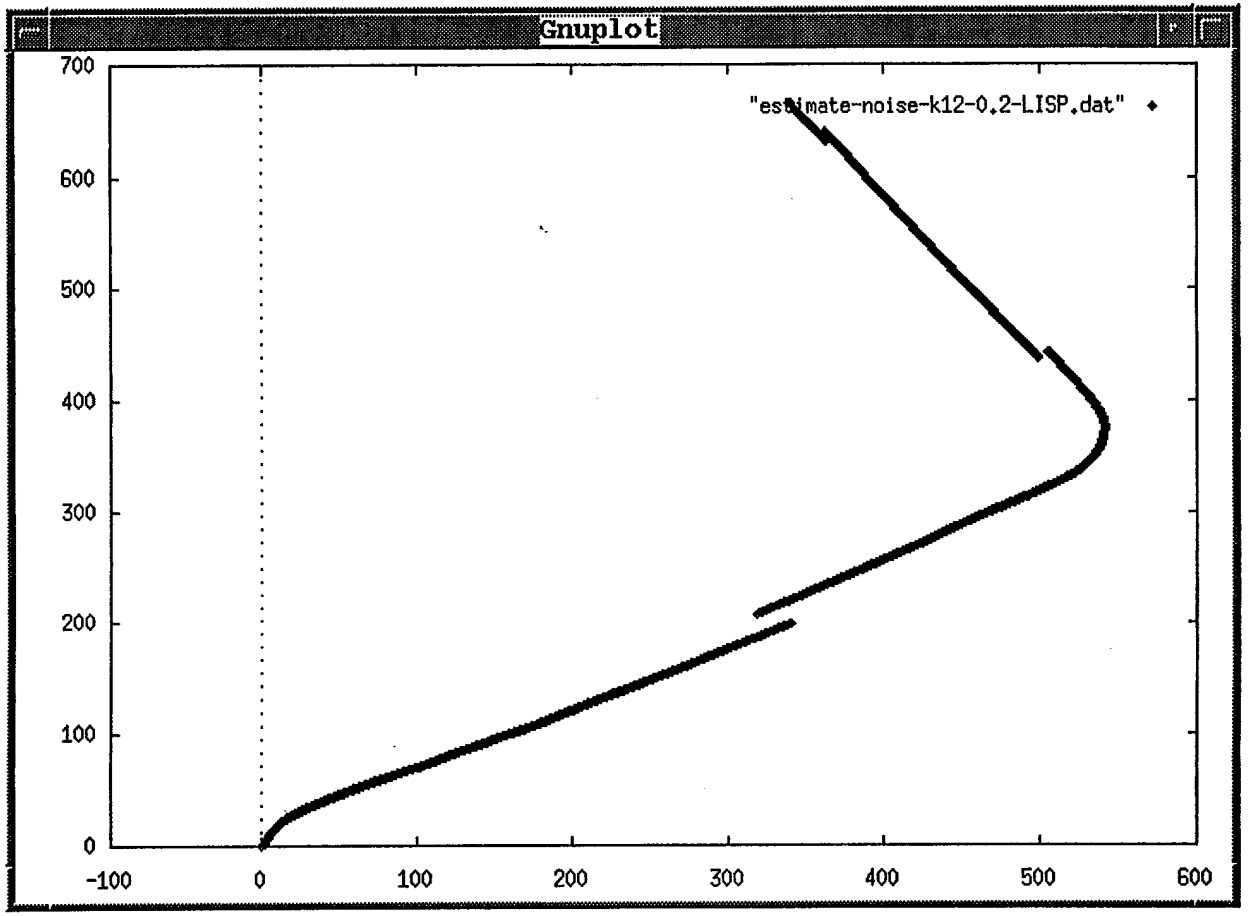


Figure A-33: New filter version, added noise, LISP, K1: 0.2; K2: 0.2; K3: 0.5; K4: 0.7

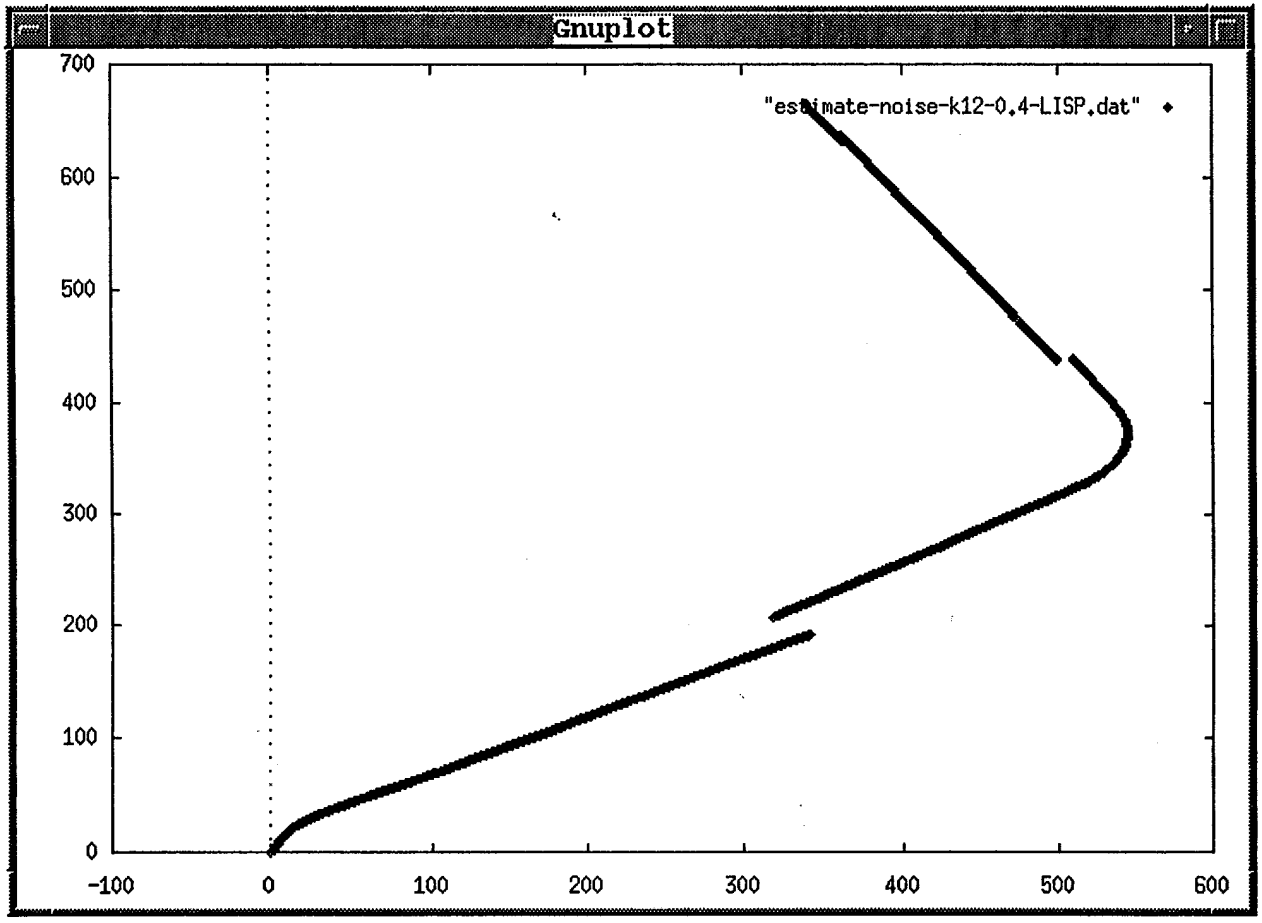


Figure A-34: New filter version, added noise, LISP, K1: 0.4; K2: 0.4; K3: 0.5; K4: 0.7

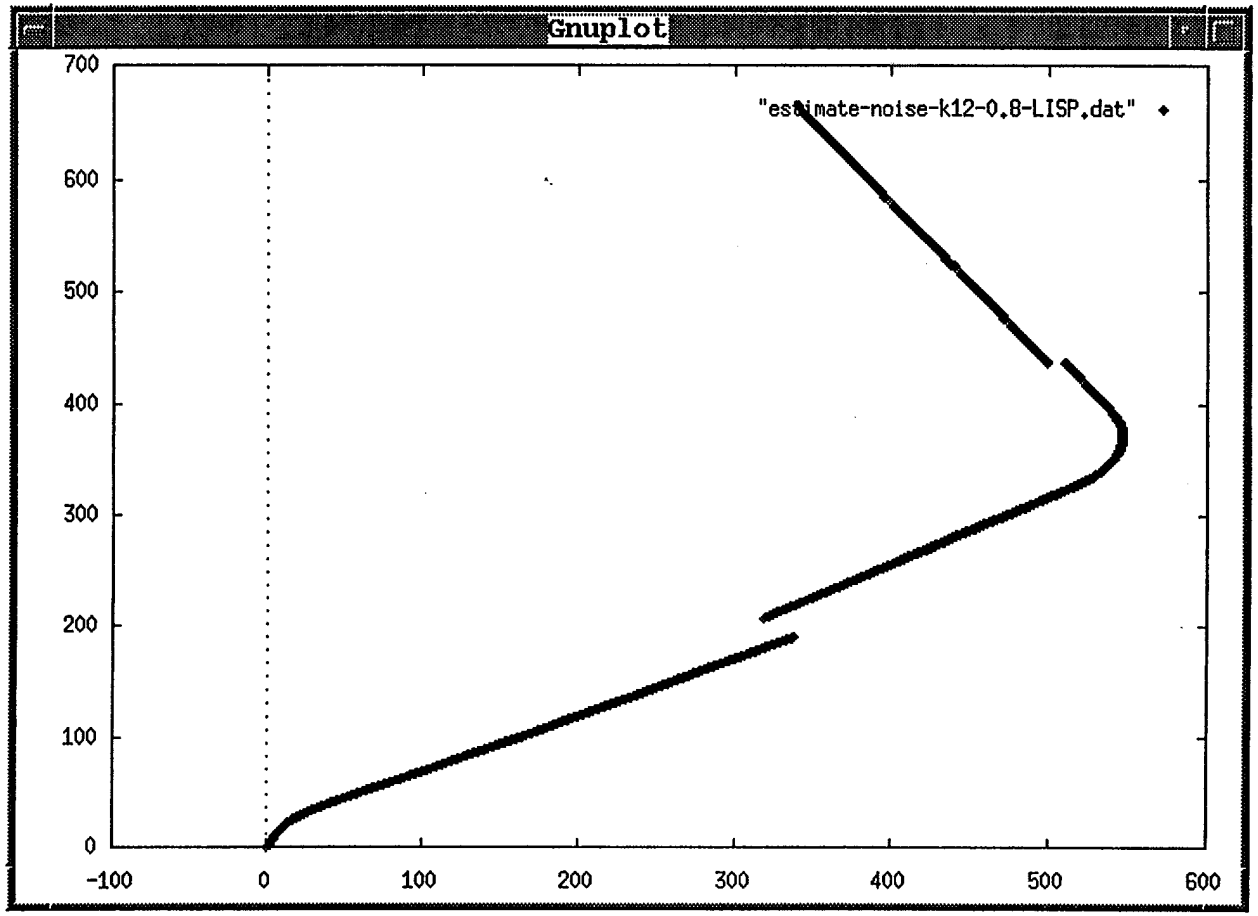


Figure A-35: New filter version, added noise, LISP, K1: 0.8; K2: 0.8; K3: 0.5; K4: 0.7

