Report No. K-TRAN: KSU-99-7
FINAL REPORT

REFINEMENT OF MEASUREMENT TECHNIQUES OF ROAD PROFILE AND INTERNATIONAL ROUGHNESS INDEX (IRI) TO SUPPORT THE KDOT PAVEMENT MANAGEMENT SYSTEM (PMS) ANNUAL ROAD-CONDITION SURVEY RESEARCH

Stephen A. Dyer, Ph.D., P.E.
Ryan Boyd
Justin S. Dyer

Kansas State University
Manhattan, Kansas

MARCH 2005

K-TRAN

A COOPERATIVE TRANSPORTATION RESEARCH PROGRAM BETWEEN:
KANSAS DEPARTMENT OF TRANSPORTATION
KANSAS STATE UNIVERSITY
THE UNIVERSITY OF KANSAS

1 Report No. K-TRAN: KSU-99-7	**2 Government Accession No.**	**3 Recipient Catalog No.**
4 Title and Subtitle REFINEMENT OF MEASUREMENT TECHNIQUES OF ROAD PROFILE AND INTERNATIONAL ROUGHNESS INDEX (IRI) TO SUPPORT THE KDOT PAVEMENT MANAGEMENT SYSTEM (PMS) ANNUAL ROAD-CONDITION SURVEY RESEARCH		**5 Report Date** March 2005 **6 Performing Organization Code**
7 Author(s) Stephen A. Dyer, Ryan Boyd and Justin S. Dyer		**8 Performing Organization Report No.**
9 Performing Organization Name and Address Kansas State University; Department of Civil Engineering 2118 Fiedler Hall Manhattan, KS 66506		**10 Work Unit No. (TRAIS)** **11 Contract or Grant No.** C1157
12 Sponsoring Agency Name and Address Kansas Department of Transportation Bureau of Materials and Research 700 SW Harrison Street Topeka, Kansas 66603-3754		**13 Type of Report and Period Covered** Final Report May 1999 – March 2001 **14 Sponsoring Agency Code** RE-0182-01

15 Supplementary Notes
For more information write to address in block 9.

16 Abstract

K-TRAN Project KSU-99-7 is the first phase of a proposed larger research effort whose goal is twofold:

1). To determine effective guidelines for collecting and processing road profiles.

2). To determine, insofar as possible, the specific causes of the poor repeatability in the data obtained by the present KDOT Pavement Management System

It was undertaken subsequent to the realization that the standards for instrumentation used to measure and reduce road profiles are not sufficiently defined to produce a highly repeatable measure of road roughness such as the International Roughness Index (IRI).

The principal outcomes of the work on this project include:

1.) An initial version of analysis-and-simulation software, written principally in C++, developed to simulate inertial profilers and perform various calculations on sample profile data.

2.) A set of recommendations regarding (1) additional information needed from the manufacturer of the profilers that KDOT uses, and (2) the profile-acquisition procedure employed by KDOT.

Appendices A and B (source codes) available on CD by request to KDOT Library; 700 SW Harrison Street, Topeka, Kansas 66603-3754; Phone: 785-291-3854; Fax: 785-291-3717; e-mail: library@ksdot.org.

17 Key Words International Roughness Index, IRI, Pavement Management System, PMS, Road Profile and Roughness	**18 Distribution Statement** No restrictions. This document is available to the public through the National Technical Information Service, Springfield, Virginia 22161

19 Security Classification (of this report) Unclassified	**20 Security Classification (of this page)** Unclassified	**21 No. of pages** 41 printed (149 with appendices – available only on CD)	**22 Price**

Form DOT F 1700.7 (8-72)

REFINEMENT OF MEASUREMENT TECHNIQUES OF ROAD PROFILE AND INTERNATIONAL ROUGHNESS INDEX (IRI) TO SUPPORT THE KDOT PAVEMENT MANAGEMENT SYSTEM (PMS) ANNUAL ROAD-CONDITION SURVEY RESEARCH

Final Report

Prepared by

Stephen A. Dyer, Ph.D., P.E.
Ryan Boyd
Justin S. Dyer
Kansas State University

A Report on Research Sponsored By

THE KANSAS DEPARTMENT OF TRANSPORTATION
TOPEKA, KANSAS

KANSAS STATE UNIVERSITY
MANHATTAN, KANSAS

March 2005

PREFACE

The Kansas Department of Transportation’s (KDOT) Kansas Transportation Research and New-Developments (K-TRAN) Research Program funded this research project. It is an ongoing, cooperative and comprehensive research program addressing transportation needs of the state of Kansas utilizing academic and research resources from KDOT, Kansas State University and the University of Kansas. Transportation professionals in KDOT and the universities jointly develop the projects included in the research program.

NOTICE

DISCLAIMER

Abstract

K-TRAN Project KSU-99-7 is the first phase of a proposed larger research effort whose goal is twofold:

1. To determine effective guidelines for collecting and processing road profiles.
2. To determine, insofar as possible, the specific causes of the poor repeatability in the data obtained by the present KDOT Pavement Management System.

It was undertaken subsequent to the realization that the standards for instrumentation used to measure and reduce road profiles are not sufficiently defined to produce a highly repeatable measure of road roughness such as the International Roughness Index (IRI).

The principal outcomes of the work on this project include:

1. An initial version of analysis-and-simulation software, written principally in C++, developed to simulate inertial profilers and perform various calculations on sample profile data.
2. A set of recommendations regarding both (1) additional information needed from the manufacturer of the profilers that KDOT uses, and (2) the profile-acquisition procedure employed by KDOT.

Contents

List of Figures

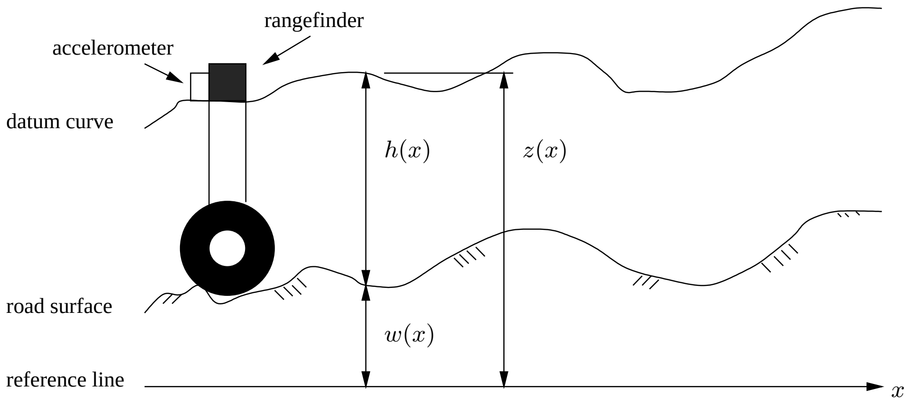


Figure 1: The relationship among quantities of interest in obtaining a road profile.

1 Introduction

1.1 Background

The standards for instrumentation used to measure and reduce road profiles are not sufficiently defined to produce a highly repeatable measure of road roughness such as the International Roughness Index (IRI). According to the Kansas Department of Transportation (KDOT), estimates from oversampled data from its Spring 1997 road-condition survey show the variability of the measure to be on the order of 15% (one standard deviation), whereas the calibration data would indicate an expected variability of less than 2%. The source of the variation is not known. It could be equipment malfunctions, natural variability of the road surface, or some other factor(s).

Instrumentation standards for collecting and reducing road profiles are noticeably absent in the literature. Parameters such as maximum vertical acceleration, minimum sensor resolution and accuracy, and sampling intervals are generally left to the discretion of the equipment vendor. Thus, an expert review of the general data-collection system for high-speed profilers has the potential for identifying and eliminating a large source of measurement variation.

1.2 Basic Principle of Operation of an Inertial Profiler

To determine a figure of merit of a road surface, we must measure, by some means, either directly or indirectly, the variation w of the road surface with respect to some reference curve. Figure 1 illustrates, among other things, an exemplar *road surface* and a horizontal *reference line*. In practice, a properly highpass-filtered version $\widehat{w}_f$ of the estimate $\widehat{w}(x)$ of $w(x)$ is sufficient for input to figure-of-merit algorithms such as the IRI; figures of merit for road roughness are relatively independent of long-wavelength variations in road surface.

Figure 2 shows the basic block diagram of an inertial-frame–based system for measuring road-surface roughness. The KDOT high-speed profilers are of this type.

A sensor—in this case a non-contact, laser rangefinder—is mounted to some point on the high-speed profiler. The rangefinder samples and reports an estimate $\widehat{h}(x)$ of the distance

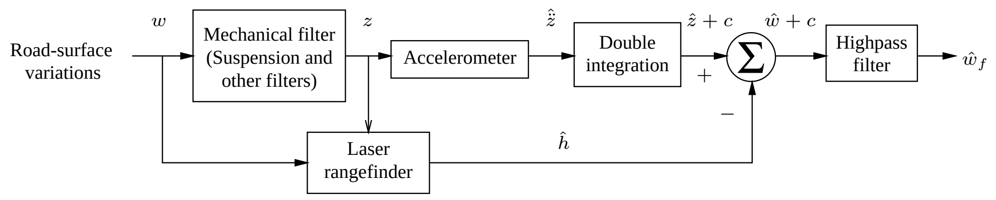


Figure 2: Basic block diagram of the road profiler's operation.

h between its mounted position and the surface of the road. The sensed distance is relative to the rangefinder, which is bouncing with the profiler. It is necessary to remove the effects of the profiler's vertical motion on its tires and suspension (assuming that the rangefinder is mounted to some sprung point on the profiler).

Consider as a datum point the point on the profiler that the rangefinder makes its measurement with respect to. We could trace out a curve painted by that datum point as the profiler moves along its path in the x direction. Denote as $z(x)$ the vertical distance, at any point x, from the reference line to the datum curve.

Now, $z(x) = h(x) + w(x)$. The rangefinder gives us $\widehat{h}(x)$, so we could obtain an estimate $\widehat{w}(x)$ of the needed quantity $w(x)$ if we knew $z(x)$. An accelerometer mounted adjacent to the rangefinder provides an estimate $\widehat{\ddot{z}}$ of the vertical component $\ddot{z}$ of acceleration, which, upon double integration, yields, to within some offset c, an estimate $\widehat{z}(x)$ of $z(x)$.

Finally, we obtain the highpass-filtered estimate of $w(x)$ as

$$\widehat{w}_f(x) = \mathrm{HP}\{\widehat{w}(x) + c\} = \mathrm{HP}\{\widehat{w}(x)\}\,.$$

1.3 Long-term Research Objectives

The work summarized in this report is the first phase of a larger research effort whose goal is twofold:

1. To determine effective guidelines for collecting and processing road profiles.

2. To determine insofar as possible the specific causes of the poor repeatability in the data obtained by the present KDOT Pavement Management System.

Following is a list of tasks that comprise the elements of a systematic approach to determining the expected performance of the KDOT PMS Annual Road-Condition Survey.

1. Review ASTM E 950-94, "Standard Test Method for Measuring the Longitudinal Profile of Traveled Surfaces with an Accelerometer Established Inertial Profiling Reference," for possible inadequacies in the specification of the test method.

2. Perform a sensitivity analysis on the various roughness indices (e.g., IRI) used by KDOT.

3. Perform a literature search to ascertain the maximum range of vertical acceleration to be expected during a profiling run, thus allowing us to specify the dynamic range required of the acceleration sensors.

4. Analyze the effects of the phase shifts associated with the digital filters used during the production of road profiles.

5. Obtain complete specifications of all sensors, of the suspension, of all mechanical filters, and of the mechanical layout of the sensor subsystems associated with KDOT profilers.

6. Analyze the effects of resolution and accuracy of the acceleration samples on the profiles and the computed road-roughness statistics.

7. Analyze the effects of resolution, accuracy, and beam size of the non-contact sensors on the profiles and the computed road-roughness statistics.

8. Analyze the effects of out-of-range errors, associated with the sensors, on the profiles and statistics obtained.

9. Perform a literature search to ascertain the expected maximum and minimum natural variability of pavement.

10. Perform an analytic comparison of time-based sampling vs. spatial (i.e., distance-based) sampling.

11. Develop an overall statistical model of each of the KDOT profilers from an instrumentation-and-measurement perspective.

12. Write an analysis-and-simulation software package to investigate the expected variability in roughness index, given particular conditions established by the user.

There are more tasks listed than can be accomplished within the scope of this project. The objective of the present project was to accomplish the subset of these tasks thought to be most important to our developing an understanding of the principal causes of the poor repeatability in the KDOT PMS data. That subset follows.

1.4 Objectives of this Project

The specific tasks to be accomplished during this project included:

1. Review ASTM E 950-94, "Standard Test Method for Measuring the Longitudinal Profile of Traveled Surfaces with an Accelerometer Established Inertial Profiling Reference," for possible inadequacies in the specification of the test method.

2. Perform a sensitivity analysis on the various roughness indices (e.g., IRI) used by KDOT.

3. Perform a literature search to ascertain the maximum range of vertical acceleration to be expected during a profiling run, thus allowing us to specify the dynamic range required of the acceleration sensors.

4. Analyze the effects of the phase shifts associated with the digital filters used during the production of road profiles.

5. Obtain, insofar as possible, complete specifications of all sensors, of the suspension, of all mechanical filters, and of the mechanical layout of the sensor subsystems associated with KDOT profilers. Participation, at a minimal level, was expected to be required of KDOT personnel to familiarize us with the KDOT profilers and their sensors.

As this project got underway, it became evident that considerable effort should be directed toward the initial design and development of the analysis-and-simulation software listed above as one of the tasks under “Long-term Research Objectives.” That development comprised the majority of time and effort spent on this project, and has resulted in a skeleton program and user-interface, along with the data classes most necessary to further development of the software package.

1.5 Possible Benefits to KDOT

KDOT collects profile data for the entire state highway system on an annual cycle. Any improvements in the accuracy of the data collected would have an immediate impact on the operation of the system. Also, the profilograph industry and the high-speed–profiler industry are beginning to merge as a result of the reduced cost of the sensor-related and computing technology previously relegated to “high-end” instrumentation. Instrumentation guidelines and standards for developing profiling equipment would lead to increased reliability for the industry and thus better quality- and management-controls for KDOT.

As described above, the results of the long-term effort have the potential of improving the quality of the PMS data collection. Reducing the variability of the IRI measurements would enhance the effectiveness with which the PMS recommends projects for substantial maintenance. Potential benefits could range in the millions of dollars annually.

2 Comments on ASTM E 950-94

ASTM E 950-94, “Standard Test Method for Measuring the Longitudinal Profile of Traveled Surfaces with an Accelerometer Established Inertial Profiling Reference,” [5] covers the measurement and recording of road profiles with an inertial-frame profiler. This standard provides general guidelines regarding test methodology, apparatus, calibration procedures, procedures associated with taking measurements, reports, and precision and bias.

We make the following summary remarks—some informational, others regarding unclear or possibly unreasonable statements made within the standard, and yet others regarding lack of guidelines within the standard.

1. That KDOT uses a sampling interval of 3 inches places its profilers in Class 2, as relates to this standard, according to [5, Table 1].

2. The only statement regarding required accuracy of the accelerometer is the following [5, Sec. 5.3.1]: “The accelerometer shall have a minimum resolution to allow profile calculation and accuracy and bias to meet the class requirements as given in this standard.” However, there are no other statements of required accuracy made within the standard.

3. Although there are requirements on vertical-measurement resolution given in the standard [5, Sec. 5.3.2 and Table 2], there is no statement regarding required accuracy of the vertical-displacement measuring device (the laser rangefinder, in KDOT’s case).

4. The requirements in [5, Sec. 5.4.3] placed on filtering (“5.4.3 Filtering that permits the computation of measured elevation profile with no attenuation or amplification of road profile wave lengths at least 60 m (200 ft) long at test speeds of 25 to 95 km/h (15 to 60 mph) shall be provided. The computer and system shall not add noise in excess of 10 % of the displacement measuring transducer resolutions given in Table 2.”) are ambiguous and unreasonable, and are almost certainly not met by any available profiler. Likewise, the requirement on noise is most probably unable to be met by any of the commercially available profilers.

5. The signal-to-noise statement in [5, Sec. 5.7] associated with storage devices (“Signal to noise (S/N) ratio shall be 10 or better.”) is ambiguous. Further, a S/N ratio of 10 is very poor for such an application, making this statement quite out of line with the aforementioned statement on noise made in [5, Sec. 5.4.3].

6. The statements in both [5, Sec. 8.2.1] on accelerometer error (“In either case, an error larger than that allowed for the class shall not be accepted.”) and [5, Sec. 8.2.2] on displacement-transducer error (“The displacement step shall be at least 25 mm (1.0 in.) and accurate within class requirement.”) have no statement of required accuracy elsewhere within ASTM E 950-94 upon which to depend.

7. [5, Sec. 9.4.2] and [5, Sec. 9.4.3] list specific requirements regarding lead-in to a test section of road.

8. [5, Sec. 12] covers precision and bias. [5, Sec. 12.1] states, “12.1 The accuracy of pavement profile measuring equipment can be defined by a statement on the precision and bias of the measuring equipment.” However, statements on precision and bias are *not* equivalent to statements on accuracy. The most evident deficiency—and possibly the one with the greatest consequences—in [5] is the general lack of specific statements on required accuracy (i.e., freedom from measurement error).

3 The International Roughness Index

The International Roughness Index, or IRI, is one way of measuring the roughness of a section of roadway. It is the reference average rectified slope (RARS$_{80}$) of a simulated standard quarter-car traveling at a speed of 80 km/h, which yields a ratio of the accumulated motion of the quarter-car's suspension to the distance traveled during the test. More specifically, the IRI is based on these five points, quoted from [2]:

1. IRI is computed from a single longitudinal profile. The sample interval should be no larger than 300 mm for accurate calculations. The required resolution depends on the roughness level, with finer resolutions being needed for smooth roads. A resolution of 0.5 mm is suitable for all conditions.
2. The profile is assumed to have a constant slope between sampled elevation points.
3. The profile is smoothed with a moving average whose baselength is 250 mm.
4. The smoothed profile is filtered using a quarter-car simulation, with specific parameter values (Golden Car), at a simulated speed of 80 km/h (49.7 mi/h).
5. The simulated suspension motion is linearly accumulated and divided by the length of the profile to yield IRI. Thus, IRI has units of slope, such as in/mi or m/km.

The guidelines published by the World Bank for conducting and calibrating roughness measurements [1] include a discussion of the IRI and its computation.

3.1 Overview of the IRI

We briefly summarize the components of the IRI in the following.

3.1.1 Moving-Average Filter

The moving average converts the measured profile height values h_p into smoothed elevation values h_{ps} according to

$$h_{ps}[i] = \frac{1}{k} \sum_{j=i}^{i+k-1} h_p[j] \; .$$

The filter-length k is given by

$$k = \max(1, \lfloor L_B/\Delta + 0.5 \rfloor)$$

where L_B is the baselength of the moving average, which is 250 mm for IRI, and $\lfloor \; \rfloor$ denotes "the floor of."

The smoothed-profile slope values are

$$s_{ps}[i] = \frac{h_{ps}[i+k] - h_{ps}[i]}{k\Delta} \; .$$

3.1.2 Quarter-Car Filter

The quarter-car filter is a simulation of the suspension of a vehicle, which in turn is used to simulate the response of a response-type road-roughness measuring (RTRRM) system. It is defined by a system of differential equations, represented in vector–matrix form as

$$\dot{\boldsymbol{h}} = \boldsymbol{A}\boldsymbol{h} + \boldsymbol{B}h_{ps}$$

where

$$\boldsymbol{h} = \begin{bmatrix} h_s \\ \dot{h}_s \\ h_u \\ \dot{h}_u \end{bmatrix}, \quad \boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -k_2 & -c & k_2 & c \\ 0 & 0 & 0 & 1 \\ \frac{k_2}{\mu} & \frac{c}{\mu} & -\frac{k_1+k_2}{\mu} & -\frac{c}{\mu} \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{k_1}{\mu} \end{bmatrix}$$

$\boldsymbol{h}$ is an array of state variables that holds the height and velocity of the sprung and unsprung masses of the quarter-car model. The variable h_s represents the height of the sprung mass, and h_u represents the height of the unsprung mass.

The "golden-car" parameters in the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices are defined as follows:

Ratio of suspension damping rate to sprung mass:	c	$=$	6.0
Ratio of tire spring rate to sprung mass:	k_1	$=$	653
Ratio of suspension spring rate to sprung mass:	k_2	$=$	63.3
Ratio of unsprung mass to sprung mass:	μ	$=$	0.15

3.1.3 Solution of the Quarter-Car Equations

The system of differential equations given above has the solution

$$\boldsymbol{h}[i] = e^{\boldsymbol{A}\Delta/V}\boldsymbol{h}[i-1] + \boldsymbol{A}^{-1}(e^{\boldsymbol{A}\Delta/V} - \boldsymbol{I})\boldsymbol{B}h_{ps}[i]$$

where $\boldsymbol{I}$ is a 4×4 identity matrix, and V is the velocity used in the simulation. ($V = 80$ km/h for the IRI, by definition.)

Alternatively, the state equation

$$\dot{\boldsymbol{s}} = \boldsymbol{A}\boldsymbol{s} + \boldsymbol{B}s_{ps}$$

can be solved, where

$$\boldsymbol{s} = \begin{bmatrix} s_s \\ \dot{s}_s \\ s_u \\ \dot{s}_u \end{bmatrix}.$$

This second state equation fulfills the assumption, given in [2], of constant slope between sampled elevation points and has the same form of solution as the state equation involving $\boldsymbol{h}$.

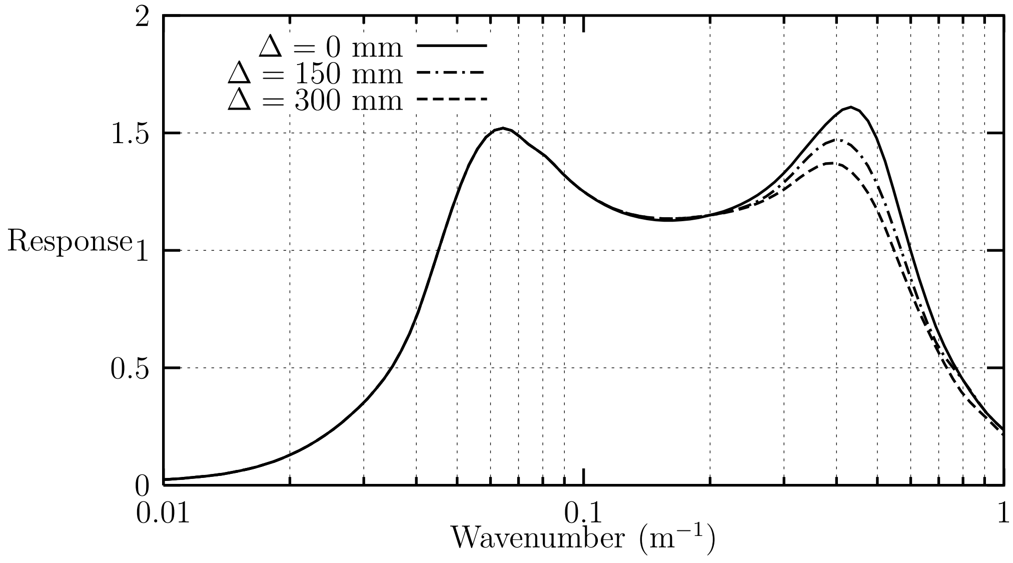


Figure 3: Response of the simulated quarter-car used in the IRI. The recommended MA pre-filter and solution via the state-transition matrix are used. (Adapted from [2].)

3.1.4 IRI Accumulation

After the moving-average and quarter-car filters have been applied, the IRI is accumulated as follows:

$$\text{IRI} = \frac{1}{n-k} \sum_{i=1}^{n-k} |s_s[i] - s_u[i]|$$

where n is the total number of samples collected.

3.2 Discussion of Sensitivity

Since the IRI is based on applying a measured road-profile to a standard model (the "golden-car" quarter-car), the major sensitivities are (1) the sampling interval Δ used when taking the profile measurements, (2) the specifics of the discrete-time simulation of the dynamics of the quarter-car model, and (3) the profile-heights input to the IRI calculation.

The moving-average (MA) filter reduces the IRI's sensitivity to the specific choice of Δ used in obtaining the profile.

Sayers [2] summarizes the response associated with the quarter-car model. Figure 3 indicates that the major sensitivity of the quarter-car response to the sampling interval Δ occurs within the range of about 0.2 to 0.6 m^{-1}, and even then is relatively insignificant for $50 \leq \Delta \leq 150$ mm (i.e., $2 \leq \Delta \leq 6$ in). This figure also indicates that the range of wavelengths contributing significantly to the IRI statistic is roughly 1 to 30 m (about 3 to 90 ft).

It can be further shown [2] that, when the solution of the quarter-car equations is obtained via the state-transition matrix, the computed IRI matches the true IRI with excellent fidelity for a wide range of Δ. The agreement is nearly perfect for $\Delta \leq 100$mm ≈ 4 in. The ratio of computed IRI to true IRI falls off gradually from 1.0 to about 0.84 as Δ is increased to 1 m, and then falls off rapidly for $\Delta > 1$ m.

Studies (e.g., by Ksaibati, Ansani, and Adkins [1993] and Noval and DeFrain [1992], summarized briefly in [3, pp. 174–175]) show a large dependence of IRI on environmental conditions. Ksaibati, *et al.,* in their study, showed the IRI for jointed plain concrete pavement to be linearly related to the amount of rainfall occurring within 72 hours before testing, and quadratically related to the change in temperature during the 24 hours previous to testing; the fit was good ($R^2 = 0.849$). Novak and DeFrain's study showed 47–163% increases from summer to winter in measured IRI for certain composite sections, and 31–46% *decreases* from summer to winter for composite sections in other locations. These reported climate-related variations in IRI seem very significant.

4 Range of Vertical Acceleration

A literature search to ascertain the maximum expected range of vertical acceleration during a profiling run yielded very little information.

There are no guidelines for dynamic range of the accelerometer set forth in the ASTM E 950-94 standard—only the following general statement: "The accelerometer range shall be large enough to accommodate the levels of acceleration expected from the bounce motions of the measuring vehicle (typically ± 1 g). The accelerometer shall be biased to account for the 1-g acceleration of gravity."

The report [4] describing the South Dakota road profiler, from which the KDOT profilers have descended, note the use of a linear servo accelerometer having a ± 2g range. Since an accelerometer in a profiler application is subjected to the earth's gravitational pull, one having a ± 2g range is useful for vertical accelerations of the profiler between $+1$g and -3g. A remark is made in [4, p. 24] regarding the adequacy of the ± 2g range "to measure significant vehicle motions ...", but without reference to any studies. Recent (2004) internal studies by another profiler manufacturer suggest that vertical acceleration can easily exceed ± 2g under normal operating conditions on only moderately-rough (~ 200 in/mi) jointed concrete pavement (JCP) surfaces.

We make recommendations in the Conclusion regarding experimental study to resolve this issue.

5 Effects of Phase Shifts Associated with Digital Filters

There is the possibility of including filters in the data-acquisition and processing chain associated with a road profiler. The time delay encountered by a signal as it passes through a filter has an effect on the accuracy of the profile-estimate recorded and of the roughness index calculated from that profile-estimate.

The time delay is related to the phase shift of the filter, and the phase shift is in turn a nonlinear function of frequency except in the case of certain types of finite-impulse-response (FIR) digital filters. The vast range of possible effects of the phase shift (and, consequently, the time delay) depends on the particular characteristics of the filters used.

We have written a set of functions that facilitate computation of frequency response (magnitude reponse, phase response, and time delay) of both continuous-time and discrete-time (digital) filters, given their transfer functions. Also included are a frequency-scaling function and transformation functions that perform lowpass-to-highpass, lowpass-to-bandpass, and lowpass-to-bandstop transformations of filters. All these functions, taken together, provide broad capability for analyzing and designing filters for road-profiler applications.

The functions are written in the C programming language, and their source code is presented in Appendix B. This set of functions can, with minor changes, be integrated into the Road Profile Analyzer program described later in this report.

6 Configuration and Application of the KDOT Profilers

During our visit to the KDOT Bureau of Materials & Research on 16 June 1999, Mr. Albert Oyerly, Pavement Management Specialist at KDOT, provided a brief overview of KDOT's Pavement Management System (PMS) and let us examine the sensor setup on KDOT's high-speed profilers.

6.1 Summary Notes

The following list is a brief summary based on answers to our questions asked during the June 1999 meeting. These notes are generally orientational in nature.

1. KDOT's high-speed profilers are examples of the so-called South Dakota road profiler. That is, their design is the same in basis as the profiler-design presented in [4]. KDOT's profilers employ laser rangefinders instead of the ultrasonic rangefinders used in the original South Dakota design.