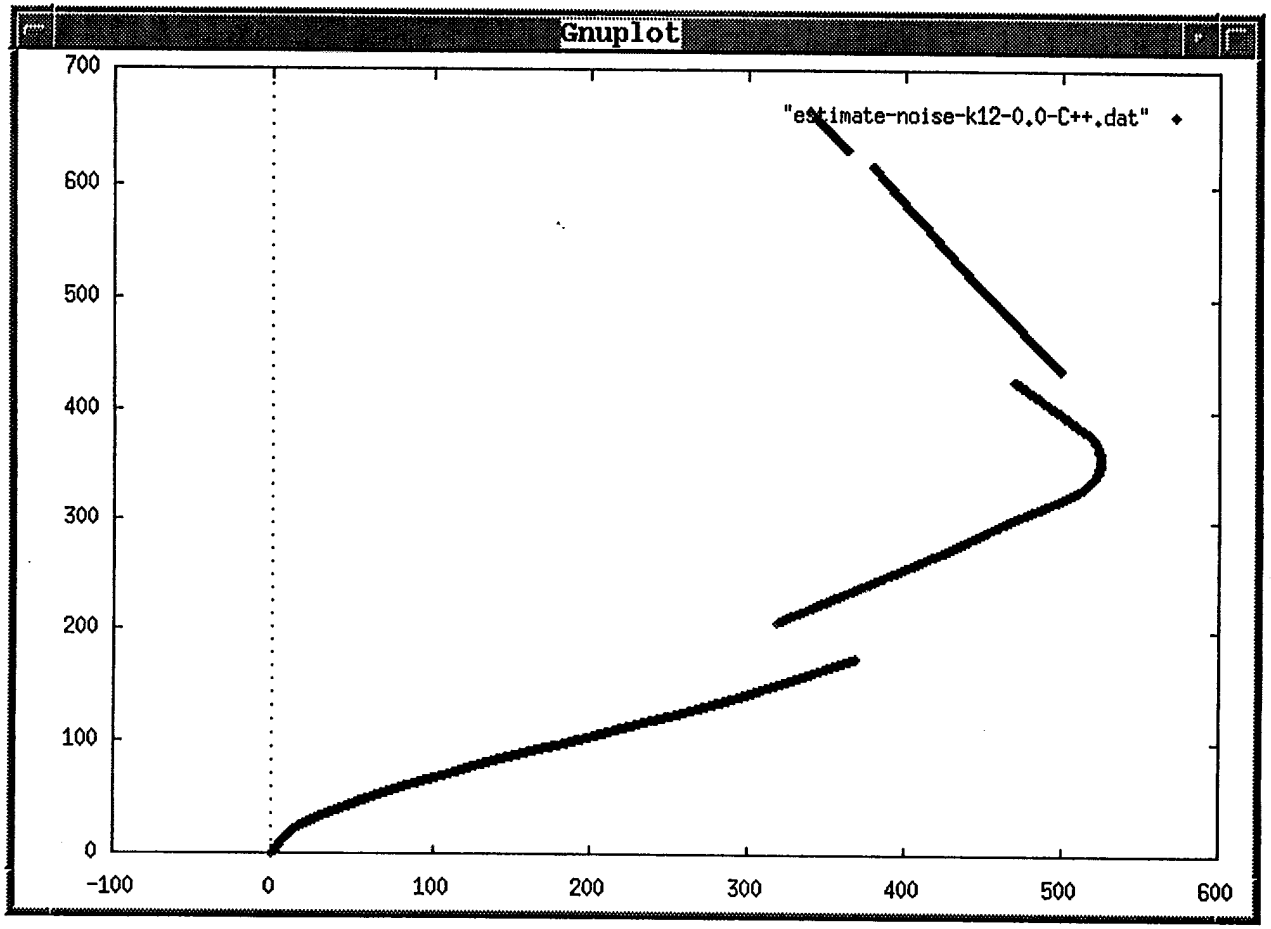


Figure A-36: New filter version, added noise, C++, K1: 0.0; K2: 0.0; K3: 0.5; K4: 0.7

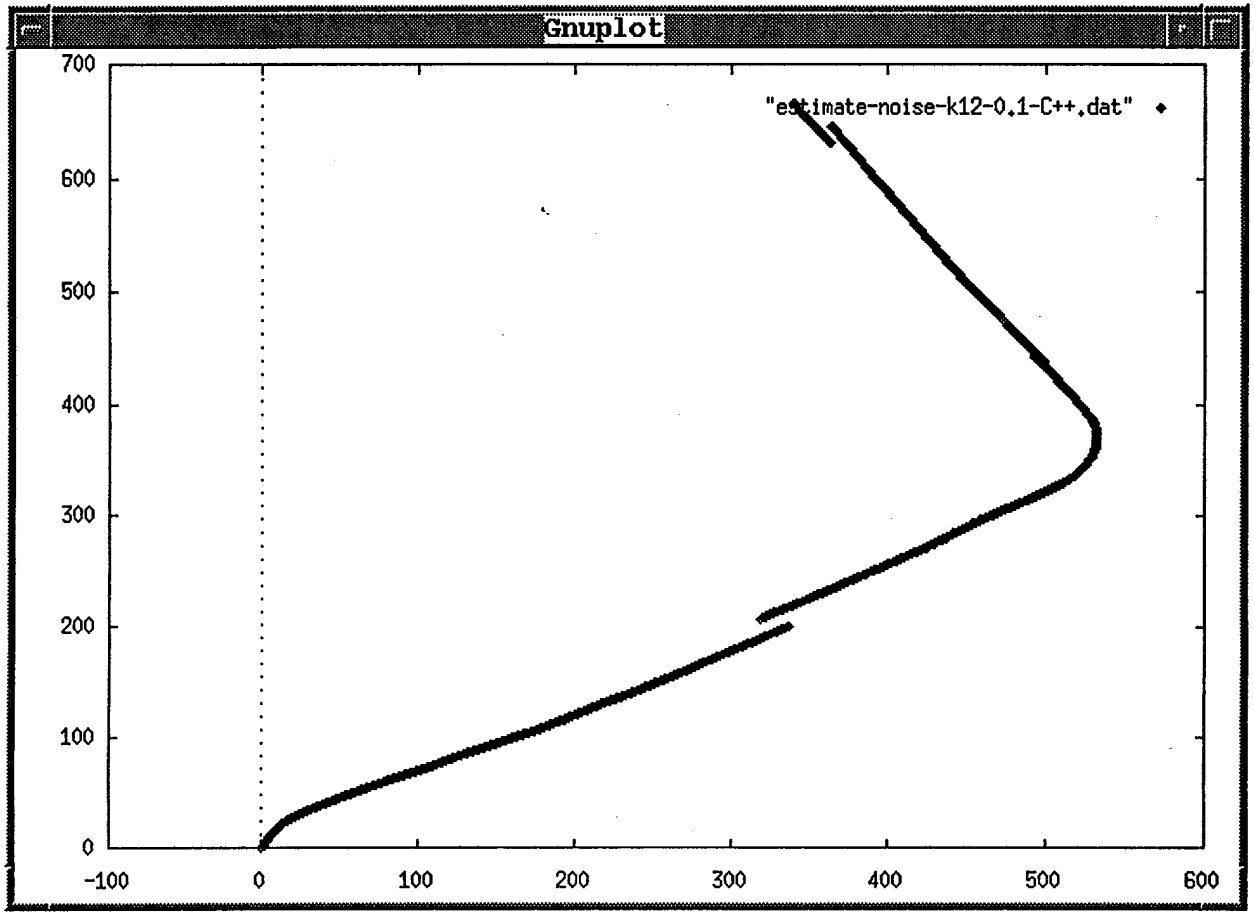


Figure A-37: New filter version, added noise, C++, K1: 0.1; K2: 0.1; K3: 0.5; K4: 0.7

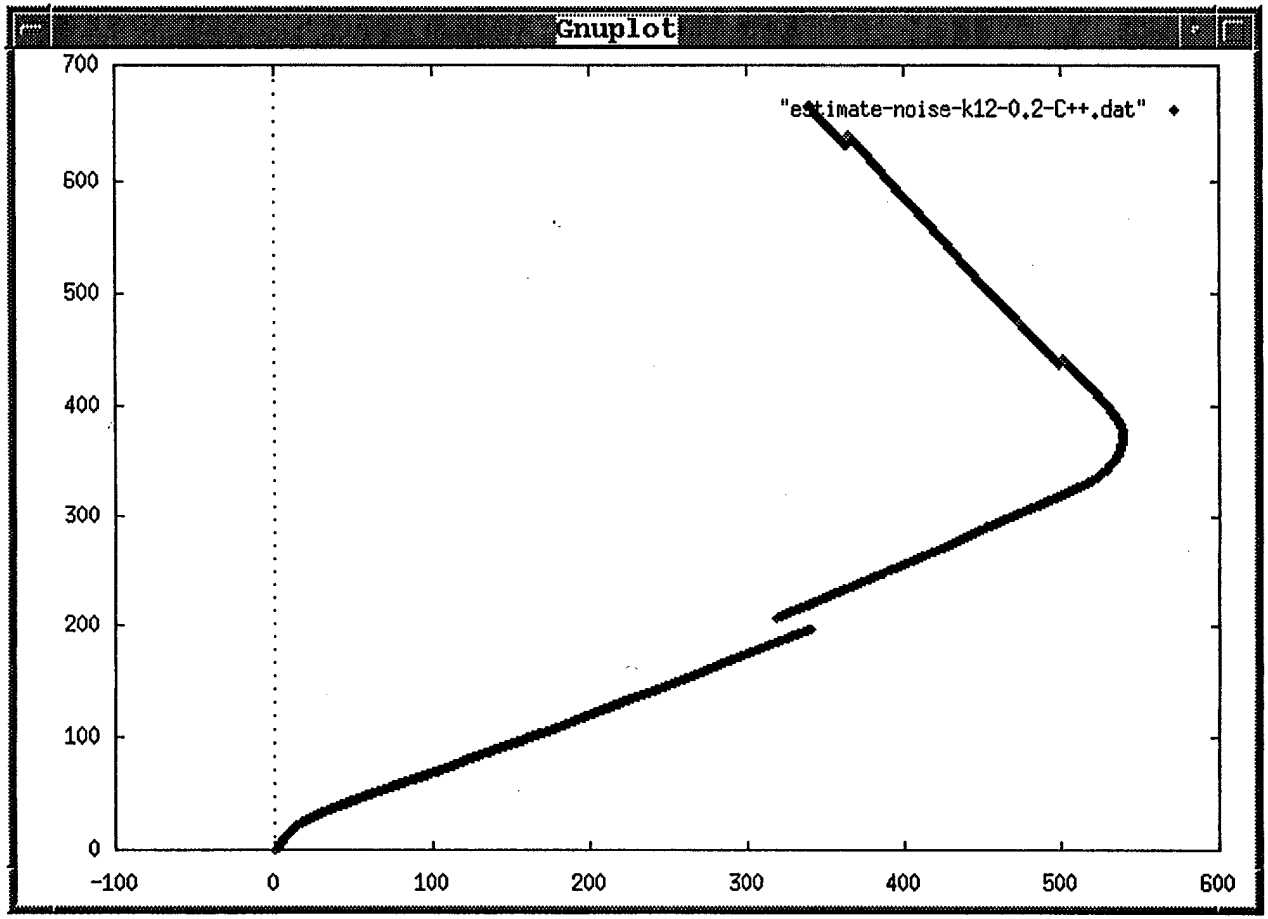


Figure A-38: New filter version, added noise, C++, K1: 0.2; K2: 0.2; K3: 0.5; K4: 0.7

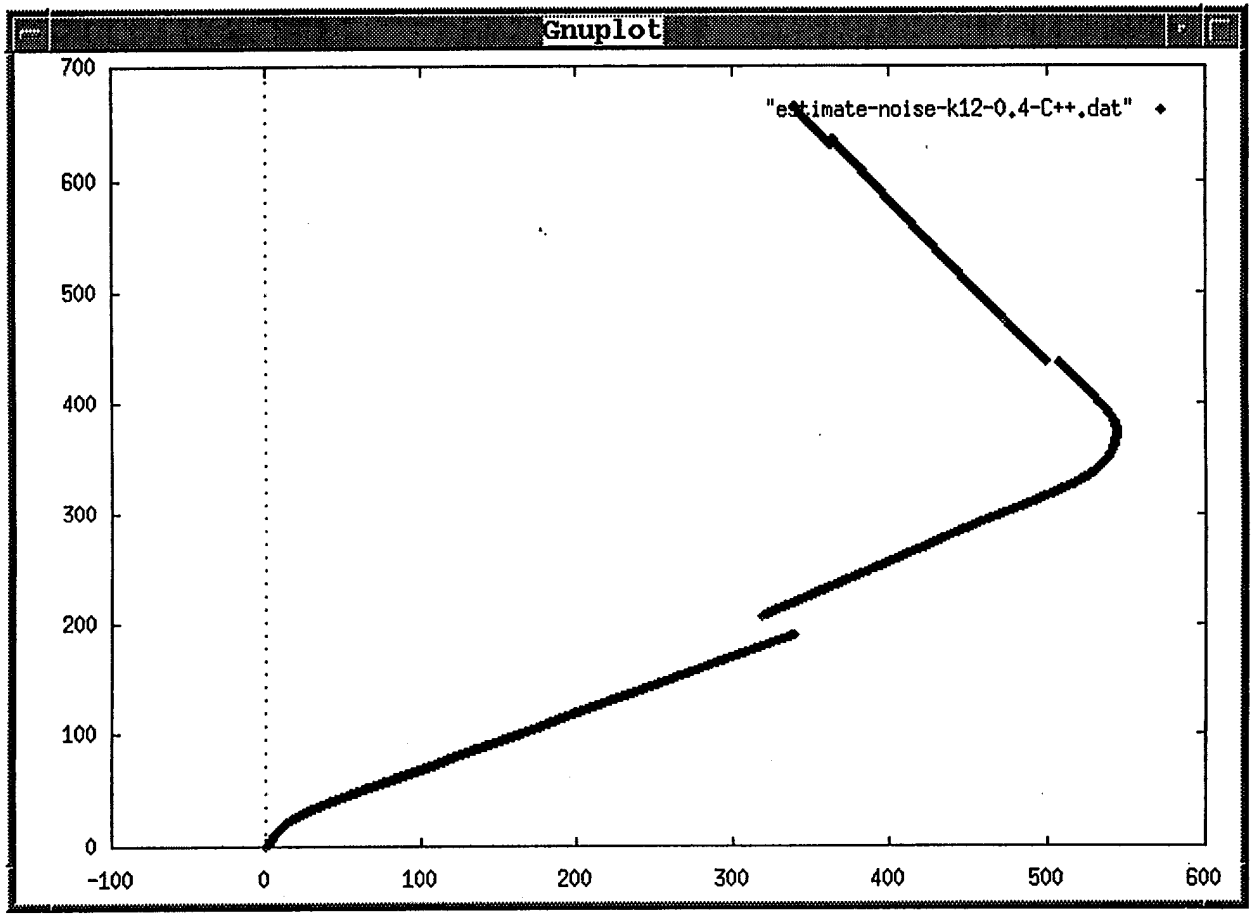


Figure A-39: New filter version, added noise, C++, K1: 0.4; K2: 0.4; K3: 0.5; K4: 0.7