2. The sensor, data-acquisition, recording, and display equipment on the KDOT profilers are manufactured by International Cybernetics Corporation, 10801-B Endeavour Way, Largo, FL 34647 (phone +1-727-547-0696). KDOT provided a copy of ICC's MDR 4080/4097 Mobile Data Recorder Operation Manual, which describes how to use the ICC system.

3. The three sensor locations used on the profiler include (1) the midline of the profiler's left-wheel path, (2) the midline of the profiler's right-wheel path, and (3) the profiler's lateral centerline.

4. Typical length of a profile-run is 20 miles, although lengths can range between roughly 0.75 mi and 40 mi. One-mile segments are run, and 0.1-mi averages are taken.

5. The county-milepost system is used to mark starting and stopping positions on profile-runs.

6. Lateral position of the profiler within the lane traveled is not marked. The driver is told to follow the evident wheel path of the lane.

7. Maximum speed of the profiler during a run is held to 70 mi/h. Minimum speed is 35 mi/h. (The ICC profiling system will not record data at speeds of 15 mi/h or less.)

8. The sampling interval used is 3 in.

9. The International Roughness Index (IRI) is the only roughness statistic used by KDOT in the PMS application.

10. Types of pavement being profiled include asphalt, concrete, composite, and some brick.

6.2 The Sensors and Their Mounting

The vehicles used by KDOT to accommodate the profiler systems are Ford Club Wagon XLT 1-ton vans.

The sensor sets each consist of a single-axis accelerometer and a laser rangefinder. (A rangefinder without accelerometer is used on the profiler's lateral centerline.) The sensors in each set mount to a subchassis, which in turn mounts directly to a channel that replaces the vehicle's front bumper.

No mechanical filter is employed between each sensor set's subchassis and the mounting channel. Likewise, there is no mechanical filter isolating the mounting channel from the vehicle's frame. Thus, the tires and the ride-suspension on the vehicle are the only mechanical filters isolating the sensors from the roadbed.

6.2.1 The Accelerometer

The two alternative accelerometers found on KDOT's road profilers are:

1. Single-axis, ±2g accelerometer manufactured by Jewell Instruments. Markings on the case are:

   Jewell Instruments
   Jewell Inertial Sensor
   Manchester NH
   LCA-165-2G
   HX
   503 023

   We were unable to obtain specifications for this accelerometer.

2. Single-axis, ±2g accelerometer manufactured by Columbia Research Laboratories, Inc., 1925 Mac Dade Blvd., Woodlyn, PA 19094 (phone +1-800-813-8471; fax +1-610-872-3882). Model number is SA-101HP.

   Following are the specifications for the Model SA-101HP accelerometer.

Operational	
Output Voltage	±7.5 V into 100K load
Output Impedance	Less than 5000Ω
Excitation	15 VDC at less than 15 mA
Accuracy$^{(1)}$	±0.075% F.R.
Repeatability	Larger of ±0.002% F.R. and 150 μg
Case Alignment	0.25 degree max.
Cross Axis Sensitivity	0.002g/g exclusive of case alignment
Natural Frequency	50 Hz to 300 Hz, dependent upon range

Damping	0.8 ± 0.2
Zero Bias	±0.05% F.R.
Bias Temp Coefficient	0.0006% F.R./°C
Noise	< 0.02% F.R. within instr. passband
Scale Factor Temp Coef.	0.02%/°C max.
Threshold and Resolution	0.0005% F.R.
Environmental	
Operating Temperature	−50°C to +90°C
Storage Temperature	−60°C to +100°C
Vibration Survival (2–2000 Hz)	20g rms, 1.0-in disp.
Shock	250g, 5 ms
Humidity	95% R.H.
Physical	
Weight	4 oz max.
Size	2.6 in L × 1.1 in W × 1.705 in H
Case Material	Nickel-plated aluminum
Sealing	Environmental
Connector	PT1H-10-6P or equivalent

Note (1): Includes nonlinearity, hysteresis and nonrepeatability.

6.2.2 The Rangefinder

The rangefinder used in KDOT's road profilers is manufactured by Selcom Laser Measurements. The U.S. contact is Selective Electronics Ave., 21654 Melrose, Southfield, MI 48075 (phone +1-810-355-5900; fax +1-810-355-3283).

The model used is the 809516 OPTOCATOR Type 2207-200/325-K, one of several sensors offered by Selcom for measurement of texture depth, macrotexture, rut, and roughness.

Specifications of the 809516 include:

Measuring Range	200 mm (7.9 in)
Standoff	325 mm (13 in)
Sampling Rate	32 kHz
Bandwidth	10 kHz

Noise at 10 kHz	0.3 mm rms
Inaccuracy	$\pm$0.4 mm (0.016 in)
Spot size diameter	Approx. 3 mm
Scale factor (1 LSB)	1/4000 of MR (12-bit data)
Resolution	0.025% of MR; 0.0015% of MR by sig. proc.
Repeatability	$\pm$0.2% of MR at 2-kHz BW
	$\pm$0.006% at 5-Hz BW
Precision	$\pm$0.01% of MR at 5-Hz BW
Nonlinearity	$\pm$0.025% of MR with mat-white target
Temperature Stability	Scale-factor change:
	Typical, $<$ 0.005% of MR per °C
	Maximum, 0.01% of MR per °C
Response Time (90%)	32 μs for 10-kHz BW
Surface Reflectivity Compens.	Fully automatic. Dynamic range 5×10^6.
	Compensation speed down to 8 μs, 100%

7 Software Development for the Road Profile Analyzer Program

Much of the work on this project involved the development of a computer program to simulate an inertial profiler and perform various calculations on sample profile data. The program is partially complete.

7.1 Current Capabilities

The Road Profile Analyzer (RPA) program currently has the ability to import profile data from Mobile Data Recorder 4080/4097 (MDR) files, the type used by KDOT. It can plot the profiles described in these files. The program calculates IRI in the standard units of m/km, in/mi, and pure slope.

7.2 Overview of the Program

The RPA program was developed using Microsoft^R Visual C++. It uses SigmaPlot^R, a plotting program distributed by SPSS Incorporated. The following paragraphs overview individual parts of the program.

7.2.1 Core and User-Interface

CRPAApp class The CRPAApp class is the parent class of the Road Profile Analyzer application. A CRPAApp object is created, and it calls functions as needed to operate. Most of the code in this section was generated automatically by the Visual C++ AppWizard. The only significant changes made to the Microsoft code were the addition of global variables for controlling SigmaPlot and of a routine to ensure that only one instance of the RPA program is running at a time.

CRPADoc class The CRPADoc class handles the data storage chores for the RPA program. As with the CRPAApp class, much of the code in the CRPADoc class was generated automatically by Visual C++. This class holds a CProfileData object that stores information about the profile being analyzed. The CRPADoc class also includes both code to handle certain menu commands and the code to save files. The code that sends profile information to SigmaPlot is in this class.

CRPAView class The CRPAView class handles the drawing of the RPA program window. This class was also generated by Visual C++. Code was added to display the SigmaPlot profile graph in the RPA program window. This class also contains functions to handle certain menu commands.

7.2.2 IRI Algorithm

CRPFile and CERDFile classes These classes control the reading of data from MDR-format files and ERD-format files, respectively. They contain variables for profile attributes such as sampling interval and profile height. The classes also have functions for parsing input files and for passing data to the CProfileData class.

CMatrix class The development of the IRI algorithm involved several steps. First, the CMatrix class was developed. This class creates a matrix object and defines various mathematical operations for the object. Using the CMatrix class, one can perform operations such as $\boldsymbol{A} + \boldsymbol{B}$ for matrices exactly as one would for scalars. This makes the code for IRI calculations very simple and clean.

CProfileData class The CProfileData class is the parent class for all IRI data and calculations. It contains a CIRIProfileArray object, which holds a CIRIProfile object (described below) for each of the channels in the profile being analyzed. The CProfileData object also holds information such as number of channels, number of samples, and sampling interval. CProfileData can be serialized. Serialization is a commonly used method for saving data in Visual C++ Programs. Code to serialize objects when saving a file is already built into a wizard-generated Visual C++ program.

CIRIProfile class The CIRIProfile class contains the important data for road profiles, as well as the functions needed to calculate IRI. The class holds values of the sampling interval, the number of samples, and the height at each sample point. CIRIProfile assumes that all distance measurements are in meters and all slope values are in m/km. The helper classes CLength and CSlope allow conversion between units.

CLength and CSlope classes These classes hold the value of a length and a slope, respectively. Each class will give its value in meters when cast as a `double` or a reference to a `double`. When cast as a `CString`, the class will return a string with the numerical value of the selected units followed by the appropriate abbreviation. This simplifies implementing dialog boxes in the Visual C++ environment.

CObjectArray, CIRIProfileArray, and CDoubleArray classes These classes serve as wrappers for arrays of the types indicated by their respective names. In future versions of the program, these should probably be combined into a single template. There were two primary reasons for using these classes to implement arrays. First, using these classes instead of regular C-style arrays allows the arrays to shrink and grow dynamically. Second, the classes can be serialized. Serialization is explained in the discussion of the CProfileData class above.

7.2.3 Plotting with SigmaPlot

The Road Profile Analyzer program uses SigmaPlot to plot data. Interfacing with SigmaPlot encompassed much of the time spent on program development.

The RPA program interacts with SigmaPlot through a mechanism known as *automation*. In theory, to control an automation server such as SigmaPlot, one can simply import a type library file, included with SigmaPlot, into the Visual C++ program. However, this procedure did not work correctly with SigmaPlot. The type library file is essentially a table that maps function names and parameter information to a numeric code that the automation server understands. After much trial and error, we discovered that numerical codes in the SigmaPlot type library did not match the codes compiled into SigmaPlot. This could be due to SPSS Software including a type library for an older version of the program. To correct the problem, we disassembled the type library using tools included with Visual C++. We then found the numerical codes using Visual C++'s `GetIDsOfNames()` function. We entered the codes into the type library source and recompiled the library. With this new type library, SigmaPlot automation worked properly.

The Road Profile Analyzer program uses *OLE embedding* to display graphs from SigmaPlot inside the program window. Some problems were encountered with the embedding process. When a program uses OLE embedding, it "locks" itself once for each open embedded object. When the program exits, it closes each embedded object and removes the associated lock. For unknown reasons, the Road Profile Analyzer program was not removing the lock for the profile plot. This caused the program to "hang" in the background after the user closed it. As a temporary work-around, the program now counts the number of plots opened and removes the appropriate number of locks when the user exits the program. This problem should be investigated in future versions of the program.

Currently, the RPA program passes profile data to SigmaPlot via SigmaPlot's *PutCell* method. The SigmaPlot documentation notes that this is a slow method of passing data. It recommends the *PutData* method, which passes an array of `variant` data type. However, a `variant` uses twice the memory of type `double`. Also, implementing `variant` arrays in Visual C++ requires significant code overhead. However, the profile-plotting process is currently quite slow, especially for large data files. Future versions of the program should investigate ways of using the *PutData* method efficiently.

7.3 Future Development of the Program

As mentioned above, the program will eventually simulate the entire profile-measurement process. It will create a "true" profile and simulate the measurement of that profile. The sampled-profile values can then be compared with the actual profile values. The program will be able to compare various statistics, such as IRI, from the actual and the measured profiles. This data will help point out possible sources of error in the measurement process. Also, the plotting capabilities of the program will be improved in future versions.

With the recent (June 2004) release of MATLAB R14 and attendant upgrades to the MATLAB Compiler, it is now possible to compile source code and user interfaces into freely distributable runtime packages. This allows for the ability to rapidly develop new functionality by harnessing the advanced matrix-manipulation and graphing capabilities of MATLAB. For future work, this is an option that should be seriously considered.

8 Operation of the Road Profile Analyzer Program

This section briefly describes how to operate the Road Profile Analyzer program. Currently, available features include importing MDR-compatible profiles, plotting the profile data, and calculating IRI for any particular section of the profile.

8.1 Importing MDR-format Files

To allow importation of a file in the MDR format, the RP090L program must be installed in the C:\MDRSW directory. Figure 4 shows the "Import MDR Profile" command, which is located on the File menu. Choosing this command brings up a standard File Open dialog box. From this box, find and select the file you wish to import.

8.2 Plotting Profile Data

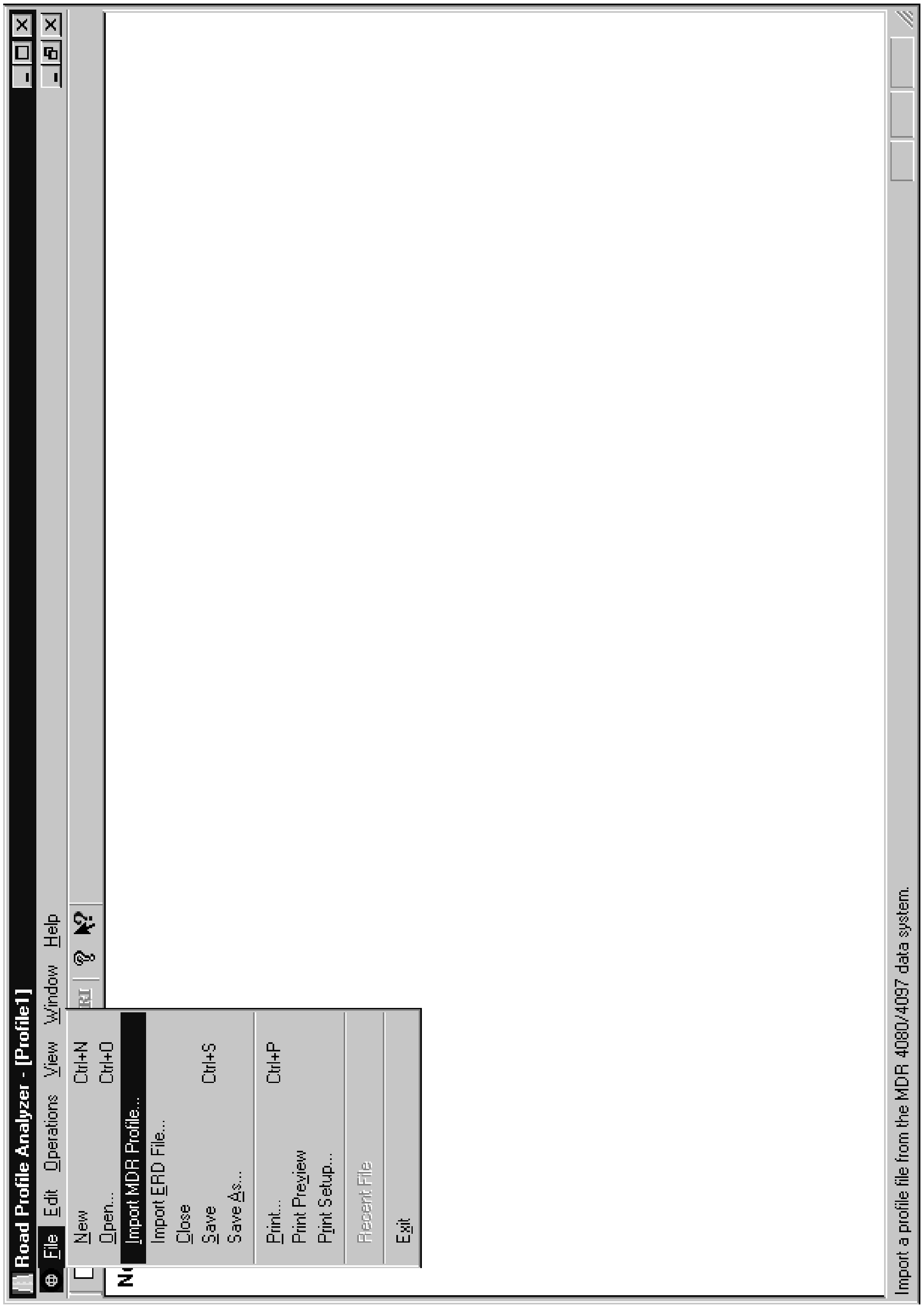
The Road Profile Analyzer program automatically displays a plot of profile height versus horizontal position when data is loaded. Importing an MDR file or opening a previously saved Road Profile Analyzer file will cause the program to display a profile plot. Figure 5 shows a sample profile plot.

8.3 Calculating IRI

The Road Profile Analyzer will calculate the IRI of each channel of the input profile. The command to calculate the IRI is located in the operations menu, as shown in Figure 6. The resulting dialog box, shown in Figure 7, allows the user to specify starting and stopping points for the IRI, as well as units for the output. Figure 8 shows the output of an IRI calculation.

8.4 Example Runs

Figure 8 shows the results of an IRI calculation on an actual KDOT file, `17731nps.p01`. Figures 9 and 10 show the results of IRI calculation on files `17731ndr.p01` and `3170dr.p01`, respectively.



Import a profile file from the MDR 4080/4097 data system.

Figure 4: The “Import MDR Profile” command.

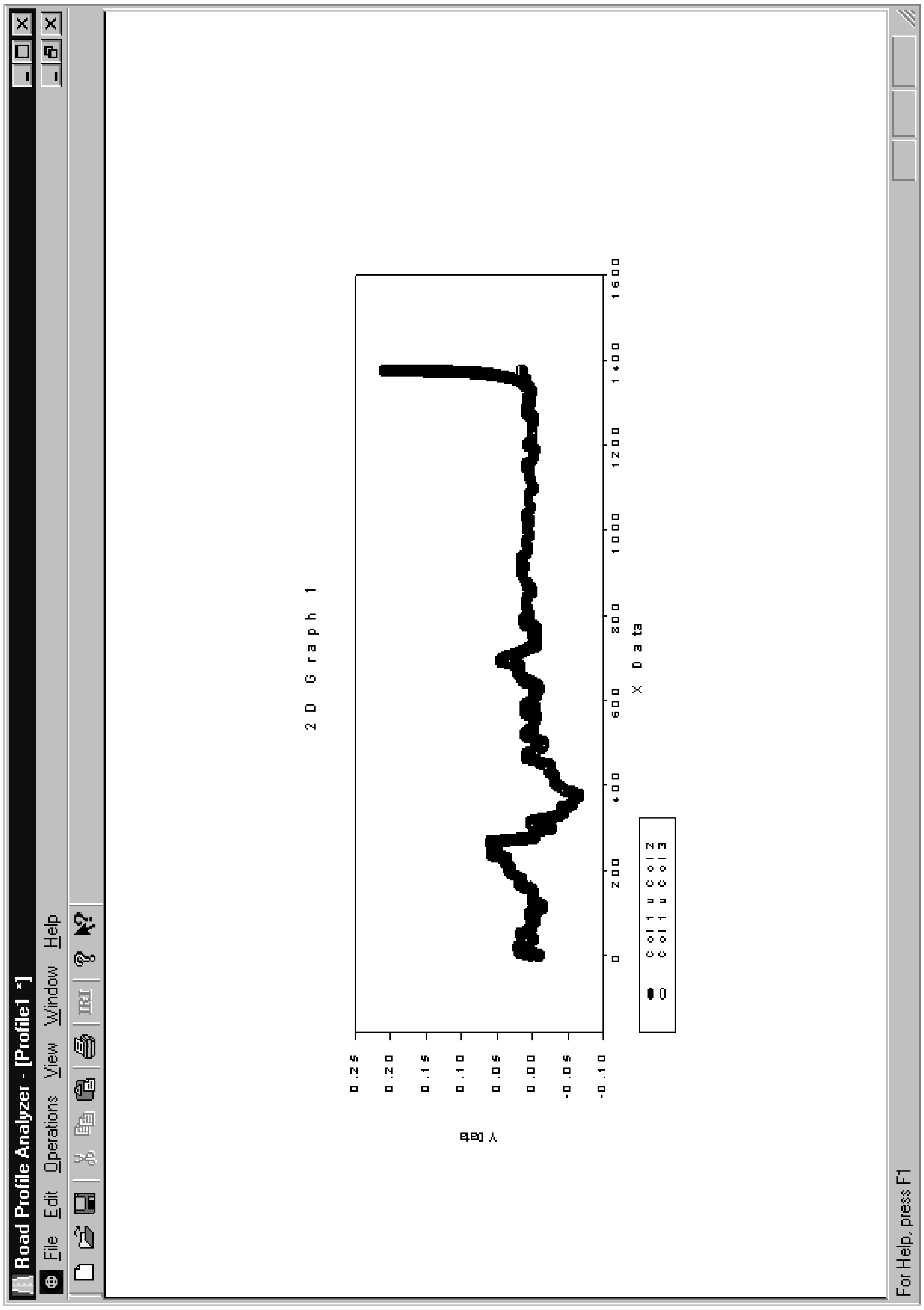


Figure 5: An example profile plot.

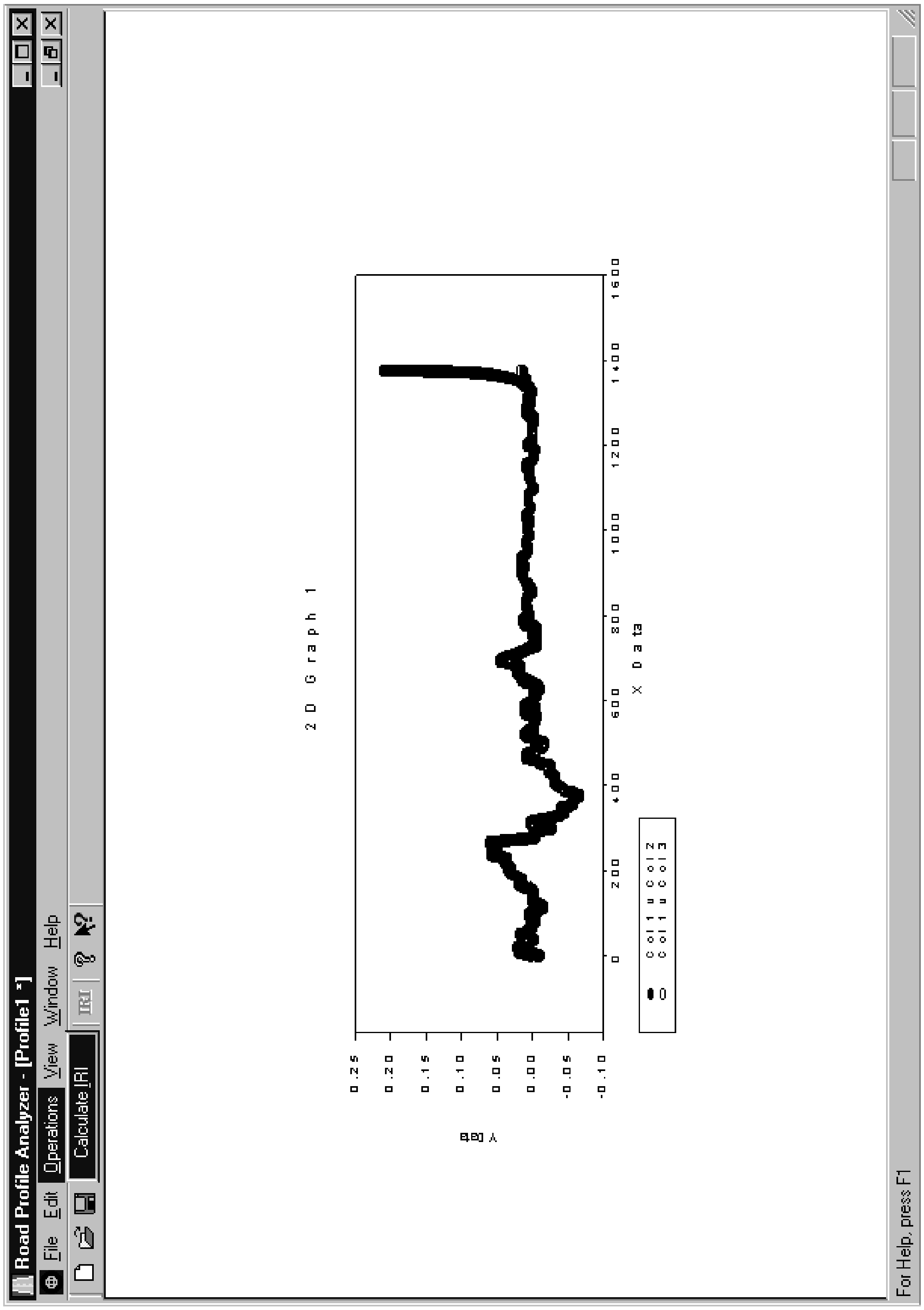
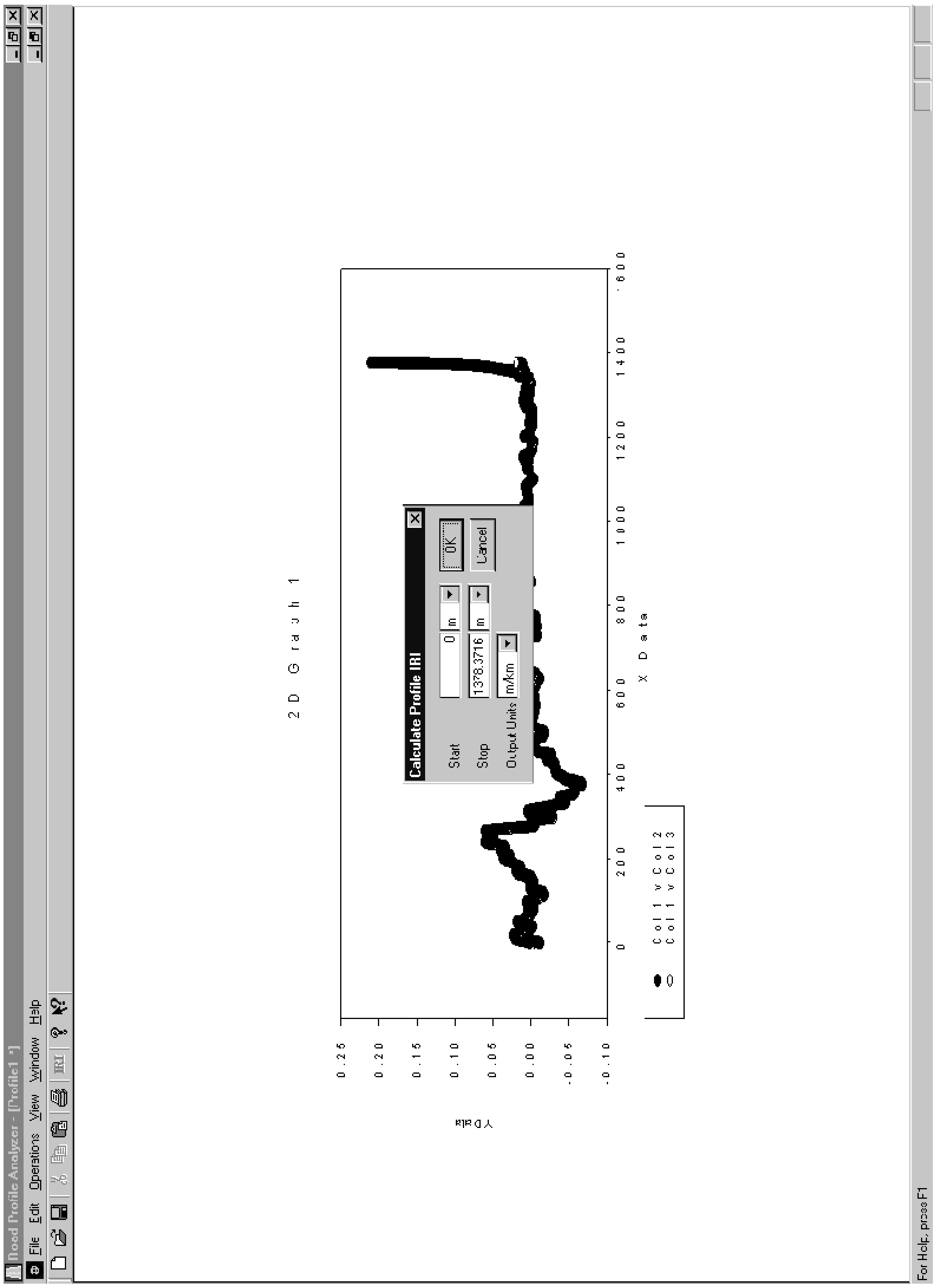


Figure 6: The “Calculate IRI” command.