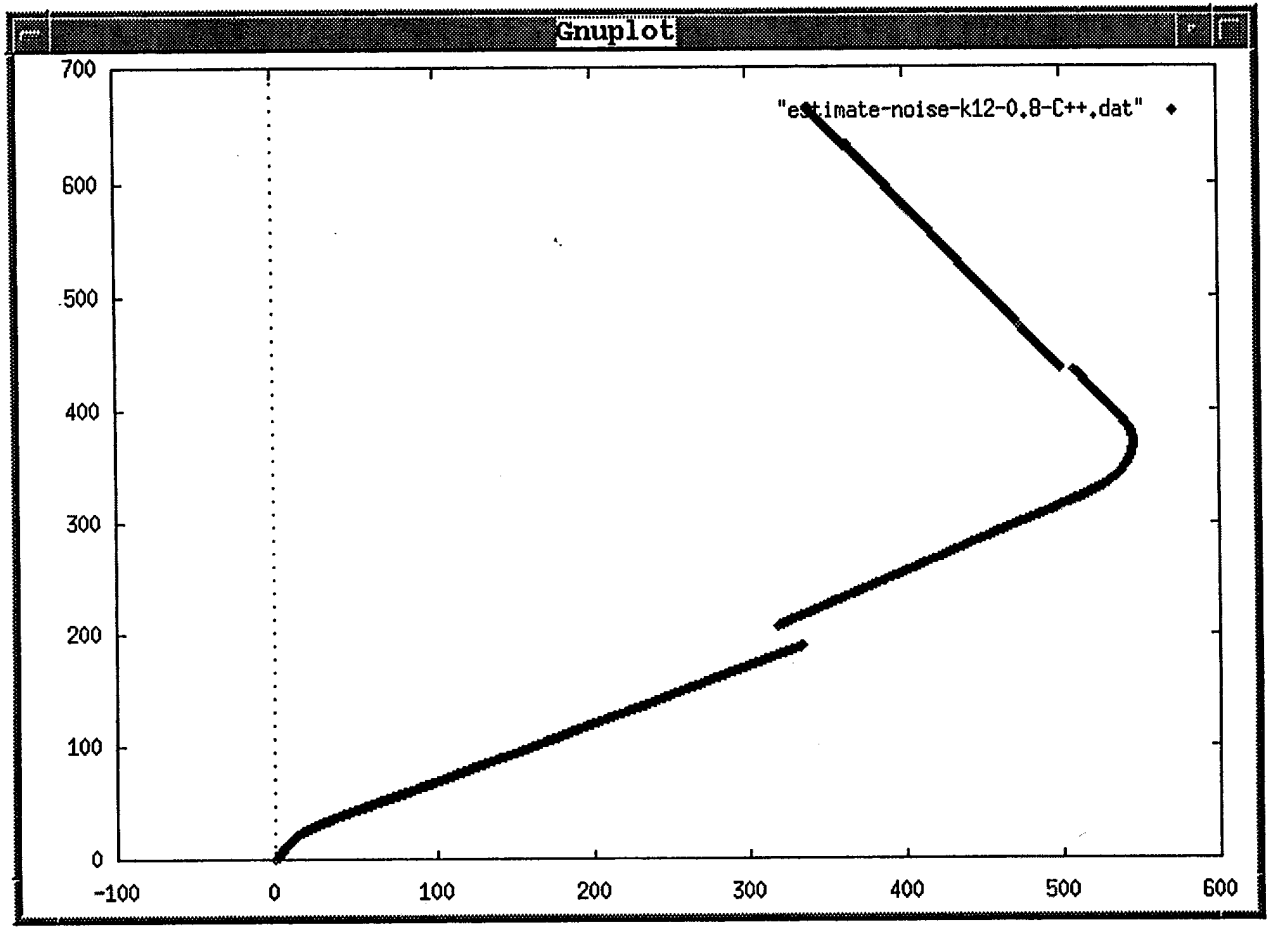


Figure A-40: New filter version, added noise, C++, K1: 0.8; K2: 0.8; K3: 0.5; K4: 0.7

APPENDIX B: SOURCE CODE (LISP)

A. PERFECT-AUTOPILOT.CL

```
;This code defines a submarine autopilot which steers the body axes of the
;submarine to a desired orientation with no time delay. It also assumes no
;sideslip angle and no angle of attack. Longitudinal speed follows commanded
;speed with a first order time lag.

;Code written by R. B. McGhee, Naval Postgraduate School, mcghee@cs.nps.navy.mil,
;Code modified by R. Steven, Naval Postgraduate School
;in Allegro Common Lisp, 1994 Release.

;(load "robot-kinematics.cl")
;(load "euler-angle-rigid-body.cl")
;(load "strobe-camera.cl")

(defclass perfect-autopilot ()
  ((vehicle-name ; This is the name of an instance of the rigid-body class.
    :accessor vehicle-name)
   (current-trajectory-segment
    :accessor current-trajectory-segment)
   (current-time
    :initform 0
    :accessor current-time)
   (longitudinal-acceleration-gain
    :accessor longitudinal-acceleration-gain)
   (dive-angle-error-gain
    :initform .8
    :accessor dive-angle-error-gain)
   (depth-error-gain
    :initform -0.08
    :accessor depth-error-gain)
   (trajectory-segment-list ;This is a list of lists. Each list contains
    :accessor trajectory-segment-list))) ;start-time and commanded speed,
;           heading-rate, depth, and roll-rate. Last segment is end-time
;           followed by nil.
```

```
(defmethod initialize-4 ((autopilot perfect-autopilot) vehicle gain trajectory)
  (setf (longitudinal-acceleration-gain autopilot) gain
        (trajectory-segment-list autopilot) (rest trajectory)
        (current-trajectory-segment autopilot) (first trajectory)
        (vehicle-name autopilot) vehicle))

(defmethod update-segment ((autopilot perfect-autopilot) time)
  (if (and (not (null (second (current-trajectory-segment autopilot))))
           (>= time (caar (trajectory-segment-list autopilot))))
      (setf (current-trajectory-segment autopilot)
            (pop (trajectory-segment-list autopilot)))))

(defmethod commanded-velocity ((autopilot perfect-autopilot) delta-t)
  (setf (current-time autopilot) (+ (current-time autopilot) delta-t))
  (update-segment autopilot (current-time autopilot))
  (if (second (current-trajectory-segment autopilot))
      (list (+ (first (velocity (vehicle-name autopilot)))
               (* (longitudinal-acceleration autopilot) delta-t))
               0 0
               (fifth (current-trajectory-segment autopilot))
               (dive-angle-rate autopilot)
               (third (current-trajectory-segment autopilot)))))

(defmethod longitudinal-acceleration ((autopilot perfect-autopilot))
  (* (longitudinal-acceleration-gain autopilot)
     (- (second (current-trajectory-segment autopilot))
        (first (velocity (vehicle-name autopilot))))))

(defmethod dive-angle-rate ((autopilot perfect-autopilot))
  (* (dive-angle-error-gain autopilot)
     (- (desired-dive-angle autopilot)
        (fifth (posture (vehicle-name autopilot))))))

(defmethod desired-dive-angle ((autopilot perfect-autopilot))
  (* (depth-error-gain autopilot)
     (- (fourth (current-trajectory-segment autopilot))
     (third (posture (vehicle-name autopilot))))))
```

```
(defmethod move-vehicle ((autopilot perfect-autopilot) delta-t)
  (setf (velocity (vehicle-name autopilot))
        (commanded-velocity autopilot delta-t))
  (when (second (velocity (vehicle-name autopilot)))
    (update-posture (vehicle-name autopilot) delta-t)
    (setf (H-matrix (vehicle-name autopilot))
          (homogeneous-transform (sixth (posture (vehicle-name autopilot)))
                                 (fifth (posture (vehicle-name autopilot)))
                                 (fourth (posture (vehicle-name autopilot)))
                                 (first (posture (vehicle-name autopilot)))
                                 (second (posture (vehicle-name autopilot)))
                                 (third (posture (vehicle-name autopilot)))))
    (transform-node-list (vehicle-name autopilot))))

(defmethod accelerometer-output ((autopilot perfect-autopilot))
  (let ( (longitudinal-velocity (first (velocity (vehicle-name autopilot))))
         (pitch-angle (fifth (posture (vehicle-name autopilot))))
         (roll-angle (fourth (posture (vehicle-name autopilot))))
         (pitch-rate (fifth (velocity (vehicle-name autopilot))))
         (yaw-rate (sixth (velocity (vehicle-name autopilot)))))
       (list (+ (longitudinal-acceleration autopilot)
                (* *gravity* (sin pitch-angle)))
             (- (* longitudinal-velocity yaw-rate)
                (* *gravity* (cos pitch-angle) (sin roll-angle)))
             (+ (- (* longitudinal-velocity pitch-rate))
                (- (* *gravity* (cos pitch-angle) (cos roll-angle)))))))

(defmethod mission-data ((autopilot perfect-autopilot))
  (append (IMU-data autopilot) (list (compass-heading autopilot))
          (list (water-speed autopilot)) (true-position autopilot)))

(defmethod IMU-data ((autopilot perfect-autopilot))
  (cons (current-time autopilot)
        (append (accelerometer-output autopilot)
                (angular-rate-output autopilot))))

(defmethod angular-rate-output ((autopilot perfect-autopilot))
  (cons (fourth (velocity (vehicle-name autopilot)))
        (cons (fifth (velocity (vehicle-name autopilot)))
              (list (sixth (velocity (vehicle-name autopilot)))))))

(defmethod water-speed ((autopilot perfect-autopilot))
  (first (velocity (vehicle-name autopilot))))
```

```
(defmethod compass-heading ((autopilot perfect-autopilot))
   (sixth (posture (vehicle-name autopilot))))

(defmethod true-position ((autopilot perfect-autopilot))
   (list  (first (posture (vehicle-name autopilot)))
          (second (posture (vehicle-name autopilot)))
          (third (posture (vehicle-name autopilot)))
          (fourth (posture (vehicle-name autopilot)))
          (fifth (posture (vehicle-name autopilot)))
          (sixth (posture (vehicle-name autopilot)))))

(defun initialize-mission (north-current east-current)
   (setf submarine-1 (make-instance 'rigid-body))
   (setf (medium-north-velocity submarine-1) north-current)
   (setf (medium-east-velocity submarine-1) east-current)
   (setf autopilot-1 (make-instance 'perfect-autopilot))
   (initialize-4 autopilot-1 submarine-1 1 *trajectory*)
   (move-vehicle autopilot-1 0)
;(setf camera-1 (make-instance 'strobe-camera))
;(move camera-1 0 (- (/ pi 2)) 0 0 0 -60)
   (setf (posture submarine-1) '(0 0 0 0 0 0))
   (setf (velocity submarine-1) '(0 0 0 0 0 0)))
;(take-picture camera-1 submarine-1))

(defun execute-mission ()
   (do* ( (mission-data (list (mission-data autopilot-1))
                        (cons (mission-data autopilot-1) mission-data))
          (new-node-list  (move-vehicle autopilot-1 .1)
                          (move-vehicle autopilot-1 .1)))
        (  (not (second (velocity (vehicle-name autopilot-1))))
           (setf *mission-data* (reverse mission-data)))))
;(take-picture camera-1 submarine-1)))

(defun mission-file (north-current east-current)
   (initialize-mission north-current east-current)
   (execute-mission)
   (tag-for-GPS-fix *mission-data*)
   (add-noise-to-mission-data *mission-data*)
   (write-mission-data-to-file))
```

```
(setf *trajectory* '((0 3 0 0 0) (1 3 .1 0 0) (2 3 .1 0 0) (11 3 0 .5 0)
                     (41 3 0 .5 .09) (43 3 0 .5 -.09) (45 3 0 .5 0)
                     (90 3 0 0 0) (120 3 0 .5 0) (150 3 -.1 .5 0)
                     (170 3 0 .5 0) (190 3 0 0 0) (220 3 0 .5 0)
                     (230 3 0 .5 -.09) (232 3 0 .5 .09) (234 3 0 .5 0)
                     (280 3 0 0 0) (300 nil)))
; components are: time, speed, heading-rate, depth, and roll-rate

(defun write-mission-data-to-file ()
   (with-open-file (output-stream "traject.dat" :direction :output)
   (dolist (element *mission-data* 'Moin!)
      (format output-stream
       "~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~f ~t ~d ~%"
         (first element) (second element) (third element)   (fourth element)
         (fifth element) (sixth element)  (seventh element) (eighth element)
         (ninth element) (nth 9 element) (nth 10 element)
         (nth 11 element) (nth 12 element) (nth 13 element)
         (nth 14 element) (nth 15 element)))))

(defun write-filtered-data-to-file ()
   (with-open-file (output-stream "estimate.dat" :direction :output)
   (dolist (element *estimated-trajectory* 'Moin!)
      (format output-stream
         "~f ~t ~f ~t ~f ~t ~f ~%"
         (first element) (nth 9 element) (nth 10 element)))))

(defun tag-for-GPS-fix (dataList)
   (let ((new-data-list nil) (new-element nil))
      (dolist (element dataList 'Moin!)
         (if (and (< (nth 11 element) .25)
                  (< (multiple-value-bind (part1 part2) (truncate (first element)) part2) .1))
            (setf new-element (append element (list 1)))
            (setf new-element (append element (list 0))))
         (setf new-data-list (append new-data-list (list new-element))))
   (setf *mission-data* new-data-list)))
```

```
(defun add-noise-to-mission-data (dataList)
  (let ((new-data-list nil) (new-element nil))
    (dolist (element dataList 'Moin!)
      (setf new-element
        (list  (first element)
               (+ (second element) (- .3 (random .6)))
               (+ (third element) (- .3 (random .6)))
               (+ (fourth element) (- .6 (random 1.2)))
               (+ (fifth element) (- .005 (random .01)))
               (+ (sixth element) (- .005 (random .01)))
               (+ (seventh element) (- .005 (random .01)))
               (+ (eighth element) (- .034 (random .068)))
               (ninth element) (nth 9 element) (nth 10 element)
               (nth 11 element) (nth 12 element) (nth 13 element)
               (nth 14 element) (nth 15 element)))
      (setf new-data-list (append new-data-list (list new-element))))
    (setf *mission-data* new-data-list)))
```

B. NAVIGATION-FILTER.CL

;This code defines a navigation filter that takes three accelerations $(\ddot{x}_a,\ \ddot{y}_a,\ \ddot{z}_a)$,
;three angular-rates $(p,\ q,\ r)$, speed through the water (u_w), and compass
;heading (ψ_c) as input and computes a north and east postition $(x_e,\ y_e)$.
;The computed position can be corrected by a GPS fix procedure.

;Code written by R. B. McGhee, Naval Postgraduate School, mcghee@cs.nps.navy.mil,
;in Allegro Common Lisp, 1994 Release.

```
; (load "perfect-autopilot.cl")

(defclass navigation-filter ()
  ((estimated-roll-angle
   :accessor estimated-roll-angle)
   (estimated-pitch-angle
   :accessor estimated-pitch-angle)
   (estimated-heading
   :accessor estimated-heading)
   (roll-error-gain
   :accessor roll-error-gain)
   (pitch-error-gain
   :accessor pitch-error-gain)
   (heading-error-gain
   :accessor heading-error-gain)
   (estimated-roll-rate-bias
   :accessor estimated-roll-rate-bias)
   (estimated-pitch-rate-bias
   :accessor estimated-pitch-rate-bias)
   (estimated-yaw-rate-bias
   :accessor estimated-yaw-rate-bias)
   (rate-bias-gain
   :accessor rate-bias-gain)
   (estimated-north-velocity
   :accessor estimated-north-velocity)  ;Velocity is relative to water.
   (estimated-east-velocity
   :accessor estimated-east-velocity)   ;Velocity is relative to water.
   (velocity-error-gain
   :accessor velocity-error-gain)
   (estimated-north-position
   :accessor estimated-north-position)
   (estimated-east-position
   :accessor estimated-east-position)
   (estimated-north-current
```

```
  :initform 0
  :accessor estimated-north-current)
  (estimated-east-current
  :initform 0
  :accessor estimated-east-current)
  (current-error-gain
  :accessor current-error-gain)
  (last-gps-fix-time
  :initform 0
  :accessor last-gps-fix-time)
  (time-stamp
  :accessor time-stamp)))

(defmethod angle-derivatives ((filter navigation-filter) measurement-vector)
  ;Order is estimated phidot, estimated thetadot, estimated psidot.
  (let* ( (ax (second measurement-vector)) (ay (third measurement-vector))
          (az (fourth measurement-vector)) (p (fifth measurement-vector))
          (q (sixth measurement-vector)) (r (seventh measurement-vector))
          (psi (eighth measurement-vector))
          (phie (estimated-roll-angle filter))
          (thetae (estimated-pitch-angle filter))
          (psie (estimated-heading filter))
          (pb (estimated-roll-rate-bias filter))
          (qb (estimated-pitch-rate-bias filter))
          (rb (estimated-yaw-rate-bias filter))
          (T-matrix (body-rate-to-euler-rate-matrix psie thetae phie))
          (euler-rates (post-multiply T-matrix (list (- p pb) (- q qb) (- r rb))))
          (thetaa (asin (/ ax *gravity*)))
          (phia (- (asin (/ ay (* *gravity* (cos thetae)))))))
      (list  (+ (first euler-rates)(* (roll-error-gain filter) (- phia phie)))
             (+ (second euler-rates)(* (pitch-error-gain filter) (- thetaa thetae)))
             (+ (third euler-rates)(* (heading-error-gain filter) (- psi psie))))))

(defmethod velocity-derivatives ((filter navigation-filter) measurement-vector)
  ;Velocity is relative to water under constant current assumption.
  (let* ( (ax (second measurement-vector)) (ay (third measurement-vector))
          (az (fourth measurement-vector)) (p (fifth measurement-vector))
          (q (sixth measurement-vector)) (r (seventh measurement-vector))
          (psi (eighth measurement-vector)) (uw (ninth measurement-vector))
          (phie (estimated-roll-angle filter))
          (thetae (estimated-pitch-angle filter))
          (psie (estimated-heading filter))
          (R-matrix (rotation-matrix psie thetae phie))
          (xa (- ax (* *gravity* (sin thetae))))
```

```
        (ya (+ ay (* *gravity* (sin phie) (cos thetae))))
        (za (+ az (* *gravity* (cos phie) (cos thetae))))
        (xdote (estimated-north-velocity filter))
        (ydote (estimated-east-velocity filter))
        (k3 (velocity-error-gain filter))
        (linear-acceleration (post-multiply R-matrix (list xa ya za)))
        (water-relative-velocity (post-multiply R-matrix (list uw 0 0))))
      (list  (+ (first linear-acceleration) (* k3 (- (first water-relative-velocity) xdote)))
             (+ (second linear-acceleration) (* k3 (- (second water-relative-velocity) ydote))))))

(defmethod rate-bias-derivatives ((filter navigation-filter) measurement-vector)
  (let ((kb (rate-bias-gain filter)))
      (list  (* kb  (- (fifth measurement-vector)
                       (estimated-roll-rate-bias filter)))
             (* kb  (- (sixth measurement-vector)
                       (estimated-pitch-rate-bias filter)))
             (* kb  (- (seventh measurement-vector)
                       (estimated-yaw-rate-bias filter))))))


(defun initialize-filter (angle-error-gain rate-bias-gain current-error-gain
                          estimated-north-current estimated-east-current)
  (setf filter-1 (make-instance 'navigation-filter) (time-stamp filter-1) 0)
  (setf  (estimated-roll-angle filter-1) 0 (estimated-pitch-angle filter-1) 0
         (estimated-heading filter-1) 0 (estimated-north-velocity filter-1) 0
         (estimated-east-velocity filter-1) 0 (velocity-error-gain filter-1) .5
         (estimated-north-position filter-1) 0
         (estimated-east-position filter-1) 0
         (roll-error-gain filter-1) angle-error-gain
         (pitch-error-gain filter-1) angle-error-gain
         (heading-error-gain filter-1) angle-error-gain
         (rate-bias-gain filter-1) rate-bias-gain (time-stamp filter-1) 0
         (current-error-gain filter-1) current-error-gain
         (estimated-north-current filter-1) estimated-north-current
         (estimated-east-current filter-1) estimated-east-current
         (estimated-roll-rate-bias filter-1) 0
         (estimated-pitch-rate-bias filter-1) 0
         (estimated-yaw-rate-bias filter-1) 0))
```

```
(defmethod update-list ((filter navigation-filter) measurement-vector)
  (let* (  (current-time (first measurement-vector))
           (delta-t (- current-time (time-stamp filter))))
       (append  (list delta-t)
                (scalar-multiply delta-t (rate-bias-derivatives filter measurement-vector))
                (scalar-multiply delta-t (angle-derivatives filter measurement-vector))
                (scalar-multiply delta-t (velocity-derivatives filter measurement-vector))
                (list (* delta-t (+  (estimated-north-velocity filter)
                                     (estimated-north-current filter))))
                (list (* delta-t (+  (estimated-east-velocity filter)
                                     (estimated-east-current filter)))))))

(defmethod update-filter ((filter navigation-filter) measurement-vector)
  (let ((update-list (update-list filter measurement-vector)))
      (setf  (time-stamp filter) (+ (first update-list) (time-stamp filter))
             (estimated-roll-rate-bias filter)
             (+ (second update-list) (estimated-roll-rate-bias filter))
             (estimated-pitch-rate-bias filter)
             (+ (third update-list) (estimated-pitch-rate-bias filter))
             (estimated-yaw-rate-bias filter)
             (+ (fourth update-list) (estimated-yaw-rate-bias filter))
             (estimated-roll-angle filter)
             (+ (fifth update-list) (estimated-roll-angle filter))
             (estimated-pitch-angle filter)
             (+ (sixth update-list) (estimated-pitch-angle filter))
             (estimated-heading filter)
             (+ (seventh update-list) (estimated-heading filter))
             (estimated-north-velocity filter)
             (+ (eighth update-list) (estimated-north-velocity filter))
             (estimated-east-velocity filter)
             (+ (ninth update-list) (estimated-east-velocity filter))
             (estimated-north-position filter)
             (+ (tenth update-list) (estimated-north-position filter))
             (estimated-east-position filter)
             (+ (nth 10 update-list) (estimated-east-position filter)))))

(defmethod state-vector ((filter navigation-filter))
  (list  (time-stamp filter) (estimated-roll-rate-bias filter)
         (estimated-pitch-rate-bias filter) (estimated-yaw-rate-bias filter)
         (estimated-roll-angle filter) (estimated-pitch-angle filter)
         (estimated-heading filter) (estimated-north-velocity filter)
         (estimated-east-velocity filter) (estimated-north-position filter)
         (estimated-east-position filter) (estimated-north-current filter)
         (estimated-east-current filter)))
```

```
(defmethod new-gps-fixp ((filter navigation-filter) measurement-vector)
   (and (not (= (time-stamp filter) 0)) (= 1 (nth 15 measurement-vector))))

(defmethod gps-reset ((filter navigation-filter) measurement-vector)
   (let* ( (gps-time-interval (-  (first measurement-vector)
                                  (last-gps-fix-time filter)))
           (north-position (tenth measurement-vector))
           (east-position (nth 10 measurement-vector))
           (k4 (current-error-gain filter))
           (north-position-error (-  north-position
                                     (estimated-north-position filter)))
           (east-position-error (-  east-position
                                    (estimated-east-position filter)))
           (north-current-increment
              (/ (* k4 north-position-error) gps-time-interval))
           (east-current-increment
              (/ (* k4 east-position-error) gps-time-interval)))
         (setf (last-gps-fix-time filter) (first measurement-vector)
           (estimated-north-position filter) north-position
           (estimated-east-position filter) east-position
           (estimated-north-current filter) (+  north-current-increment
                                                (estimated-north-current filter))
           (estimated-east-current filter) (+  east-current-increment
                                               (estimated-east-current filter)))))

(defmethod estimated-trajectory ((filter navigation-filter) mission-data)
   (do (  (data mission-data (rest data))
          (trajectory nil (cons (state-vector filter) trajectory)))
       (  (null data) (reverse trajectory))
       (update-filter filter (first data))
       (if (new-gps-fixp filter (first data)) (gps-reset filter (first data)))))

(defun test-filter ()
   (initialize-filter (first *filter-parameter-list*)
                      (second *filter-parameter-list*)
                      (fifth *filter-parameter-list*)
                      (sixth *filter-parameter-list*)
                      (seventh *filter-parameter-list*))
   (mission-file  (third *filter-parameter-list*)
                  (fourth *filter-parameter-list*))
   (setf *estimated-trajectory* (estimated-trajectory filter-1 *mission-data*))
   (write-filtered-data-to-file))

(setf *filter-parameter-list* '(.6 0 .5 5 .7 0 0))
```

```
;Components are: angle-error-gain, rate-bias-gain, north-current, east-current,
;current-error-gain, estimated-north-current, estimated-east-current.
```