Calculate Profile IRI

Start 0 m

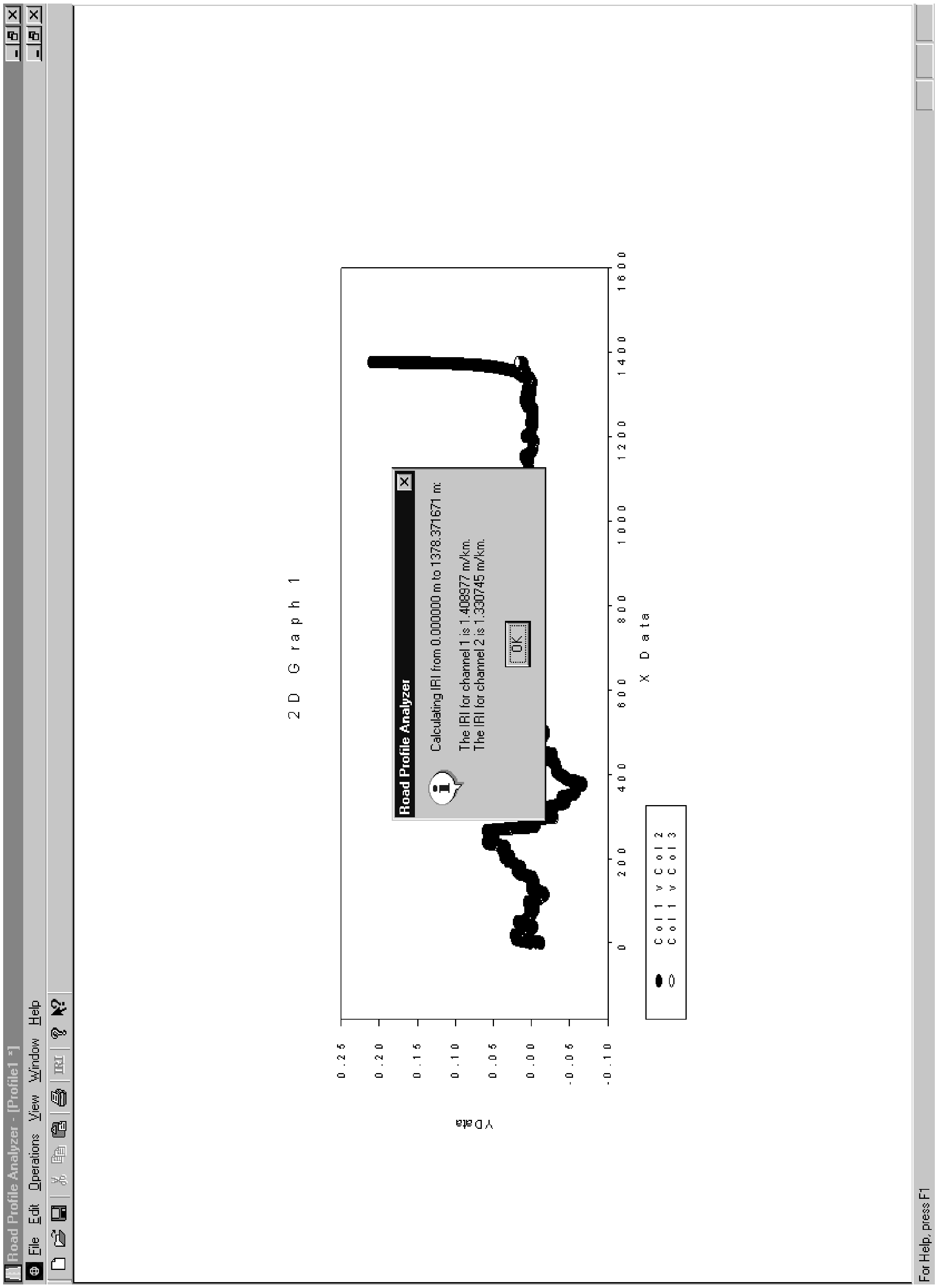
Stop 1378.3716 m

Output Units m/km

OK

Cancel

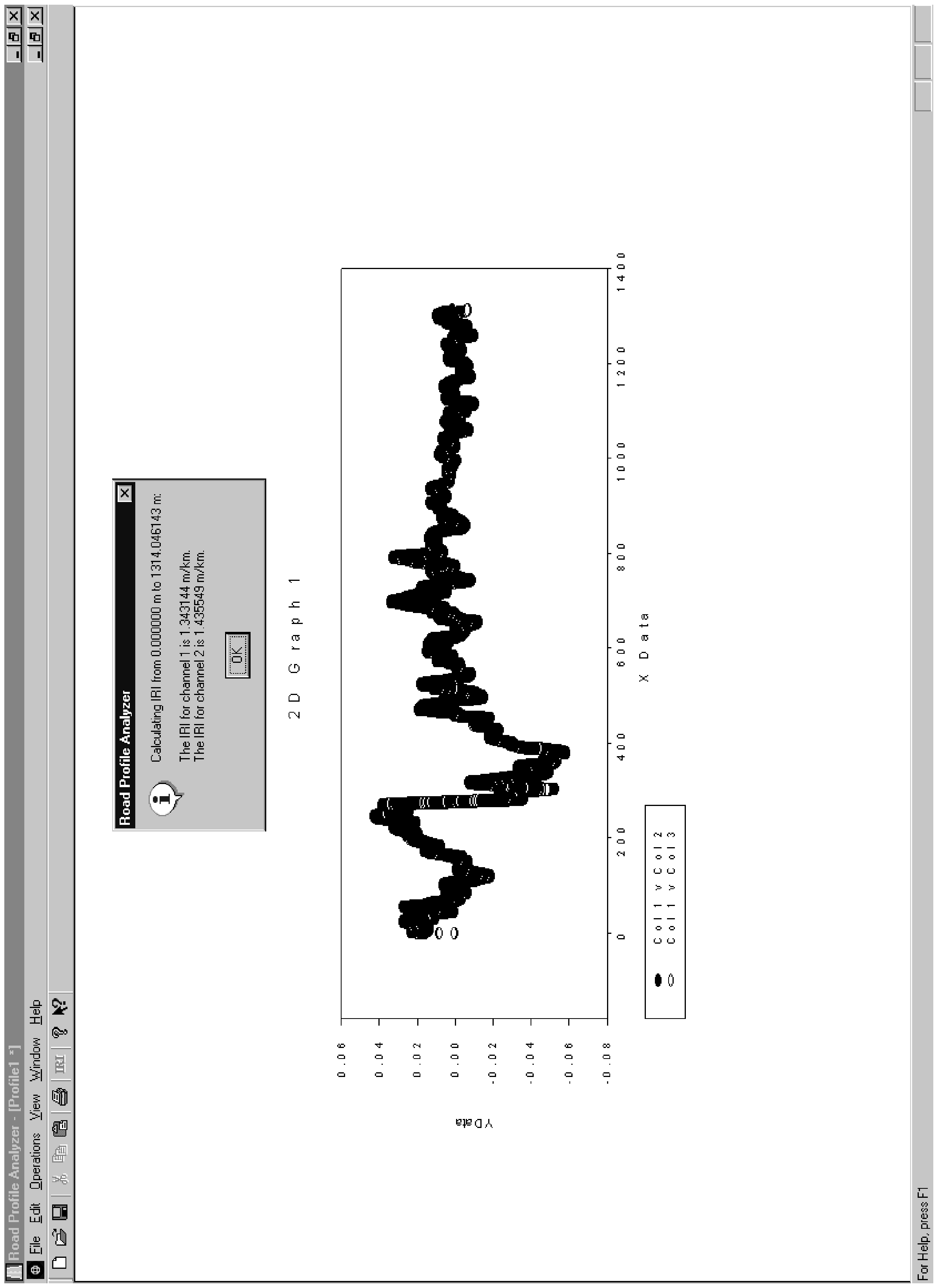
Figure 7: The IRI calculation dialog box.



Calculating IRI from 0.000000 m to 1378.371671 m:

The IRI for channel 1 is 1.408977 m/km.
The IRI for channel 2 is 1.330745 m/km.

Figure 8: The results of an IRI calculation.

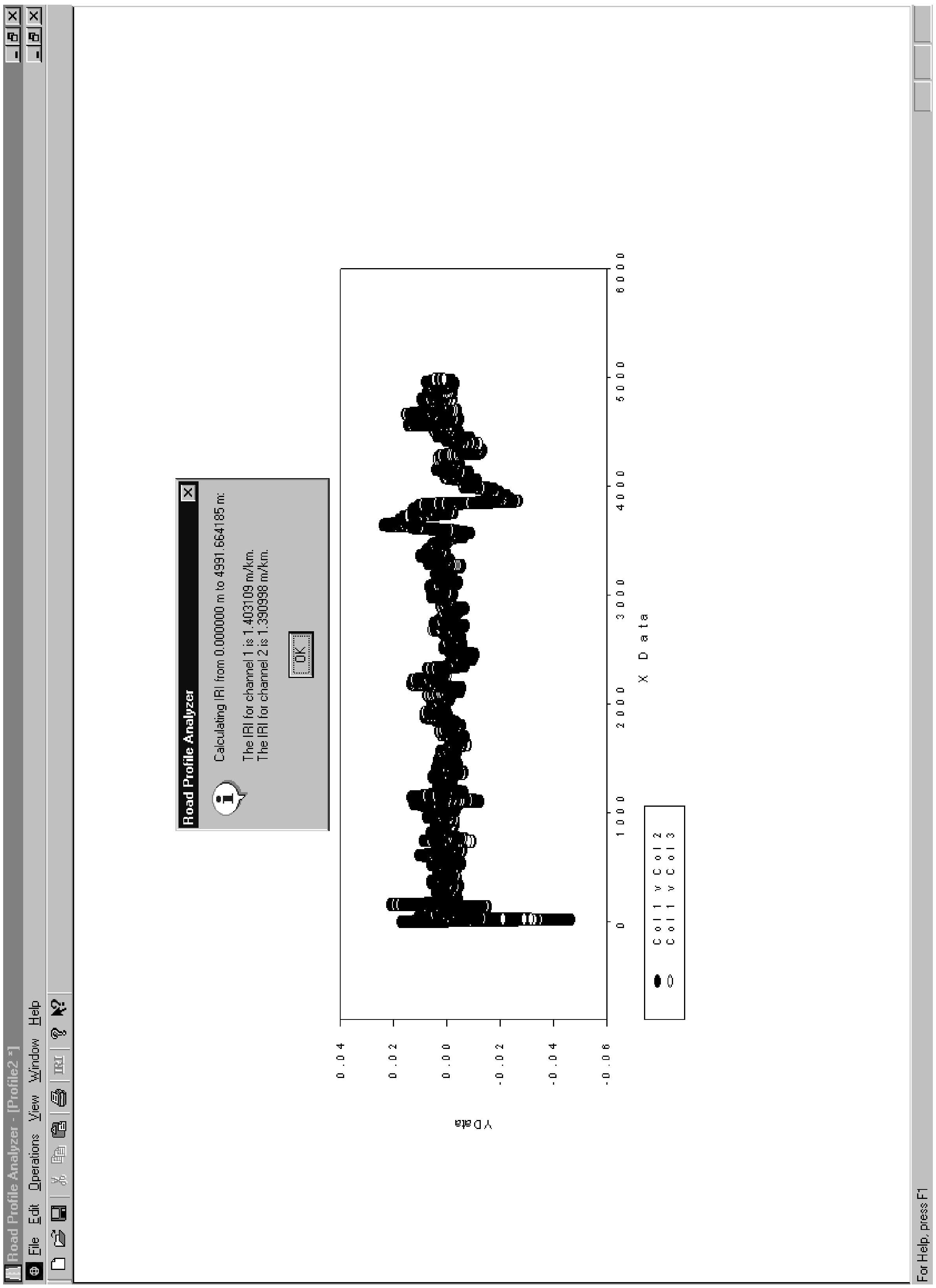


Road Profile Analyzer

Calculating IRI from 0.000000 m to 1314.046143 m:

The IRI for channel 1 is 1.343144 m/km.
The IRI for channel 2 is 1.435549 m/km.

Figure 9: Results of IRI calculation on `17731ndr.p01`.



Road Profile Analyzer

Calculating IRI from 0.000000 m to 4991.664185 m:

The IRI for channel 1 is 1.403109 m/km.
The IRI for channel 2 is 1.390998 m/km.

Figure 10: Results of IRI calculation on `3170dr.p01`.

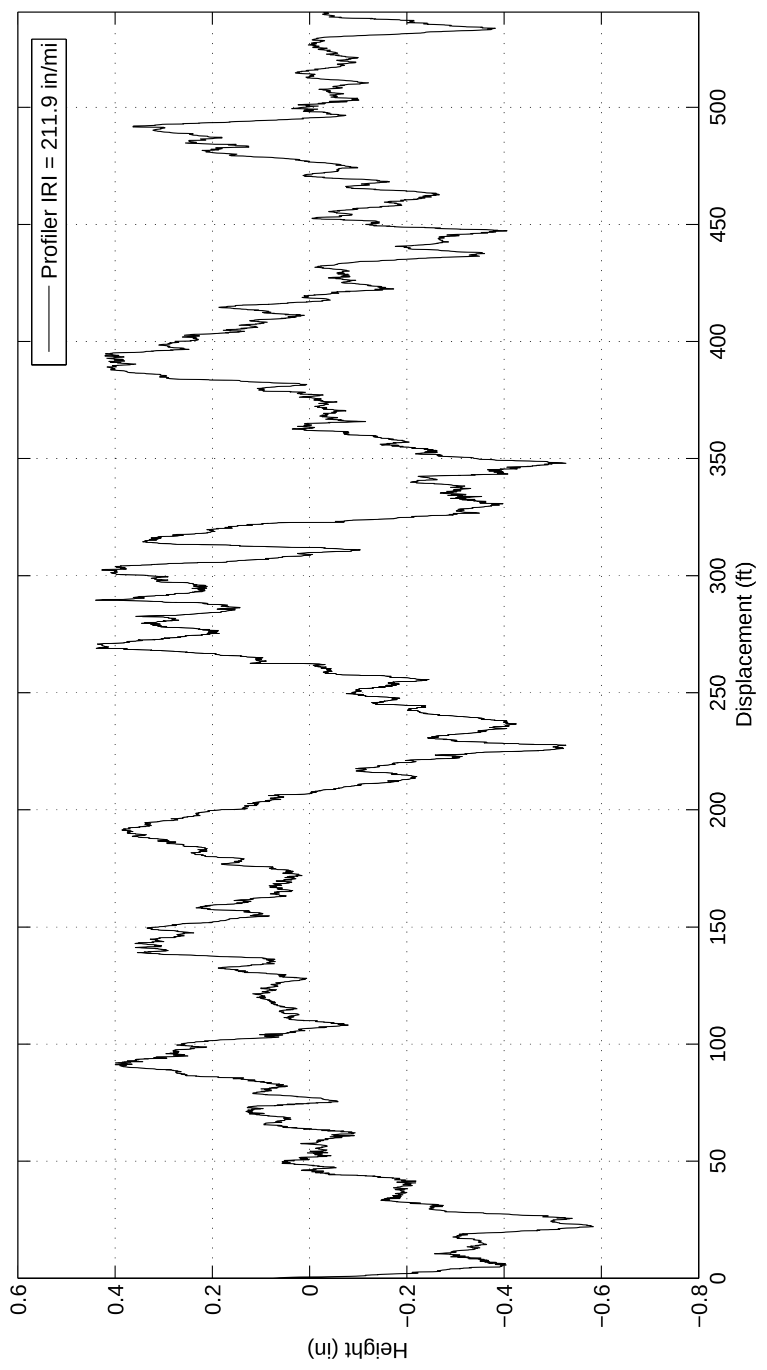


Figure 11: Sample MATLAB–produced profile graph and IRI calculation.