C. EULER-ANGLE-RIGID-BODY.CL

```
;This code models a single rigid body moving in an irrotational medium which has
;a constant linear velocity (current or wind) relative to an inertial frame
;(flat earth).

;Code written by R. B. McGhee, Naval Postgraduate School, mcghee@cs.nps.navy.mil,
;in Allegro Common Lisp, 1994 Release.

(defclass rigid-body ()
 ((posture              ;The vector (xe ye ze phi theta psi).
   :initform '(0 0 0 0 0 0)
   :initarg :posture
   :accessor posture)
  (posture-rate ;The vector (xe-dot ye-dot ze-dot phi-dot theta-dot psi-dot).
   :initarg :posture-rate
   :accessor posture-rate)
  (velocity             ;The six-vector (u v w p q r) in body coordinates.
   :initform '(1 1 1 .1 .1 .1)
   :initarg :velocity
   :accessor velocity)
  (velocity-growth-rate  ;The vector (u-dot v-dot w-dot p-dot q-dot r-dot).
   :accessor velocity-growth-rate)
  (forces-and-torques    ;The vector (Fx Fy Fz L M N) in body coordinates.
   :initform (list 0 0 (- *gravity*) 0 0 0)
   :accessor forces-and-torques)
  (moments-of-inertia    ;The vector (Ix Iy Iz) in principal axis coordinates.
   :initform '(1 1 1)
   :initarg :moments-of-inertia
   :accessor moments-of-inertia)
  (mass
   :initform 1
   :initarg :mass
   :accessor mass)
  (node-list    ;(x y z 1) in body coord for each node. Starts with (0 0 0 1).
   :initform '((0 0 0 1) (4 0 0 1) (2 0 0 1) (-4 0 0 1) (-5 0 -2 1)
             (-6 -1.5 -2 1) (-6 1.5 -2 1) (-2 6 -2 1) (-2 -6 -2 1)
             (-2 0 0 1)) ;Defines a simple "airplane" as default rigid body.
   :initarg :node-list
```

```
    :accessor node-list)
   (polygon-list
    :initform '((1 3 4 5 4 3) (4 6) (7 2 8 9))
    :initarg :polygon-list
    :accessor polygon-list)
   (transformed-node-list  ;(x y z 1) in earth coord for each node in node-list.
    :accessor transformed-node-list)
   (H-matrix
    :initform (unit-matrix 4)
    :accessor H-matrix)
   (medium-north-velocity
    :initform 0
    :accessor medium-north-velocity)
   (medium-east-velocity
    :initform 0
    :accessor medium-east-velocity)
   (time-stamp
    :accessor time-stamp)))

(defmethod initialize ((body rigid-body))
   (setf (transformed-node-list body) (node-list body))
   (setf (velocity-growth-rate body) (update-velocity-growth-rate body))
   (setf (posture-rate body) (earth-velocity body))
   (setf (time-stamp body) (get-internal-real-time)))

(defmethod move ((body rigid-body) azimuth elevation roll x y z)
   (setf (posture body) (list x y z roll elevation azimuth))
   (setf (H-matrix body)
      (homogeneous-transform azimuth elevation roll x y z))
   (transform-node-list body))

(defmethod get-delta-t ((body rigid-body)) 0.1)
;  (let* ((new-time (get-internal-real-time))
;        (delta-t (/ (- new-time (time-stamp body)) 1000)))
;   (setf (time-stamp body) new-time)
;   delta-t))
```

```
(defmethod update-rigid-body ((body rigid-body))     ;Euler integration.
  (let* ((delta-t (get-delta-t body)))
    (update-posture body delta-t)
    (setf (H-matrix body) (homogeneous-transform (sixth (posture body))
      (fifth (posture body)) (fourth (posture body)) (first (posture body))
      (second (posture body)) (third (posture body))))
    (transform-node-list body)
    (update-velocity body delta-t)
    (update-velocity-growth-rate body)))

(defmethod update-velocity-growth-rate ((body rigid-body))
  (setf (velocity-growth-rate body)        ;Assumes principal axis coordinates with
    (multiple-value-bind                   ;origin at center of gravity of body.
      (Fx Fy Fz L M N   u v w p q r  Ix Iy Iz) ;Declares local variables.
      (values-list                         ;Values assigned.
        (append
          (forces-and-torques body) (velocity body) (moments-of-inertia body)))
      (list  (+  (* v r) (* -1 w q) (/ Fx (mass body))
                 (* *gravity* (first (third (H-matrix body)))))
             (+  (* w p) (* -1 u r) (/ Fy (mass body))
                 (* *gravity* (second (third (H-matrix body)))))
             (+  (* u q) (* -1 v p) (/ Fz (mass body))
                 (* *gravity* (third (third (H-matrix body)))))
             (/ (+ (* (- Iy Iz) q r) L) Ix)
             (/ (+ (* (- Iz Ix) r p) M) Iy)
             (/ (+ (* (- Ix Iy) p q) N) Iz)))))  ;Value returned.

(defmethod update-velocity ((body rigid-body) delta-t) ;Euler integration.
  (setf (velocity body)
    (vector-add  (velocity body)
                 (scalar-multiply delta-t (velocity-growth-rate body)))))

(defmethod update-posture ((body rigid-body) delta-t) ;Euler integration.
  (setf (posture-rate body) (earth-velocity body))
  (setf (posture body)
    (vector-add (posture body) (scalar-multiply delta-t (posture-rate body)))))

(defmethod transform-node-list ((body rigid-body))
  (setf (transformed-node-list body)
    (mapcar #'(lambda (node-location)
                (post-multiply (H-matrix body) node-location))
            (node-list body))))

(defconstant *gravity* 32.2185)
```

```
(defmethod earth-velocity ((body rigid-body))
   (let* ( (linear-velocity (firstn 3 (velocity body)))
           (rotational-velocity (cdddr (velocity body)))
           (posture (posture body))
           (R-matrix (rotation-matrix  (sixth posture) (fifth posture) (fourth posture)))
           (current-vector (list (medium-north-velocity body) (medium-east-velocity body) 0))
           (linear-earth-velocity
               (vector-add current-vector (post-multiply R-matrix linear-velocity)))
           (T-matrix (body-rate-to-euler-rate-matrix (sixth posture)
                         (fifth posture) (fourth posture)))
           (rotational-earth-velocity (post-multiply T-matrix rotational-velocity)))
       (append linear-earth-velocity rotational-earth-velocity)))

(defun test-rigid-body ()
   (setf airplane-1 (make-instance 'rigid-body))
   (initialize airplane-1)
;(setf camera-1 (make-instance 'strobe-camera))
;(move camera-1 0 (- (/ pi 2)) 0 0 0 -30)
;(take-picture camera-1 airplane-1)
   (dotimes (i 20 'done) (update-rigid-body airplane-1)))
;(take-picture camera-1 airplane-1))
```

D. ROBOT-KINEMATICS.CL

```
;This code contains functions for matrix and vector computations. It also contains
;functions for translational, rotational and transformation operators.

;Code written by R. B. McGhee, Naval Postgraduate School, mcghee@cs.nps.navy.mil,
;in Allegro Common Lisp, 1994 Release.

(defun transpose (matrix)                    ;A matrix is a list of row vectors.
   (cond  ((null (cdr matrix)) (mapcar 'list (car matrix)))
          (t (mapcar 'cons (car matrix) (transpose (cdr matrix))))))

(defun dot-product (vector-1 vector-2)       ;A vector is a list of numerical atoms.
   (apply '+ (mapcar '* vector-1 vector-2)))

(defun vector-magnitude (vector) (sqrt (dot-product vector vector)))

(defun post-multiply (matrix vector)
   (cond  ((null (rest matrix)) (list (dot-product (first matrix) vector)))
          (t (cons (dot-product (first matrix) vector)
                   (post-multiply (rest matrix) vector)))))
```

```
(defun pre-multiply (vector matrix)
   (post-multiply (transpose matrix) vector))

(defun matrix-multiply (A B)                ;A and B are conformable matrices.
   (cond  ((null (cdr A)) (list (pre-multiply (car A) B)))
          (t (cons (pre-multiply (car A) B) (matrix-multiply (cdr A) B)))))

(defun chain-multiply (L)                   ;L is a list of names of conformable matrices.
   (cond  ((null (cddr L)) (matrix-multiply (eval (car L)) (eval (cadr L))))
          (t (matrix-multiply (eval (car L)) (chain-multiply (cdr L))))))

(defun cycle-left (matrix) (mapcar 'row-cycle-left matrix))

(defun row-cycle-left (row) (append (cdr row) (list (car row))))

(defun cycle-up (matrix) (append (cdr matrix) (list (car matrix))))

(defun unit-vector (one-column length)          ;Column count starts at 1.
   (do ((n length (1- n))
        (vector nil (cons (cond ((= one-column n) 1) (t 0)) vector)))
       ((zerop n) vector)))

(defun unit-matrix (size)
   (do (  (row-number size (1- row-number))
          (I nil (cons (unit-vector row-number size) I)))
       ((zerop row-number) I)))

(defun concat-matrix (A B)   ;A and B are matrices with equal number of rows.
   (cond  ((null A) B)
          (t (cons (append (car A) (car B)) (concat-matrix (cdr A) (cdr B))))))

(defun augment (matrix)
   (concat-matrix matrix (unit-matrix (length matrix))))

(defun normalize-row (row) (scalar-multiply (/ 1.0 (car row)) row))

(defun scalar-multiply (scalar vector)
   (cond  ((null vector) nil)
          (t (cons (* scalar (car vector))
                   (scalar-multiply scalar (cdr vector))))))
```

```
(defun solve-first-column (matrix)                    ;Reduces first column to (1 0 ... 0).
  (do* ((remaining-row-list matrix (rest remaining-row-list))
     (first-row (normalize-row (first matrix)))
     (answer (list first-row)
             (cons  (vector-add  (first remaining-row-list)
                                 (scalar-multiply  (- (caar remaining-row-list))
                                                   first-row))
             answer)))
    ((null (rest remaining-row-list)) (reverse answer))))

(defun vector-add (vector-1 vector-2) (mapcar '+ vector-1 vector-2))

(defun vector-subtract (vector-1 vector-2) (mapcar '- vector-1 vector-2))

(defun first-square (matrix)              ;Returns leftmost square matrix from argument.
(do ( (size (length matrix))
     (remainder matrix (rest remainder))
     (answer nil (cons (firstn size (first remainder)) answer)))
    ((null remainder) (reverse answer))))

(defun firstn (n list)
  (cond (  (zerop n) nil)
           (t (cons (first list) (firstn (1- n) (rest list))))))

(defun max-car-firstn (n list)
  (append (max-car-first (firstn n list)) (nthcdr n list)))

(defun matrix-inverse (M)
  (do (  (M1  (max-car-first (augment M))
              (cond  ((null M1) nil)                ;Abort for singular matrix.
                     (t (max-car-firstn n (cycle-left (cycle-up M1))))))
         (n (1- (length M)) (1- n)))
       (  (or (minusp n) (null M1)) (cond ((null M1) nil) (t (first-square M1))))
       (setq M1 (cond ((zerop (caar M1)) nil) (t (solve-first-column M1))))))

(defun max-car-first (L)   ;L is a list of lists. This function finds list with
  (cond  ((null (cdr L)) L) ;largest car and moves it to head of list of lists.
         (t (if (> (abs (caar L)) (abs (caar (max-car-first (cdr L))))) L
                (append (max-car-first (cdr L)) (list (car L)))))))
```

```
(defun dh-matrix (cosrotate sinrotate costwist sintwist length translate)
   (list  (list  cosrotate (- (* costwist sinrotate))
                 (* sintwist sinrotate) (* length cosrotate))
          (list  sinrotate (* costwist cosrotate)
                 (- (* sintwist cosrotate)) (* length sinrotate))
          (list 0. sintwist costwist translate) (list 0. 0. 0. 1.)))

(defun homogeneous-transform (azimuth elevation roll x y z)
   (let (  (spsi (sin azimuth)) (cpsi (cos azimuth)) (sth (sin elevation))
           (cth (cos elevation)) (sphi (sin roll)) (cphi (cos roll)))
       (list  (list  (* cpsi cth) (- (* cpsi sth sphi) (* spsi cphi))
                     (+ (* cpsi sth cphi) (* spsi sphi)) x)
              (list  (* spsi cth) (+ (* cpsi cphi) (* spsi sth sphi))
                     (- (* spsi sth cphi) (* cpsi sphi)) y)
              (list  (- sth) (* cth sphi) (* cth cphi) z)
              (list 0. 0. 0. 1.))))

(defun inverse-H (H)          ;H is a 4x4 homogeneous transformation matrix.
   (let* ( (minus-P (list  (- (fourth (first H)))
                           (- (fourth (second H)))
                           (- (fourth (third H)))))
           (inverse-R (transpose (first-square (reverse (rest (reverse H))))))
           (inverse-P (post-multiply inverse-R minus-P)))
       (append  (concat-matrix inverse-R (transpose (list inverse-P)))
                (list (list 0 0 0 1)))))

(defun rotation-matrix (azimuth elevation roll)
   (let (  (spsi (sin azimuth)) (cpsi (cos azimuth)) (sth (sin elevation))
           (cth (cos elevation)) (sphi (sin roll)) (cphi (cos roll)))
       (list  (list  (* cpsi cth) (- (* cpsi sth sphi) (* spsi cphi))
                     (+ (* cpsi sth cphi) (* spsi sphi)))
              (list  (* spsi cth) (+ (* cpsi cphi) (* spsi sth sphi))
                     (- (* spsi sth cphi) (* cpsi sphi)))
              (list  (- sth) (* cth sphi) (* cth cphi)))))

(defun body-rate-to-euler-rate-matrix (azimuth elevation roll)
   (let (  (sth (sin elevation)) (cth (cos elevation)) (tth (tan elevation))
           (sphi (sin roll)) (cphi (cos roll)))
       (list  (list 1 (* tth sphi) (* tth cphi))
              (list 0 cphi (- sphi))
              (list 0 (/ sphi cth) (/ cphi cth)))))

(defun rad-to-deg (angle) (* 57.2957795130823 angle))
(defun deg-to-rad (angle) (* .017453292519943295 angle))
```

APPENDIX C: SOURCE CODE (C++)

A. TOETYPES.H

```
/*********************************************************************

    FILE:       TOETYPES.H

    AUTHOR:     Eric Bachmann, Dave Gay
                modified by R. Steven

    DATE:       11 July 1995, modified 15 December 1995

*********************************************************************/

#ifndef __TOETYPES_H
#define __TOETYPES_H

#include <stdio.h>

#define ONE_G 32.2185                   // One g in feet per second
#define GRAVITY 32.2185                 // In feet per second

#define K1 1.0
#define K2 1.0
#define K3 0.5
#define K4 0.7

#define BIAS_1 0.0
#define BIAS_2 0.0
#define BIAS_3 0.0

enum Boolean   {FALSE, TRUE};

typedef char  ONEBYTE;
typedef short TWOBYTE;
typedef long  FOURBYTE;

typedef unsigned char  UNSIGNED_ONEBYTE;
typedef unsigned short UNSIGNED_TWOBYTE;
typedef unsigned long  UNSIGNED_FOURBYTE;
```

```
// Holds a latitude and longitude expressed as doubles
struct latLongPosition {
   double  latitude;
   double  longitude;
};

// Holds a grid position
struct grid {
   double  north, east, depth;
};

// 3 X 3 matrix
struct matrix {
   float element[3][3];
};

// 3 X 1 matrix or vector
struct vector {
   float element[3];
};

// Structure for passing around various types of INS information.
// The positions in the sample field of a stampedSample structure
        // sample[0]: x acceleration
        // sample[1]: y acceleration
        // sample[2]: z acceleration
        // sample[3]: phi
        // sample[4]: theta
        // sample[5]: psi
        // sample[6]: water speed
        // sample[7]: heading
        // sample[8]: true north position
        // sample[9]: true east position
        // sample[10]: true depth                sample [10] - [13] inserted
        // sample[11]: true posture[4]           only to accomodate a changed
        // sample[12]: true posture[5]           mission-data format
        // sample[13]: true posture[6]
```

```
struct stampedSample {
   grid     est;                    //position as estimated by the INS.
   double   sample[14];             //sampler converted sample.
   double   time;
   int      GPSfix;                 // 1: GPSfix, 0: no GPSfix
   float    deltaT;
};

#endif
```

B. POSTFISH.CPP

```
/*********************************************************************

     FILE:      POSTFISH.CPP

     AUTHOR:    Eric Bachmann, Dave Gay
                modified by R. Steven

     DATE:      11 July 1995, modified 15 December 1995

*********************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>

#include "toetypes.h"
#include "postnav.h"

void printPosition (const latLongPosition&);

int
main (int argc, char *argv[])
{
// char goOn = 'y';
   latLongPosition  currentLocation;          // Lat/Long of most recent fix
   int              fixCount(0);
   char             *inputFile;
   Boolean          runInComplete(TRUE);
```

```
    if (argc == 2) {
        inputFile = new char[strlen(argv[1])];
        strcpy(inputFile, argv[1]);                    //explicit script file only

        //Instantiate the navigator
        navigator nav1(inputFile);

        cout << "\nReading data from " << inputFile;

        //Initialize the navigator
        currentLocation = nav1.initializeNavigator();

        cout << "Initial Position:\n";
        //Print the initial position
        printPosition(currentLocation);

        runInComplete = nav1.navPosit(currentLocation);

        while (runInComplete) {
            ++fixCount;

            // Attempt to get a fix from the navigator
            runInComplete = nav1.navPosit(currentLocation);

        }

    }
    else {
        cout << "\nEnter the data file.\n";
    }

        cout << "\nTotal fixes: " << fixCount << endl;
        return 0;
}
```

```
/*********************************************************************

     PROGRAM:     printPositon

     AUTHOR:      Eric Bachmann, Dave Gay
                  modified by R. Steven

     DATE:        11 July 1995, modified 15 December 1995

     FUNCTION:    Displays position to the screen

     RETURNS:     void

     CALLED BY:   mail

     CALLS:       none

*********************************************************************/

void
printPosition (const latLongPosition& posit)
{
   cout << "Latitude: " << posit.latitude << endl;
   cout << "Longitude: " << posit.longitude << endl;
}
```

C. POSTNAV.H

```
/*********************************************************************

     FILE:        POSTNAV.H

     AUTHOR:      Eric Bachmann, Dave Gay
                  modified by R. Steven

     DATE:        11 July 1995, modified 15 December 1995

*********************************************************************/

#ifndef __NAVIGATOR_H
#define __NAVIGATOR_H

#include <stdio.h>
```

```
#include <fstream.h>
#include <iostream.h>
#include <math.h>

#include "toetypes.h"
#include "postins.h"

/*********************************************************************
     CLASS:      navigator

     AUTHOR:     Eric Bachmann, Dave Gay
                 modified by R. Steven

     DATE:       11 July 1995, modified 15 Dec 1995

     FUNCTION:   Combines GPS and INS information to return the current
                 estimated position.

*********************************************************************/

class navigator {
  public:

        //Constructor, opens script and data files
        navigator(char* dataFile)
        :  positionData ("postScrp"),
           inputFile(dataFile),
           elapsedTime (0.0) {}

        //Destructor, closes script and data files
        ~navigator() {positionData.close();}

        //provides the navigator's best estimate of current position
        Boolean navPosit (latLongPosition&);

        //Initialize the navigator
        latLongPosition initializeNavigator();

  private:

        INS ins1;                    //INS object instance.

        ofstream positionData;       // Position script file.
        ifstream inputFile;          //Post processing read file.
```

```
        latLongPosition origin;          // stores origin, normally (0, 0)

        //Write position information to script file
        void writeScriptPosit(int, const grid&, char);

        float elapsedTime;               // Tracks elapsed time for output to script file
};

#endif
```

D. POSTNAV.CPP

```
/*********************************************************************

     FILE:       POSTNAV.CPP

     AUTHOR:     Eric Bachmann, Dave Gay
                 modified by R. Steven

     DATE:       11 July 1995, modified 15 December 1995

*********************************************************************/

#include "postnav.h"

/*********************************************************************

     PROGRAM:    navPosit

     AUTHOR:     Eric Bachmann, Dave Gay
                 modified by R. Steven

     DATE:       11 July 1995, modified 15 Dec 1995

     FUNCTION:   Provides the navigator's best estimate of current position.
     Attempts to obtain GPS and INS position fixes from the gps
     and ins objects and copies the most accurate fix available
     into the input argument 'navPosition'. Writes the raw position
     fix data to the output file for post processing. Sets a return
     flag to indicate whether a valid fix was obtained.

     RETURNS:    TRUE, a valid position fix is in the variable 'navPosition'.
                 FALSE, otherwise.
```

```
    CALLED BY:  towfish.cpp (main)


    CALLS:    correctPosition (ins.h)
              insPosition (ins.h)
              writeScriptPosit (nav.h)

*************************************************************************/

Boolean
navigator::navPosit (latLongPosition& navPosition)
{
  stampedSample  newSample;
  grid           position;                    // to pass position to ins::correctposition

  static  int    fixCount(0);
  Boolean        gpsFlag, insFlag;

  if (inputFile.eof()) return FALSE;
  else
  {
    inputFile >> newSample.time          >> newSample.sample[0]
              >> newSample.sample[1]     >> newSample.sample[2]
              >> newSample.sample[3]     >> newSample.sample[4]
              >> newSample.sample[5]     >> newSample.sample[7]
              >> newSample.sample[6]     >> newSample.sample[8]
              >> newSample.sample[9]     >> newSample.sample[10]
              >> newSample.sample[11]    >> newSample.sample[12]
              >> newSample.sample[13]    >> newSample.GPSfix;

    if (newSample.GPSfix == 1)
    {
      // prepare position to pass to ins::correctposition
      // and writeScriptPosit
      position.north = newSample.sample[8];
      position.east  = newSample.sample[9];
      position.depth = 0.0;
      // Update time for output to file
      elapsedTime = newSample.time;
      // Write new position to file
      writeScriptPosit(++fixCount, position, 'G');
      // Pass GPS position to INS object for navigation corrections.
      ins1.correctPosition(position, newSample.time);
      // Update navPosition
```

```
            navPosition.latitude  = position.north;
            navPosition.longitude = position.east;
            return TRUE;

        }
        else                    // GPSfix == 0
        {
            ins1.insPosition(newSample);
            // Update time for output to file
            elapsedTime = newSample.time;
            //Write new position to script file
            position.north = newSample.sample[0];
            position.east  = newSample.sample[1];
            position.depth = 0.0;
            writeScriptPosit(++fixCount, position, 'I');
            navPosition.latitude  = position.north;
            navPosition.longitude = position.east;
            return TRUE;
        }
    }
}
```

```
/*********************************************************************
     PROGRAM:     writeScriptPosit

     AUTHOR:      Eric Bachmann, Dave Gay

     DATE:        11 July 1995

     FUNCTION:    Writes the fix number, the position in milSec and the type
                  of fix to the script file.

     RETURNS:     void

     CALLED BY:   navPosit  (nav.cpp)
                  initialPosit  (nav.cpp)

     CALLS:       None

*********************************************************************/

void
navigator::writeScriptPosit(int fixNumber, const grid& posit, char fixType)
{
   positionData  << fixNumber    << ' '
                 << posit.north  << ' '
                 << posit.east   << ' '
                 << fixType      << ' '
                 << elapsedTime  << endl;
}

/*********************************************************************
     PROGRAM:     initializeNavigator

     AUTHOR:      Eric Bachmann, Dave Gay
                  modified by R. Steven

     DATE:        11 July 1995, modified 15 Dec 1995

     FUNCTION:    Obtains an initial GPS fix for use as a navigational origin for
                  grid positions used by the INS object. Saves the origin and passes
                  it to the INS object in latLong form.

     RETURNS:     TRUE

     CALLED BY:   towfish (main)
```

```
    CALLS:          correctPosition (ins.cpp)

*********************************************************************/

latLongPosition
navigator::initializeNavigator()
{
   int              originFixNum = 0;
   grid             origin;
   latLongPosition  tmpPosition;

   // Set the origin to 0, 0
   origin.north  =  0.0;
   origin.east   =  0.0;

   writeScriptPosit (originFixNum, origin, 'G');

   tmpPosition.latitude    =  origin.north;
   tmpPosition.longitude   =  origin.east;

   ins1.insSetUp(inputFile);
   //Return the initial position to the caller.
   return tmpPosition;
}
```

E. POSTINS.H

```
/*********************************************************************

    FILE:       POSTINS.H

    AUTHOR:     Eric Bachmann, Dave Gay
                modified by R. Steven

    DATE:       11 July 1995, modified 15 December 1995

*********************************************************************/

#ifndef _INS_H
#define _INS_H

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

```
#include <iostream.h>

#include "toetypes.h"

/*************************************************************************
     CLASS:        ins

     AUTHOR:       Eric Bachmann, Dave Gay

     DATE:         11 July 1995

     FUNCTION:     Takes in linear accelerations, angular rates, speed and
                   heading information and uses kalman filtering techniques to return
                   a dead reconing position.