9 Conclusion

K-TRAN Project KSU-99-7 was a beginning step in a systematic study of the elements affecting the performance of inertial profilers. This report has summarized the results of the effort directed toward the five project-tasks, as well as provided information on an early version of the Road Profile Analyzer software.

9.1 Recommendations

Recommendations for further work follow.

Required Range for Accelerometer It is still uncertain what range of vertical acceleration should be expected to be encountered during a profiling run. The references found and cited imply that a range of ±1g (on top of the +1g due to gravity) should be sufficient. The validity of this statement should be ascertained for the profilers in use within KDOT.

Raw acceleration data is available in the form of analog-to-digital–converter counts from the ICC Model MDR 4080/4097 Mobile Data Recorder employed in KDOT's profilers; however, lack of information from the manufacturer regarding how the double integration is performed and what additional filtering is done makes it impossible to reliably "back out" (i.e., apply inverse-filtering principles to obtain) the accelerometer-output information from the accelerometer height-data provided by the data recorder.

Our recommendation is to do one of the following to obtain raw vertical-acceleration data.

1. Request the manufacturer, International Cybernetics Corporation, to modify its software so that it will provide a record of the raw vertical-acceleration data from the accelerometer also. Representative profiling runs made over expected "worst-case" road surfaces will then indicate whether or not the ±2g accelerometers used remain within their linear range of operation.

2. Construct a system which incorporates the accelerometers used on the profilers, but which uses its own instrumentation amplifiers, data-acquisition hardware, filters, etc. Make the above-mentioned profiling runs to determine the adequacy of the ±2g range on the accelerometers.

Accelerometer-data Processing KDOT should obtain from ICC, if possible, the details of its processing of the accelerometer data. Specifically,

1. Is the accelerometer output sampled at fixed time intervals?

2. Is there any analog signal conditioning done prior to the double integration? If so, what?

3. How is the double integration accomplished, by analog or digital means? What are the details of the integrators (e.g., if numerical integration is performed, what algorithm is used)?

4. Is there any highpass filtering done to remove dc drift, etc. from the accelerometer-output data? If so, what are the details?

5. Is there any other pre- or post-filtering done to the doubly integrated accelerometer-output data? If so, what?

6. Under what conditions does the accelerometer saturate?

7. How is the accelerometer data handled in the case where the accelerometer saturates?

Rangefinder-data Processing KDOT should obtain from ICC, if possible, the details of its processing of the rangefinder data. Specifically,

1. How is the rangefinder data acquired, by sampling at fixed time intervals or by sampling based on longitudinal distance traveled?

2. What sort of filtering is done to the rangefinder data before using it as height information?

Profile-acquisition Procedure KDOT should review its procedure for acquiring profiles. The review should be designed to assure that the procedure itself promotes good consistency of data acquired on repeated runs. As a minimum, it should ascertain that the procedures enumerated in Section 9 of ASTM E 950-94 are followed.

ASTM E 950-94 KDOT should investigate the possibility of obtaining a seat on the ASTM Committee on Vehicle–Pavement Systems, which has jurisdiction over the ASTM E 950-94 standard, to provide input on deficiencies in the current version of the standard.

Climate-related Variations in IRI The effect of environmental conditions on variations in IRI should be studied. A literature search of previous studies and a summary of their findings should provide indication as to whether or not climate should be expected to factor significantly into the variation in IRI observed by KDOT.

A second step, if deemed warranted, could involve a correlation study, conducted on selected road segments under various environmental conditions, to corroborate the results of the studies found in the literature.

Continued Research KDOT should consider augmenting the research carried out to include tasks that were listed within the Introduction, but which were outside the scope of this project.

In addition, the recent improvements to the MATLAB programming environment should be taken into consideration as future work could take advantage of these improvements to provide a much richer feature set in further revisions to the Road Profile Analyzer software. Figure 11 shows example output from a rudimentary MATLAB–based simulation program. The plot shows a profile collected from an actual road profiler and presents the accompanying IRI statistic.

9.2 Closure

The overall research effort, when completed, should provide KDOT with the required tools and information to direct modification of its PMS program, incorporating any changes that, according to the research results, have high probability of significantly reducing the variance in future data collected through the PMS program.

References

[1] Sayers, M.W., T.D. Gillespie, and W.D.O. Paterson, *Guidelines for Conducting and Calibrating Road Roughness Measurements.* Technical Paper 46. Washington, DC: The World Bank. 1986.

[2] Sayers, M.W., *On the Calculation of IRI from Longitudinal Road Profile.* Paper no. 95 0842. Washington, DC: Transportation Research Board 74th Annual Meeting. 1995.

[3] Smith, K.L., K.D. Smith, L.D. Evans, T.E. Hoerner, M.I. Darter, and J.H. Woodstrom, *Smoothness Specifications for Pavements.* NCHRP 1-30. National Research Council. 1997.

[4] Huft, D.L., *Description and Evaluation of the South Dakota Road Profiler.* Report FHWA-DP-89-072-002. Washington, DC: U.S. Dept. of Transportation, Federal Highway Administration. 1989.

[5] ASTM E 950-94, *Standard Test Method for Measuring the Longitudinal Profile of Traveled Surfaces with an Accelerometer Established Inertial Profiling Reference.* Philadelphia, PA: American Society for Testing and Materials. 1994.

A Source Code for the Road Profile Analyzer Program

This appendix presents the relevant source code for the Road Profile Analyzer program. Some code which was generated automatically by Visual C++ is not included here. Such code includes the CMainFrame, CChildFrm, and CCntrItem classes, as well as the files StdAfx.cpp and StdAfx.h. Classes associated with dialog boxes are not shown either, as they too are almost entirely automatically generated.

The code that follows is organized by C++ class. All of the classes contained here are described in Section 7.

A.1 CRPAApp Class

```
// RPA.h : main header file for the RPA application
//

#if !defined(AFX_RPA_H__FDF284C8_4F00_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_RPA_H__FDF284C8_4F00_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

extern SigmaPlot::IJAutoApplicationPtr pSP;
extern int m_unlockHack;

/////////////////////////////////////////////////////////////////////////////
// CRPAApp:
// See RPA.cpp for the implementation of this class
//

class CRPAApp : public CWinApp
{
public:
    BOOL FirstInstance();
    CRPAApp();
    virtual ~CRPAApp();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CRPAApp)
    public:
    virtual BOOL InitInstance();
    virtual int ExitInstance();
    //}}AFX_VIRTUAL

// Implementation
    COleTemplateServer m_server;
        // Server object for document creation
    //{{AFX_MSG(CRPAApp)
    afx_msg void OnAppAbout();
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
// the previous line.

#endif // !defined(AFX_RPA_H__FDF284C8_4F00_11D3_B226_0050041CB445__INCLUDED_)
```

```
// RPA.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "RPA.h"
#include "MainFrm.h"
#include "ChildFrm.h"
#include "RPADoc.h"
#include "RPAView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// Global variables for SigmaPlot control:
SigmaPlot::IJAutoApplicationPtr pSP;
SigmaPlot::IJAutoNotebookPtr pNb;
int m_unlockHack = 0;

/////////////////////////////////////////////////////////////////////////////
// CRPAApp

BEGIN_MESSAGE_MAP(CRPAApp, CWinApp)
    //{{AFX_MSG_MAP(CRPAApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG_MAP
    // Standard file based document commands
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CRPAApp construction

CRPAApp::CRPAApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

CRPAApp::~CRPAApp()
{
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CRPAApp object

CRPAApp theApp;

// This identifier was generated to be statistically unique for your app.
// You may change it if you prefer to choose a specific identifier.

// {FDF284C3-4F00-11D3-B226-0050041CB445}
static const CLSID clsid =
{ 0xfdf284c3, 0x4f00, 0x11d3, { 0xb2, 0x26, 0x0, 0x50, 0x4, 0x1c, 0xb4, 0x45 } };

/////////////////////////////////////////////////////////////////////////////
// CRPAApp initialization

// Add a static BOOL that indicates whether the class was
// registered so that you can unregister it in ExitInstance
static BOOL bClassRegistered = FALSE;
```

```
BOOL CRPAApp::InitInstance()
{
    // If a previous instance of the application is already running,
    // then activate it and return FALSE from InitInstance to
    // end the execution of this instance.

    if(!FirstInstance())
        return FALSE;

        // Register your unique class name that you wish to use
        WNDCLASS wndcls;

        memset(&wndcls, 0, sizeof(WNDCLASS));    // start with NULL
                                                // defaults

        wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
        wndcls.lpfnWndProc = ::DefWindowProc;
        wndcls.hInstance = AfxGetInstanceHandle();
        wndcls.hIcon = LoadIcon(IDR_MAINFRAME); // or load a different
                                                // icon
        wndcls.hCursor = LoadCursor( IDC_ARROW );
        wndcls.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
        wndcls.lpszMenuName = NULL;

        // Specify your own class name for using FindWindow later
        wndcls.lpszClassName = _T("RoadProfileAnalyzer");

        // Register the new class and exit if it fails
        if(!AfxRegisterClass(&wndcls))
        {
            TRACE("Class Registration Failed\n");
            return FALSE;
        }
        bClassRegistered = TRUE;


    // Initialize OLE libraries
    if (!AfxOleInit())
    {
        AfxMessageBox(IDP_OLE_INIT_FAILED);
        return FALSE;
    }

    AfxEnableControlContainer();

    // Standard initialization
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    //  the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();            // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();    // Call this when linking to MFC statically
#endif

    // Change the registry key under which our settings are stored.
    // TODO: You should modify this string to be something appropriate
    // such as the name of your company or organization.
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    LoadStdProfileSettings();  // Load standard INI file options (including MRU)

    // Register the application's document templates.  Document templates
    //  serve as the connection between documents, frame windows and views.

    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
        IDR_RDPRFTYPE,
        RUNTIME_CLASS(CRPADoc),
        RUNTIME_CLASS(CChildFrame), // custom MDI child frame
        RUNTIME_CLASS(CRPAView));
    pDocTemplate->SetContainerInfo(IDR_RDPRFTYPE_CNTR_IP);
    AddDocTemplate(pDocTemplate);

    // Connect the COleTemplateServer to the document template.
    //  The COleTemplateServer creates new documents on behalf
    //  of requesting OLE containers by using information
    //  specified in the document template.
    m_server.ConnectTemplate(clsid, pDocTemplate, FALSE);

    // Register all OLE server factories as running.  This enables the
    //  OLE libraries to create objects from other applications.
    COleTemplateServer::RegisterAll();
        // Note: MDI applications register all server objects without regard
        //  to the /Embedding or /Automation on the command line.

    // create main MDI Frame window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    m_pMainWnd = pMainFrame;

    // Enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    // Enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);

    // Parse command line for standard shell commands, DDE, file open
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // cmdInfo.m_nShellCommand is defaulted to be FileNew,
    // change to FileNothing = -1; // RYAN_ADDED
    //cmdInfo.m_nShellCommand = CCommandLineInfo::FileNothing; // RYAN_ADDED

    // Check to see if launched as OLE server
    if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
    {
        // Application was run with /Embedding or /Automation.  Don't show the
        //  main window in this case.
        return TRUE;
    }

    // When a server application is launched stand-alone, it is a good idea
    //  to update the system registry in case it has been damaged.
    m_server.UpdateRegistry(OAT_DISPATCH_OBJECT);
    COleObjectFactory::UpdateRegistryAll();

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;

    // The main window has been initialized, so show and update it.
    pMainFrame->ShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();


//    pSP.CreateInstance(L"SigmaPlot.Application.1");

    return TRUE;
}
```

```
int CRPAApp::ExitInstance()
{
    // TODO: Add your specialized code here and/or call the base class

//    CoUninitialize();

    if(pSP != NULL)
    {
        pSP->Visible = VARIANT_TRUE;    // DEBUG

        // Quit SigmaPlot:
        pSP->Quit();
    }

    if(bClassRegistered)
        ::UnregisterClass(_T("RoadProfileAnalyzer"),AfxGetInstanceHandle());

    return CWinApp::ExitInstance();
}

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    //{{AFX_MSG(CAboutDlg)
        // No message handlers
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // No message handlers
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CRPAApp::OnAppAbout()
```

```
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////////////////
// CRPAApp message handlers



BOOL CRPAApp::FirstInstance()
{

    CWnd *pWndPrev, *pWndChild;

    // Determine if another window with your class name exists...
    if (pWndPrev = CWnd::FindWindow(_T("RoadProfileAnalyzer"),NULL))
    {
        // If so, does it have any popups?
        pWndChild = pWndPrev->GetLastActivePopup();

        // If iconic, restore the main window
        if (pWndPrev->IsIconic())
            pWndPrev->ShowWindow(SW_RESTORE);

        // Bring the main window or its popup to
        // the foreground
        pWndChild->SetForegroundWindow();

        // and you are done activating the previous one.
        return FALSE;
    }
    // First instance. Proceed as normal.
    else
        return TRUE;
}
```