*************************************************************************/

class INS {

   public:
      //Constructor initializes gains
      INS();

      ~INS() {}

      //returns the ins estimated position
      Boolean insPosition (stampedSample&);

      //Updates the x, y and z of the vehicle posture
      void correctPosition (const grid&, double);

      //records the initial position of and time
      void insSetUp (ifstream&);

   private:
      double posture[6];              // ins estimated posture (x y z phi theta psi)
      double velocities[6];           // ins estimated linear and angular velocities

                                      // x-dot y-dot z-dot phi-dot theta-dot psi-dot

      double current[3];              // ins estimated error current (x-dot y-dot z-dot)

      double lastTime;                //time of last ins position fix
      double lastGPStime;             //time of last gps position fix
```

```
    matrix rotationMatrix;              //body to euler transformation matrix

    double biasCorrection[8];           //Software bias corrections for IMU rate sensors

    // Kalman filter gains.
    float Kone1, Kone2, Ktwo, Kthree1, Kthree2, Kfour1, Kfour2;

    // Transforms from body coordinates to earth coordinates
    // and removes the gravity component
    void transformAccels (double[]);

    // Transforms water speed reading to x and y components
    void transformWaterSpeed (double, double[]);

    // Tranforms body euler rates to earth euler rates.
    void transformBodyRates (double[]);

    // Euler integrates the accelerations and updates the velocities
    void updateVelocities (stampedSample&);

    // Euler integrates the velocities and update the posture
    void updatePosture (stampedSample&);

    // Builds the body to euler rate matrix
    matrix buildBodyRateMatrix ();

    // Builds the body to earth rotation matrix
    void buildRotationMatrix ();

    // Convert magnetic direction based magnetic variation.
    double trueHeading (const double);

    //Calculates the imu bias correction during set up
    void calculateBiasCorrections ();

    //Applies bias corrections to a sample
    void applyBiasCorrections(stampedSample&);

};

// Post multiply a matrix times a vector and return result.
vector operator* (matrix&, double[]);

#endif
```

F. POSTINS.CPP

```
/*********************************************************************

     FILE:        POSTINS.CPP

     AUTHOR:      Eric Bachmann, Dave Gay
                  modified by R. Steven

     DATE:        11 July 1995, modified 15 December 1995

*********************************************************************/

#include <iostream.h>

#include "postins.h"

/*********************************************************************

     PROGRAM:     ins (constructor)

     AUTHOR:      Eric Bachmann, Dave Gay

     DATE:        11 July 1995

     FUNCTION:    Constructor initializes kalman filter gains and linear and
                  angular velocities.

     RETURNS:     nothing

     CALLED BY:   navigator class

     CALLS:       none

*********************************************************************/

INS::INS()  :  Kone1(K1), Kone2(K1), Ktwo(K2),
               Kthree1(K3), Kthree2(K3), Kfour1(K4), Kfour2(K4)
{
   velocities[0] = 0.0;          // x dot
   velocities[1] = 0.0;          // y dot
   velocities[2] = 0.0;          // z dot
   velocities[3] = 0.0;          // phi dot
   velocities[4] = 0.0;          // theta dot
```

```
   velocities[5] = 0.0;            // psi dot

   // Initialize error bias to zero
   current[0] = 0.0;
   current[1] = 0.0;
   current[2] = 0.0;
}

/*********************************************************************

     PROGRAM:      insPosit

     AUTHOR:       Eric Bachmann, Dave Gay
                   modified by R. Steven

     DATE:         11 July 1995, modified 15 Dec 1995

     FUNCTION:     Make dead reckoning position estimation using kalman
                   filtering. Inputs are linear accelerations, angular rates, speed and
                   heading. Primary input data is obtained from a sampler object via the
                   getSample method. This data is stored in the sample filed of a
                   stampedSample structure called newSample. The sample field is then
                   used as a working variable as the linear accelerations and angular
                   rates it contains are converted to earth coordinates and integrated
                   to determine current velocities and posture. The data is complimentary
                   filtered against itself, speed and magnetic heading.

     RETURNS:      position in grid coordinates as estimated by the INS

     CALLED BY:    navPosit (nav.cpp)

     CALLS:        findDeltaT (ins.cpp)
                   transformBodyRates (ins.cpp)
                   buildRotationMatrix (ins.cpp)
                   transformAccels (ins)
                   transformWaterSpeed (ins)

*********************************************************************/

Boolean
INS::insPosition(stampedSample& newSample)
{
   double  thetaA, phiA, xIncline, yIncline;
   double  deltaT;                                  // Integration interval
```

```
double   waterSpeedCorrection[3];                    // Filter correction for drift
                                                     // and water speed

static float elapsedTime = 0.0;                      // Maintains elapsed time

applyBiasCorrections (newSample);

deltaT = newSample.time - lastTime;

newSample.deltaT = deltaT;

xIncline = newSample.sample[0] / GRAVITY;
yIncline = newSample.sample[1] / (GRAVITY * cos(posture[4]));

if (fabs(yIncline) > 1.0) {
   cerr << "\n Inclination error! \n";
   return FALSE;
}

//Calculate low freq pitch and roll
thetaA = asin(xIncline);
phiA = -asin(yIncline);

//Transform body rates to transform euler rates.
transformBodyRates(newSample.sample);

//Calculate estimated pitch rate (phi-dot).
velocities[3] = newSample.sample[3] + Kone1 * (phiA - posture[3]);
//Calculate estimated roll rate (theta-dot).
velocities[4] = newSample.sample[4] + Kone2 * (thetaA - posture[4]);
//Calculate estimated heading rate (psi-dot).
velocities[5] = newSample.sample[5] + Ktwo * (newSample.sample[7] - posture[5]);

//integrate estimated pitch rate to obtain pitch angle
posture[3] += deltaT * velocities[3];
//integrate estimated roll rate to obtain roll angle
posture[4] += deltaT * velocities[4];
//integrate estimated yaw rate to obtain heading
posture[5] += deltaT * velocities[5];

elapsedTime += deltaT;

buildRotationMatrix ();
```

```
    //Transform accels to earth coordinates
    transformAccels (newSample.sample);


    //Transform water speed to earth coordinates
    transformWaterSpeed (newSample.sample[6], waterSpeedCorrection);

    // Subtract out previous velocity and apply statistical gain
    // This is the NEW version of the filter
    waterSpeedCorrection[0] = Kthree1 * (waterSpeedCorrection[0] - velocities[0]);
    waterSpeedCorrection[1] = Kthree2 * (waterSpeedCorrection[1] - velocities[1]);

    // Determine filtered accelerations
    newSample.sample[0] += waterSpeedCorrection[0];
    newSample.sample[1] += waterSpeedCorrection[1];

    //Integrate accelerations to obtain velocities
    velocities[0] += newSample.sample[0] * deltaT;
    velocities[1] += newSample.sample[1] * deltaT;
    velocities[2] += newSample.sample[2] * deltaT;

    //Integrate velocities to obtain posture
    // This is the NEW version of the filter
    posture[0] += (velocities[0] + current[0]) * deltaT;
    posture[1] += (velocities[1] + current[1]) * deltaT;
    posture[2] += (velocities[2] + current[2]) * deltaT;

    lastTime = newSample.time;

    newSample.sample[0] = posture[0];
    newSample.sample[1] = posture[1];
    newSample.sample[2] = posture[2];
    newSample.sample[3] = posture[3];
    newSample.sample[4] = posture[4];
    newSample.sample[5] = posture[5];

    newSample.est.north = posture[0];
    newSample.est.east  = posture[1];
    newSample.est.depth = posture[2];

    return TRUE;
}
```

```
/*********************************************************************

     PROGRAM:   correctPosition

     AUTHOR:    Eric Bachmann, Dave Gay

     DATE:      11 July 1995

     FUNCTION:  Reinitializes the INS based on a known position and compute
                apparent current based on past accumulated errors of the INS. It is
                called by the navigator each time a new GPS (true) fix is obtained.

     RETURNS:   void

     CALLED BY: navPosit (nav)

     CALLS:     none

*********************************************************************/
void
INS::correctPosition(const grid& truePosit, double positTime)
{
   double deltaT;

   // Find time since last gps fix.
   deltaT = positTime - lastGPStime;

   // Detemine INS error since last gps fix
   double deltaX = truePosit.north  -  posture[0];
   double deltaY = truePosit.east   -  posture[1];

   // Reinitialize posture to known position (gps fix)
   posture[0] = truePosit.north;
   posture[1] = truePosit.east;
   posture[2] = 0.0;

   // Add gain filtered error to previous errors
   current[0] += Kfour1 * (deltaX / deltaT);
   current[1] += Kfour2 * (deltaY / deltaT);

   // Save the time of the gps fix for next calculation
   lastGPStime = positTime;
}
```

```
/*********************************************************************

    PROGRAM:     insSetUp

    AUTHOR:      Eric Bachmann, Dave Gay

    DATE:        11 July 1995

    FUNCTION:    Initializes the INS system. Sets the posture to the origin.
                 Initializes the heading using magnetic compass information. Initilizes
                 the times of the last GPS fix and last IMU information.

    RETURNS:     void

    CALLED BY:   initializeNavigator (nav)

    CALLS:       calulateBiasCorrections (ins)
                 buildRotationMatrix (ins)
                 transformWaterSpeed (ins)

*********************************************************************/

void
INS::insSetUp (ifstream& inputFile)
{
   stampedSample newSample;

   //Set posture to straight and level at the origin.
   posture[0] = 0.0;
   posture[1] = 0.0;
   posture[2] = 0.0;
   posture[3] = 0.0;
   posture[4] = 0.0;

   //set imu biases
   calculateBiasCorrections();
   cout << "\nBiases: "      << biasCorrection[3]  << ' '
        << biasCorrection[4] << ' '
        << biasCorrection[5] << endl;

   inputFile >> newSample.time          >> newSample.sample[0]
             >> newSample.sample[1]  >> newSample.sample[2]
             >> newSample.sample[3]  >> newSample.sample[4]
             >> newSample.sample[5]  >> newSample.sample[7]
```

```
            >> newSample.sample[6]  >> newSample.sample[8]
            >> newSample.sample[9]  >> newSample.sample[10]
            >> newSample.sample[11] >> newSample.sample[12]
            >> newSample.sample[13] >> newSample.GPSfix;

   //set initial true heading
   posture[5] = newSample.sample[7];

   //set initial speed
   buildRotationMatrix ();
   transformWaterSpeed (newSample.sample[6], velocities);

   // initialize times
   lastTime = newSample.time;
   lastGPStime = 0.0;
}

/*************************************************************************

      PROGRAM:    transformAccels

      AUTHOR:     Eric Bachmann, Dave Gay

      DATE:       11 July 1995

      FUNCTION:   Transforms linear accelerations from body coordinates to
                  earth coordinates and removes the gravity component in the z direction.