A.2 CRPADoc Class

```
// RPADoc.h : interface of the CRPADoc class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_RPADOC_H__FDF284D1_4F00_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_RPADOC_H__FDF284D1_4F00_11D3_B226_0050041CB445__INCLUDED_

#include "ERDFile.h"    // Added by ClassView
#include "ProfileData.h"    // Added by ClassView
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CRPADoc : public COleDocument
{
protected: // create from serialization only
    CRPADoc();
    DECLARE_DYNCREATE(CRPADoc)

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CRPADoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    //}}AFX_VIRTUAL

// Implementation
public:
    void SendProfileToSigmaPlot();
    bool plotCurrent;
    void SetModifiedFlag(BOOL bModified = TRUE);
    CProfileData* m_pProfileData;
    virtual ~CRPADoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CRPADoc)
    afx_msg void OnFileImportERDFile();
    afx_msg void OnOperationsCalculateIRI();
    afx_msg void OnFileImportMDRProfile();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()

    // Generated OLE dispatch map functions
    //{{AFX_DISPATCH(CRPADoc)
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
    DECLARE_INTERFACE_MAP()
};
```

```
/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
// the previous line.

#endif // !defined(AFX_RPADOC_H__FDF284D1_4F00_11D3_B226_0050041CB445__INCLUDED_)
```

```
// RPADoc.cpp : implementation of the CRPADoc class
//

#include "stdafx.h"
#include "RPA.h"

#include "RPADoc.h"
#include "RPAView.h"
#include "CntrItem.h"
#include "RPFile.h"    // DEBUG

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

extern SigmaPlot::IJAutoApplicationPtr pSP;
//extern IJAutoApplication SP;
extern SigmaPlot::IJAutoNotebookPtr pNb;
//extern IJAutoNotebook Nb;

/////////////////////////////////////////////////////////////////////////////
// CRPADoc

IMPLEMENT_DYNCREATE(CRPADoc, COleDocument)

BEGIN_MESSAGE_MAP(CRPADoc, COleDocument)
    //{{AFX_MSG_MAP(CRPADoc)
    ON_COMMAND(ID_FILE_IMPORTERDFILE, OnFileImportERDFile)
    ON_COMMAND(ID_OPERATIONS_CALCULATEIRI, OnOperationsCalculateIRI)
    ON_COMMAND(ID_FILE_IMPORTMDRPROFILE, OnFileImportMDRProfile)
    //}}AFX_MSG_MAP
    // Enable default OLE container implementation
    ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE, COleDocument::OnUpdatePasteMenu)
    ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE_LINK, COleDocument::OnUpdatePasteLinkMenu)
    ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_CONVERT, COleDocument::OnUpdateObjectVerbMenu)
    ON_COMMAND(ID_OLE_EDIT_CONVERT, COleDocument::OnEditConvert)
    ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_LINKS, COleDocument::OnUpdateEditLinksMenu)
    ON_COMMAND(ID_OLE_EDIT_LINKS, COleDocument::OnEditLinks)
    ON_UPDATE_COMMAND_UI_RANGE(ID_OLE_VERB_FIRST, ID_OLE_VERB_LAST,
        COleDocument::OnUpdateObjectVerbMenu)
END_MESSAGE_MAP()

BEGIN_DISPATCH_MAP(CRPADoc, COleDocument)
    //{{AFX_DISPATCH_MAP(CRPADoc)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //      DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

// Note: we add support for IID_IRPA to support typesafe binding
//  from VBA.  This IID must match the GUID that is attached to the
//  dispinterface in the .ODL file.

// {FDF284C5-4F00-11D3-B226-0050041CB445}
static const IID IID_IRPA =
{ 0xfdf284c5, 0x4f00, 0x11d3, { 0xb2, 0x26, 0x0, 0x50, 0x4, 0x1c, 0xb4, 0x45 } };

BEGIN_INTERFACE_MAP(CRPADoc, COleDocument)
    INTERFACE_PART(CRPADoc, IID_IRPA, Dispatch)
END_INTERFACE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CRPADoc construction/destruction

CRPADoc* junk;
CRPADoc::CRPADoc() : m_pProfileData(NULL), plotCurrent(false)
```

```
{
    // Use OLE compound files
    EnableCompoundFile();

    // TODO: add one-time construction code here
    junk = this;

    EnableAutomation();

    AfxOleLockApp();
}

CRPADoc::~CRPADoc()
{
    AfxOleUnlockApp();
    if(m_pProfileData != NULL)
        delete m_pProfileData;
}

BOOL CRPADoc::OnNewDocument()
{
    if (!COleDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)

    return TRUE;
}



/////////////////////////////////////////////////////////////////////////////
// CRPADoc serialization

void CRPADoc::Serialize(CArchive& ar)
{
//    m_pProfileData->Serialize(ar);

    if (ar.IsStoring())
    {
        // TODO: add storing code here
        ar << m_pProfileData;
    }
    else
    {
        // TODO: add loading code here
        ar >> m_pProfileData;
    }

    // Calling the base class COleDocument enables serialization
    //  of the container document's COleClientItem objects.
    COleDocument::Serialize(ar);
}

/////////////////////////////////////////////////////////////////////////////
// CRPADoc diagnostics

#ifdef _DEBUG
void CRPADoc::AssertValid() const
{
    COleDocument::AssertValid();
}

void CRPADoc::Dump(CDumpContext& dc) const
{
    COleDocument::Dump(dc);
}
```

```
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CRPADoc commands

void CRPADoc::OnFileImportERDFile()
{
    if(m_pProfileData != NULL)
    {
        if(AfxMessageBox(
          "This will destroy your current profile data.\nDo you still wish to continue?",
          MB_YESNO) == IDNO)
            return;
        delete m_pProfileData;    // DEBUG
    }

    CERDFile* pERDFile = new CERDFile;
    if(!pERDFile->Import())
        return;

    m_pProfileData = new CProfileData;

    m_pProfileData->SetNumberOfChannels(pERDFile->GetNChan());
    m_pProfileData->SetNumberOfSamples(pERDFile->GetNSamp());
    m_pProfileData->SetSampleInterval(pERDFile->GetStep());
    for(int n = 0; n < pERDFile->GetNChan(); n++)
        m_pProfileData->SetProfile(n, pERDFile->GetProfile(n));

    plotCurrent = false;
    SetModifiedFlag();
    UpdateAllViews(NULL);

    delete pERDFile;
}

void CRPADoc::OnOperationsCalculateIRI()
{
    if(m_pProfileData == NULL)
    {
        AfxMessageBox("There is no profile data.");
        return;
    }

    CIRIDialog dlg(*m_pProfileData);
    CString tmpString, mesg, mesgtail;
    //int n;//, nChan;

    if(dlg.DoModal() == IDOK)
    {
        BeginWaitCursor();

        CLength start, stop;
        start = dlg.m_start;
        stop = dlg.m_stop;

        //nChan = m_pProfileData.GetNumberOfChannels();

        for(int n = 0; n < /*nChan*/m_pProfileData->GetNumberOfChannels(); n++)
        {
            CSlope IRI(m_pProfileData->GetChannelIRI(n, start, stop), CSlope::pureSlope);
            IRI.SetUnits(dlg.m_units);
            tmpString.Format("The IRI for channel %d is %s.\n", n+1, (CString)IRI);
            mesgtail += tmpString;
        }
        mesg.Format("Calculating IRI from %s to %s:\n\n", (CString)start, (CString)stop);
        mesg += mesgtail;

        EndWaitCursor();
        AfxMessageBox(mesg, MB_OK | MB_ICONINFORMATION);
    }
}

void CRPADoc::SetModifiedFlag(BOOL bModified)
{
    CDocument::SetModifiedFlag(bModified);
    CString title = GetTitle();
    if(bModified == TRUE && title.Right(1) != '*')
    {
        title += " *";
        SetTitle(title);
    }
    else if(bModified == FALSE && title.Right(1) == '*')
    {
        title.Delete(title.GetLength() - 2, 2);
        SetTitle(title);
    }
}

void CRPADoc::SendProfileToSigmaPlot()
{
    if(!plotCurrent)
    {
        // Plot the data:
        using namespace SigmaPlot;
//        IJAutoNotebooksPtr pNbs(pSP->Notebooks);
//        IJAutoNotebookPtr pNb(pNbs->Add());
//        IJAutoNotebookItemsPtr pItems(pNb->NotebookItems);
        IJAutoNativeWorksheetItemPtr pWS(pNb->CurrentDataItem);
        //IJAutoNativeWorksheetItem WS(Nb.GetCurrentDataItem());
        IJAutoDataTablePtr pDT(pWS->DataTable);
        //IJAutoDataTable DT(WS.GetDataTable());

        int i, j;
        const int nChan = m_pProfileData->GetNumberOfChannels();
        const int nSamp = m_pProfileData->GetNumberOfSamples();
        const double delta = m_pProfileData->GetSampleInterval();

//        pDT->Cell[0][0] = "X position";

//        for(i = 1; i <= nChan; i++)
//        {
//            CString label;
//            label.Format("Channel %d", i);
//            pDT->Cell[i][0] = (LPCSTR)label;
//        }

        for(j = 0; j < nSamp; j++)
            pDT->Cell[0][j] = j * delta;
            //DT.SetCell(0, j, COleVariant(j * delta));

        for(i = 0; i < nChan; i++)
            for(j = 0; j < nSamp; j++)
                pDT->Cell[i+1][j] = m_pProfileData->GetSamplePoint(i, j);
                //DT.SetCell(i+1, j, COleVariant(m_pProfileData->GetSamplePoint(i, j)));

//        plotCurrent = true;
    }
}

void CRPADoc::OnFileImportMDRProfile()
{
    if(m_pProfileData != NULL)
    {
        if(AfxMessageBox(
          "This will destroy your current profile data.\nDo you still wish to continue?",
          MB_YESNO) == IDNO)
            return;
        delete m_pProfileData;    // DEBUG
    }
    CRPFile* pRPFile = new CRPFile;

    if(!pRPFile->Import())
        return;

    m_pProfileData = new CProfileData;

    pRPFile->GetProfileData(*m_pProfileData);
}
```