      RETURNS:    void

      CALLED BY:  navPosit

      CALLS:      none

*************************************************************************/

void
INS::transformAccels (double newSample[])
{
   vector earthAccels;

   newSample[0]  -=  GRAVITY * sin(posture[4]);
   newSample[1]  +=  GRAVITY * sin(posture[3]) * cos(posture[4]);
   newSample[2]  +=  GRAVITY * cos(posture[3]) * cos(posture[4]);
```

```
   earthAccels = rotationMatrix * newSample;

   newSample[0] = earthAccels.element[0];
   newSample[1] = earthAccels.element[1];
   newSample[2] = earthAccels.element[2];
}

/*********************************************************************

     PROGRAM:      transformWaterSpeed

     AUTHOR:       Eric Bachmann, Dave Gay

     DATE:         11 July 1995

     FUNCTION:     Transforms water speed into a vector in earth coordinates
                   and returns them in the speedCorrection variable.

     RETURNS:      void

     CALLED BY:    navPosit

     CALLS:        none

*********************************************************************/

void
INS::transformWaterSpeed (double waterSpeed, double speedCorrection[])
{
   double water[3] = {waterSpeed, 0.0, 0.0};
   vector waterVelocities = rotationMatrix * water;

   speedCorrection [0] = waterVelocities.element[0];
   speedCorrection [1] = waterVelocities.element[1];
   speedCorrection [2] = waterVelocities.element[2];
}
```

```
/*********************************************************************

     PROGRAM:   transformBodyRates

     AUTHOR:    Eric Bachmann, Dave Gay
                modified by R. Steven

     DATE:      11 July 1995, modified 15 Dec 1995

     FUNCTION:  Tranforms body euler rates to earth euler rates

     RETURNS:    none

     CALLED BY:  insPosit

     CALLS:      buildBodyRateMatrix

*********************************************************************/

void
INS::transformBodyRates (double newSample[])
{
   vector earthRates = buildBodyRateMatrix() * &(newSample[3]);

   newSample[3] = earthRates.element[0];
   newSample[4] = earthRates.element[1];
   newSample[5] = earthRates.element[2];
}
```

```
/*********************************************************************

     PROGRAM:      buildBodyRateMatrix

     AUTHOR:       Eric Bachmann, Dave Gay

     DATE:         11 July 1995

     FUNCTION:     Builds body to Euler rate translation matrix.

     RETURNS:      rate translation matrix

     CALLED BY:    insPosit

     CALLS:        none

*********************************************************************/

matrix
INS::buildBodyRateMatrix ()
{
   matrix rateTrans;

   float tth    = tan (posture[4]),
         sphi   = sin(posture[3]),
         cphi   = cos(posture[3]),
         cth    = cos(posture[4]);

   rateTrans.element[0][0] = 1.0;
   rateTrans.element[0][1] = tth * sphi;
   rateTrans.element[0][2] = tth * cphi;
   rateTrans.element[1][0] = 0.0;
   rateTrans.element[1][1] = cphi;
   rateTrans.element[1][2] = -sphi;
   rateTrans.element[2][0] = 0.0;
   rateTrans.element[2][1] = sphi / cth;
   rateTrans.element[2][2] = cphi / cth;

   return rateTrans;
}
```

```
/*********************************************************************

     PROGRAM:   buildRotationMatrix

     AUTHOR:    Eric Bachmann, Dave Gay

     DATE:      11 July 1995

     FUNCTION:  Sets the body to earth coordinate rotation matrix.

     RETURNS:   void

     CALLED BY: insPosit
                insSetUp

     CALLS:     none

*********************************************************************/

void
INS::buildRotationMatrix ()
{
   float spsi  = sin(posture[5]),
         cpsi  = cos(posture[5]),
         sth   = sin(posture[4]),
         sphi  = sin(posture[3]),
         cphi  = cos(posture[3]),
         cth   = cos(posture[4]);

   rotationMatrix.element[0][0] = cpsi * cth;
   rotationMatrix.element[0][1] = (cpsi * sth * sphi) - (spsi * cphi);
   rotationMatrix.element[0][2] = (cpsi * sth * cphi) + (spsi * sphi);
   rotationMatrix.element[1][0] = spsi * cth;
   rotationMatrix.element[1][1] = (cpsi * cphi) + (spsi * sth * sphi);
   rotationMatrix.element[1][2] = (spsi * sth * cphi) - (cpsi * sphi);
   rotationMatrix.element[2][0] = -sth;
   rotationMatrix.element[2][1] = cth * sphi;
   rotationMatrix.element[2][2] = cth * cphi;
}
```

```
/*********************************************************************

     PROGRAM:     post multiplication operator *

     AUTHOR:      Eric Bachmann, Dave Gay

     DATE:        11 July 1995

     FUNCTION:    Post multiply a 3 X 3 matrix times a 3 X 1 vector and
                  return the result.

     RETURNS:     3 X 1 vector

     CALLED BY:

     CALLS:       None

*********************************************************************/

vector
operator* (matrix& transform, double state[])
{
   vector result;

   for (int i = 0; i < 3; i++) {
      result.element[i] = 0.0;
      for (int j = 0; j < 3; j++) {
         result.element[i] += transform.element[i][j] * state[j];
      }
   }

   return result;
}
```

```
/*********************************************************************

     PROGRAM:     calculateBiasCorrections

     AUTHOR:      Eric Bachmann, Dave Gay

     DATE:        11 July 1995

     FUNCTION:    Calculates the initial imu bias by averaging a number of
                  imu readings.

     RETURNS:     none

     CALLED BY:   insSetup

     CALLS:       none

*********************************************************************/

void
INS::calculateBiasCorrections ()
{
   biasCorrection[3] = BIAS_1;
   biasCorrection[4] = BIAS_2;
   biasCorrection[5] = BIAS_3;
}
```

```
/*********************************************************************

     PROGRAM:     applyBiasCorrections

     AUTHOR:      Eric Bachmann, Dave Gay

     DATE:        11 July 1995

     FUNCTION:    Applies updated bias corrections to a sample.

     RETURNS:     void

     CALLED BY:   insPosit

     CALLS:       none

*********************************************************************/

void
INS::applyBiasCorrections (stampedSample& sample)
{
   static const float biasWght(0.999), sampleWght(0.001);

   biasCorrection[3] = (biasWght * BIAS_1) - (sampleWght * sample.sample[3]);
   biasCorrection[4] = (biasWght * BIAS_2) - (sampleWght * sample.sample[4]);
   biasCorrection[5] = (biasWght * BIAS_3) - (sampleWght * sample.sample[5]);

   sample.sample[3] += biasCorrection[3];
   sample.sample[4] += biasCorrection[4];
   sample.sample[5] += biasCorrection[5];
}
```

LIST OF REFERENCES

Atwood, D.K., Leonard, J.J., Bellingham, J.G., Moran, B.A., An Acoustic Navigation System for Multi-Vehicle Operations, in: 9th International Symposium on Unmanned Untethered Submersible Technology, 1995, pp. 202 - 208

Austin, T.C., The Application of Spread Spectrum Signaling Techniques to Underwater Acoustic Navigation, in: Proceedings of the 1994 Symposium on Autonomous Underwater Vehicle Technology, pp. 443 - 449

Bachmann, E., Gay, D., Design and Evaluation of an Integrated GPS/INS System for Shallow Water Navigation, Master's Thesis, Naval Postgraduate School, Monterey, California, 1995

Balasuriya, B.A.A.P., Fuji, T., Ura, T., Underwater Pattern Observation for Positioning and Communication of AUVs, in: 9th International Symposium on Unmanned Untethered Submersible Technology, 1995, pp. 193 - 201

Bellingham, J.G., Consi, T.R., Tedrow, U., Di Massa, D., Hyperbolic Acoustic Navigation for Underwater Vehicles: Implementation and Demonstration, in: 1992 IEEE Symposium on Autonomous Underwater Vehicle Technology, pp. 304 - 309

Byrnes, R., The Rational Behavior Model: A Multi-Paradigm, Tri-Level Software Architecture for the Control of Autonomous Vehicles, PhD Dissertation, Naval Postgraduate School, Monterey, California, 1993

Carof, A.H., Acoustic Differential Delay & Doppler Tracking System for Long Range AUV Positioning and Guidance, in: Proceedings of the 1994 Symposium on Autonomous Underwater Vehicle Technology, pp. 370 - 375

Consi, T.R., Atema, J., Goudey, C.A., Cho, J., Chryssostomidis, C., AUV Guidance with Chemical Signals, in: Proceedings of the 1994 Symposium on Autonomous Underwater Vehicle Technology, pp. 450 - 453

Cox, R., Wei, S., Advances in the State of the Art for AUV Inertial Sensors and Navigation Systems, in: IEEE Journal of Oceanic Engineering, October 1995, Number 4, pp. 361 - 366

Craig, J.J., Introduction to Robotics, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989

Healey, A.J., Marco, D.B., McGhee, R.B., Brutzman, D.P., Cristi, R., Papoulias, F.A., Kwak, S.H., Tactical/Execution Level Coordination for Hover Control of the NPS AUV II using Onboard Sonar Servoing, in: Proceedings of the 1994 Symposium on Autonomous Underwater Vehicle Technology, pp. 129 - 137

McGhee, R.B., Frank, A.A., Some Considerations Relating to the Design of Autopilots for Legged Vehicles, in: Journal of Terramechanics, Vol. 6, Nr. 1, 1969, pp. 23 - 31

McGhee, R.B., Cooke, J.M., Zyda, M.J., Pratt, D.R., NPSNET: Flight Simulation Dynamic Modeling Using Quaternions, in: Presence, Vol. 1, Nr. 4, Fall 1992, 1993, p. 411

McGhee, R.B., Clynch, J.R., Healey, A.J., Kwak, S.H., Brutzman, D.P., Yun, X.P., Norton, N.A., Whalen, R.H., Bachmann, E.R., Gay, D.L., Schubert, W.R., An Experimental Study of an Integrated GPS/INS System for Shallow-Water AUV Navigation, in: 9th International Symposium on Unmanned Untethered Submersible Technology, 1995, pp. 153 - 167

McGhee, R.B., Derivation of SANS filter equations, classnotes, Naval Postgraduate School, Monterey, CA 93943, Feb 3, 1996, p. 2

McGhee, R.B., Course Notes for CS 4920, Naval Postgraduate School, Monterey, CA 93943, Spring, 1996

McPhail, S., Development of a Simple Navigation System for the Autosub Autonomous Underwater Vehicle, in: Proc. Ocean 93, Victoria, pp. 504 - 509

Koschmann, T.D., The Common LISP Companion, John Wiley & Sons, New York 1990

Norton, N.A., Evaluation of Hardware and Software for a Small Autonomous Underwater Vehicle Navigation System (SANS), Master's thesis, Naval Postgraduate School, Monterey, California. 1994

Pohl, I., Object-Oriented Programming Using C++, The Benjamin/Cummings Publishing Company, Redwood City, 1993

Rendas, M.J., Lourtie, I.M.G., Hybrid Navigation System for Long Range Navigation, in: Proceedings of the 1994 Symposium on Autonomous Underwater Vehicle Technology, pp. 353 - 359

Scherbatyuk, A.P., One Approach to Designing of an Intelligent Positioning System for AUV, in: 9th International Symposium on Unmanned Untethered Submersible Technology, 1995, pp. 139 - 142

Smith, R., Stevens, A., Durrent-Whyte, H., A Navigation System for Accurate Localization in the Vicinity of Underwater Structure, in: 9th International Symposium on Unmanned Untethered Submersible Technology, 1995, pp. 209 - 219

Vaganay, J., Rigaud, V., Supervised Navigation: Optimal Algorithm Example and Needs for Autonomous Supervision, in: Proceedings of 1995 IEEE International Conference on Robotics and Automation, pp. 1874 - 1879

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Director, Training and Education 1
   MCCDC, Code C46
   1019 Elliot Rd.
   Quantico, VA 22134-5027

4. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Dr. Robert B. McGhee, Code CS/Mz 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Dr. Don Brutzman, Code UW/Br 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

7. Dr. James Clynch, Code OC/CL 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

8. LCDR Ruediger Steven 2
   Stapelholmer Weg 67
   24988 Oeversee
   Germany

9. Lt. Eric Bachmann 1
   1281 Spruance Rd.
   Monterey, CA 93940

10. LCDR David Gay 1
P.O. Box 2448
Deming, NM 88031

11. Russ Whalen, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000

12. Guy Oliver 1
University of California, Santa Cruz
233 Northrop Place
Santa Cruz, CA 95060

13. Dr. Anthony J. Healey, Code ME/Hy 1
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, CA 93943-5000

14. Mr. Norman Caplan 1
National Science Foundation
BES, Room 565
4201 Wilson Blvd.
Arlington, VA 22230