A.3 CRPAView Class

```
// RPAView.h : interface of the CRPAView class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_RPAVIEW_H__FDF284D3_4F00_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_RPAVIEW_H__FDF284D3_4F00_11D3_B226_0050041CB445__INCLUDED_

#include "CntrItem.h"    // Added by ClassView
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CRPACntrItem;

class CRPAView : public CScrollView
{
protected: // create from serialization only
    CRPAView();
    DECLARE_DYNCREATE(CRPAView)

// Attributes
public:
    CRPADoc* GetDocument();
    // m_pSelection holds the selection to the current CRPACntrItem.
    // For many applications, such a member variable isn't adequate to
    //  represent a selection, such as a multiple selection or a selection
    //  of objects that are not CRPACntrItem objects.  This selection
    //  mechanism is provided just to help you get started.

    // TODO: replace this selection mechanism with one appropriate to your app.
    CRPACntrItem* m_pSelection;
//    CRPACntrItem* pItem;

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CRPAView)
    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual void OnInitialUpdate(); // called first time after construct
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnPrint(CDC* pDC, CPrintInfo* pInfo);
    virtual BOOL IsSelected(const CObject* pDocItem) const;// Container support
    //}}AFX_VIRTUAL

// Implementation
public:
    CRPACntrItem m_graphItem;
    void RunSigmaPlot();
    virtual ~CRPAView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CRPAView)
```

```
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    afx_msg void OnDestroy();
    afx_msg void OnSetFocus(CWnd* pOldWnd);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnInsertObject();
    afx_msg void OnCancelEditCntr();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in RPAView.cpp
inline CRPADoc* CRPAView::GetDocument()
   { return (CRPADoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
// the previous line.

#endif // !defined(AFX_RPAVIEW_H__FDF284D3_4F00_11D3_B226_0050041CB445__INCLUDED_)
```

```
// RPAView.cpp : implementation of the CRPAView class
//

#include "stdafx.h"
#include "RPA.h"

#include "RPADoc.h"
#include "CntrItem.h"
#include "RPAView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

extern SigmaPlot::IJAutoApplicationPtr pSP;
//extern IJAutoApplication SP;
extern SigmaPlot::IJAutoNotebookPtr pNb;
//extern IJAutoNotebook Nb;

/////////////////////////////////////////////////////////////////////////////
// CRPAView

IMPLEMENT_DYNCREATE(CRPAView, CScrollView)

BEGIN_MESSAGE_MAP(CRPAView, CScrollView)
    //{{AFX_MSG_MAP(CRPAView)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    ON_WM_DESTROY()
    ON_WM_SETFOCUS()
    ON_WM_SIZE()
    ON_COMMAND(ID_OLE_INSERT_NEW, OnInsertObject)
    ON_COMMAND(ID_CANCEL_EDIT_CNTR, OnCancelEditCntr)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CRPAView construction/destruction
CRPAView::CRPAView() //: pItem(NULL)
{
    m_pSelection = NULL;
    // TODO: add construction code here
}

CRPAView::~CRPAView()
{
//    if( pItem != NULL )
//        delete pItem;
}

BOOL CRPAView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs

    return CScrollView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CRPAView drawing

void CRPAView::OnDraw(CDC* pDC)
```

```
{
    CRPADoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here

    // Check to see if we have profile data:
    if(pDoc->m_pProfileData == NULL)
    {
        pDC->TextOut(5, 5, "No Data to Plot");
//        pSP->Visible = VARIANT_FALSE;
        return;
    }

    // We have data, so plot it:
    // AARGH
    // DEBUG

    // Display the plot:

    if(!pDoc->plotCurrent)
        RunSigmaPlot();

    //pDoc->SendProfileToSigmaPlot();
    //pSP->Visible = VARIANT_TRUE;


    // TODO: remove this code when final draw code is complete.
    if (m_pSelection == NULL)
    {
        POSITION pos = pDoc->GetStartPosition();
        m_pSelection = (CRPACntrItem*)pDoc->GetNextClientItem(pos);
    }

    if (m_pSelection != NULL)
    {
        CRect rect;
        GetClientRect(rect);
        m_pSelection->Draw(pDC, rect);
    }
}

void CRPAView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();


    // TODO: remove this code when final selection model code is written
    m_pSelection = NULL;    // initialize selection

    //Active documents should always be activated
    COleDocument* pDoc = (COleDocument*) GetDocument();
    if (pDoc != NULL)
    {
        // activate the first one
        POSITION posItem = pDoc->GetStartPosition();
        if (posItem != NULL)
        {
            CDocItem* pItem = pDoc->GetNextItem(posItem);

            // only if it's an Active document
            COleDocObjectItem *pDocObjectItem =
                DYNAMIC_DOWNCAST(COleDocObjectItem, pItem);

            if (pDocObjectItem != NULL)
            {
                pDocObjectItem->DoVerb(OLEIVERB_SHOW, this);
            }
        }
    }
    CSize sizeTotal;
    // TODO: calculate the total size of this view
    sizeTotal.cx = sizeTotal.cy = 100;
    SetScrollSizes(MM_TEXT, sizeTotal);
}

/////////////////////////////////////////////////////////////////////////////
// CRPAView printing

BOOL CRPAView::OnPreparePrinting(CPrintInfo* pInfo)
{
    if (!CView::DoPreparePrinting(pInfo))
        return FALSE;

    if (!COleDocObjectItem::OnPreparePrinting(this, pInfo))
        return FALSE;

    return TRUE;
}

void CRPAView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CRPAView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

void CRPAView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: add customized printing code here
    if(pInfo->m_bDocObject)
        COleDocObjectItem::OnPrint(this, pInfo, TRUE);
    else
        CView::OnPrint(pDC, pInfo);
}

void CRPAView::OnDestroy()
{
    // Deactivate the item on destruction; this is important
    // when a splitter view is being used.
   CScrollView::OnDestroy();
   COleClientItem* pActiveItem = GetDocument()->GetInPlaceActiveItem(this);
   if (pActiveItem != NULL && pActiveItem->GetActiveView() == this)
   {
      pActiveItem->Deactivate();
      ASSERT(GetDocument()->GetInPlaceActiveItem(this) == NULL);
   }
}


/////////////////////////////////////////////////////////////////////////////
// OLE Client support and commands

BOOL CRPAView::IsSelected(const CObject* pDocItem) const
{
    // The implementation below is adequate if your selection consists of
    //  only CRPACntrItem objects.  To handle different selection
    //  mechanisms, the implementation here should be replaced.

    // TODO: implement this function that tests for a selected OLE client item

    return pDocItem == m_pSelection;
}
```

```
void CRPAView::OnInsertObject()
{
    // Invoke the standard Insert Object dialog box to obtain information
    //  for new CRPACntrItem object.
    COleInsertDialog dlg;
    if (dlg.DoModal(/*COleInsertDialog::DocObjectsOnly*/) != IDOK)
        return;

    BeginWaitCursor();

    CRPACntrItem* pItem = NULL;
    TRY
    {
        // Create new item connected to this document.
        CRPADoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pItem = new CRPACntrItem(pDoc);
        ASSERT_VALID(pItem);

        // Initialize the item from the dialog data.
        if (!dlg.CreateItem(pItem))
            AfxThrowMemoryException();  // any exception will do
        ASSERT_VALID(pItem);

        if (dlg.GetSelectionType() == COleInsertDialog::createNewItem)
            pItem->DoVerb(OLEIVERB_SHOW, this);

        ASSERT_VALID(pItem);

        // As an arbitrary user interface design, this sets the selection
        //  to the last item inserted.

        // TODO: reimplement selection as appropriate for your application

        m_pSelection = pItem;   // set selection to last inserted item
        pDoc->UpdateAllViews(NULL);
    }
    CATCH(CException, e)
    {
        if (pItem != NULL)
        {
            ASSERT_VALID(pItem);
            pItem->Delete();
        }
        AfxMessageBox(IDP_FAILED_TO_CREATE);
    }
    END_CATCH

    EndWaitCursor();
}

// The following command handler provides the standard keyboard
//  user interface to cancel an in-place editing session.  Here,
//  the container (not the server) causes the deactivation.
void CRPAView::OnCancelEditCntr()
{
    // Close any in-place active item on this view.
    COleClientItem* pActiveItem = GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
    {
        pActiveItem->Close();
    }
    ASSERT(GetDocument()->GetInPlaceActiveItem(this) == NULL);
}

// Special handling of OnSetFocus and OnSize are required for a container
//  when an object is being edited in-place.
void CRPAView::OnSetFocus(CWnd* pOldWnd)
```

```
{
    COleClientItem* pActiveItem = GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL &&
        pActiveItem->GetItemState() == COleClientItem::activeUIState)
    {
        // need to set focus to this item if it is in the same view
        CWnd* pWnd = pActiveItem->GetInPlaceWindow();
        if (pWnd != NULL)
        {
            pWnd->SetFocus();   // don't call the base class
            return;
        }
    }

    CScrollView::OnSetFocus(pOldWnd);
}

void CRPAView::OnSize(UINT nType, int cx, int cy)
{
    CScrollView::OnSize(nType, cx, cy);
    COleClientItem* pActiveItem = GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
        pActiveItem->SetItemRects();
}

/////////////////////////////////////////////////////////////////////////////
// CRPAView diagnostics

#ifdef _DEBUG
void CRPAView::AssertValid() const
{
    CScrollView::AssertValid();
}

void CRPAView::Dump(CDumpContext& dc) const
{
    CScrollView::Dump(dc);
}

CRPADoc* CRPAView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CRPADoc)));
    return (CRPADoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CRPAView message handlers

void CRPAView::RunSigmaPlot()
{
    CRPACntrItem* pItem = NULL;

    //Change the cursor so the user knows something exciting is going
    //on.
    BeginWaitCursor();

    TRY
    {
        //Get the document associated with this view, and be sure it's
        //valid.
        CRPADoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        //Create a new item associated with this document, and be sure
        //it's valid.
        pItem = new CRPACntrItem(pDoc);
        ASSERT_VALID(pItem);
```

```
// Get Class ID for SigmaPlot notebook.
// This is used in creation.
CLSID clsid;
if(FAILED(::CLSIDFromProgID(L"SigmaPlot.JNB.Page.3",&clsid)))
    //Any exception will do. We just need to break out of the
    //TRY statement.
    AfxThrowMemoryException();

// Create the SigmaPlot embedded item.
if(!pItem->CreateNewItem(clsid))
    //Any exception will do. We just need to break out of the
    //TRY statement.
    AfxThrowMemoryException();

// A nasty hack because reference counts are not being handled properly:
m_unlockHack++;

//Make sure the new CContainerItem is valid.
ASSERT_VALID(pItem);

// Launch the server to edit the item.
//pItem->DoVerb(OLEIVERB_SHOW, this);

// As an arbitrary user interface design, this sets the
// selection to the last item inserted.
m_pSelection = pItem;    // set selection to last inserted item
pDoc->UpdateAllViews(NULL);

//Query for the dispatch pointer for the embedded object. In
//this case, this is the SigmaPlot notebook.
LPDISPATCH lpDisp;
lpDisp = pItem->GetIDispatch();

using namespace SigmaPlot;

// Get pointers to SigmaPlot objects:
pNb = IJAutoNotebookPtr(lpDisp);

IJAutoGraphItemPtr pGraphItem(pNb->CurrentPageItem);

pDoc->SendProfileToSigmaPlot();

SAFEARRAY FAR* pArr;
SAFEARRAYBOUND pBound[1];
pBound[0].lLbound = 0;
pBound[0].cElements = pDoc->m_pProfileData->GetNumberOfChannels() + 1;
pArr = SafeArrayCreate(VT_VARIANT, 1, pBound);
long ai[1];
VARIANT val;
val.vt = VT_I4;

for(long i = 0; i < pDoc->m_pProfileData->GetNumberOfChannels() + 1; i++)
{
    ai[0] = i;
    val.lVal = i;
    SafeArrayPutElement(pArr, ai, &val);
}
VARIANT ColumnArray;
ColumnArray.vt = VT_ARRAY | VT_VARIANT;
ColumnArray.parray = pArr;

IJAutoNativeWorksheetItemPtr pWS(pNb->CurrentDataItem);
IJAutoDataTablePtr pDT(pWS->DataTable);

pGraphItem->CreateWizardGraph("Scatter Plot", "Multiple Scatter",
    "X Many Y", ColumnArray);
```

```
        pDoc->plotCurrent = true;
        pDoc->UpdateAllViews(NULL);
    }

    //Here, we need to do clean up if something went wrong.
    CATCH(CException, e)
    {
        if (pItem != NULL)
        {
            ASSERT_VALID(pItem);
            pItem->Delete();
        }
        AfxMessageBox(IDP_FAILED_TO_CREATE);
    }
    END_CATCH

    //Set the cursor back to normal so the user knows exciting stuff
    //is no longer happening
    EndWaitCursor();
}
```

A.4 CRPFile Class

```
// RPFile.h: interface for the CRPFile class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_RPFILE_H__ECB2D7A0_44FB_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_RPFILE_H__ECB2D7A0_44FB_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "ProfileData.h"

// The CRPfile class handles the RP090L program profile files.  It reads and
// parses data from those files and stores it in a member variable.  The class
// can also assign the profile data to a CIRIProfile object using
// GetProfileData().

class CRPFile
{
public:
    void GetProfileData(CProfileData& profileData);
    CRPFile();
    virtual ~CRPFile();
    bool Import();

private:
    CProfileData m_profileData;
    enum DataType
    {
        UNKNOWN, PROFILE_RATE_DIST
    };
    enum DataType Classify(CString label);
};

#endif // !defined(AFX_RPFILE_H__ECB2D7A0_44FB_11D3_B226_0050041CB445__INCLUDED_)
```

```
// RPFile.cpp: implementation of the CRPFile class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "RPFile.h"
#include "Length.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CRPFile::CRPFile()
{
    return;
}

CRPFile::~CRPFile()
{
    return;
}

bool CRPFile::Import()
{
    // Create an open file dialog box:
    static char szFilter[] = "Profile Files (*.p\?\?)|*.p??|All Files (*.*)|*.*||";
    CFileDialog dlg(true, "txt", NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
        szFilter);

    // Exit on anything but ok:
    if(dlg.DoModal() != IDOK)
        return false;

    dlg.BeginWaitCursor();

    // Create a temporary file in temp directory:
    char cmdline[256];
    char tmpdir[_MAX_DIR];
    char tmpfile[_MAX_PATH];
    GetTempPath(_MAX_PATH, tmpdir);
    GetTempFileName(tmpdir, "~rp", 0, tmpfile);

    // Run RP090L hidden and wait for it to finish:
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    memset(&si, 0, sizeof(si));
       si.cb = sizeof(STARTUPINFO);
    si.dwFlags = STARTF_USESHOWWINDOW;
    si.wShowWindow = SW_HIDE;
    sprintf(cmdline, "C:\\TEMP\\rp090l.exe /if %s /profile /of %s /ext",
        dlg.GetPathName(), tmpfile); // DEBUG
    if(!CreateProcess(NULL, cmdline, NULL, NULL, FALSE, NORMAL_PRIORITY_CLASS,
        NULL, NULL, &si, &pi))
    {
        // Give a message on error:
        CString mesg;
        mesg.Format("Error: Could not open RP090L.EXE.\nError code %d",
            GetLastError());
        AfxMessageBox(mesg);
        return false;
    }
    WaitForSingleObject(pi.hProcess, INFINITE);

    // Open the temp file:
    CStdioFile rpfile(tmpfile, CFile::modeRead | CFile::shareDenyNone |
        CFile::typeText);

    // Parse the file one line at a time:
    CString buffer;
    do
    {
        CString label, data;
        if(!rpfile.ReadString(buffer))
            // Error:  No Data
            throw;        // DEBUG

        buffer.TrimRight();
        label = buffer.Left(20);
        label.TrimRight();
        if(buffer.GetLength() > 20)
            data = buffer.Mid(20);
        else
            data.Empty();
        data.TrimRight();

        switch(Classify(label))
        {
        case PROFILE_RATE_DIST:
            {
                double sampleInterval;
                data.Delete(data.GetLength() - 1, 1);
                sscanf(data, "%lf", &sampleInterval);
                m_profileData.SetSampleInterval(CLength(sampleInterval,
                    CLength::ft));
            }
            break;
        default:
            break;
        }
    } while(buffer.Left(1) != '-');

    // Read profile values:
    m_profileData.SetNumberOfChannels(2);
    int index = 0;
    while(rpfile.ReadString(buffer))
    {
        double left = 0, right = 0, refpost = 0;
        int dmi = 0;

        sscanf(buffer, "%lf %d %lf %lf", &refpost, &dmi, &left, &right);
        m_profileData.SetSamplePoint(0, index, CLength(left, CLength::ft));
        m_profileData.SetSamplePoint(1, index, CLength(right, CLength::ft));

        index++;
    }
    m_profileData.SetNumberOfSamples(index);
    rpfile.Close();
    dlg.EndWaitCursor();

    // Delete temporary file:
    CFile::Remove(rpfile.GetFilePath());

    return true;
}

enum CRPFile::DataType CRPFile::Classify(CString label)
{
// Classify function:  Determines the type of value read from the input
//                     file.
```

```
    // Only looking for sample interval right now:
    if(label.CompareNoCase("PROFILE_RATE_DIST") == 0)
        return PROFILE_RATE_DIST;

    return UNKNOWN;
}

void CRPFile::GetProfileData(CProfileData &profileData)
{
    // Assign member profile data to the specified object:
    profileData = m_profileData;
}
```

A.5 CMatrix Class

```
//////////////////////////////////////////////////////////////////////
// Matrix.h: interface for the CMatrix class.                      //
//////////////////////////////////////////////////////////////////////
//                                                                  //
// Class CMatrix defines a dynamically-allocated matrix object and  //
// several operations that can be performed on matrix objects.      //
//                                                                  //
//                                                                  //
// CMatrix objects can be declared in the following ways:           //
//                                                                  //
//    CMatrix matrix(r, c);                                         //
//       This declares a matrix with r rows and c columns.          //
//                                                                  //
//    CMatrix matrix;                                               //
//       This declares an empty matrix, 0 rows and 0 columns.       //
//       It needs to be redimensioned using the resize() function   //
//       to be made useful.  This is mainly needed for declaring    //
//       of CMatrix objects, where constructor arguments cannot be  //
//       specified.                                                 //
//                                                                  //
//                                                                  //
// Given a CMatrix object matrix, the following operations are      //
// available:                                                       //
//                                                                  //
//    matrix.resize(r, c);                                          //
//       This destroys the current matrix and re-creates it with r  //
//       rows and c columns.  This is primarily used on empty       //
//       matrices created with the default constructor, as noted    //
//       above.                                                     //
//                                                                  //
//    matrix[i][j]:                                                 //
//       This expression allows read and write access to the matrix //
//       element at row i and column j.  Be careful not to access   //
//       rows or columns beyond the dimensions of the matrix, just  //
//       as one would be careful not to access elements past the    //
//       of a C-style array.                                        //
//                                                                  //
//    operators +, -, *, /, =, +=, -=, *=, /=:                      //
//       These operators work just as one would expect their        //
//       built-in C counterparts to function.  Make sure that the   //
//       dimensions of the matrices involved are appropriate for    //
//       particular operation.                                      //
//                                                                  //
//    conditional operators == and !=:                              //
//       These operators test for equality/inequality up to a given //
//       tolerance (since we cannot directly test real numbers for  //
//       equality).  The tolerance is specified below in the        //
//       definition of EPSILON.                                     //
//                                                                  //
//    matrix.cofactor(r, c);                                        //
//       Returns the cofactor associated with row r and column c.   //
//                                                                  //
//    matrix.determinant();                                         //
//       Returns the determinant of the matrix.                     //
//                                                                  //
//    matrix.GetMaxAbsElement();                                    //
//       Returns the absolute value of the matrix element with the  //
//       largest absolute value.  Used in the == function to test   //
//       matrices for equality.                                     //
//                                                                  //
//    matrix.identity();                                            //
//       Returns an identity matrix with the same dimensions as     //
//       matrix.  Make sure the original matrix is square.          //
//                                                                  //
//    matrix.inverse();                                             //
//       Returns the inverse of matrix.  Make sure that matrix is   //
//       square and not singular (its determinant is non-zero).     //
//                                                                  //
//    matrix.minor(r, c);                                           //
//       Returns the minor associated with row r and column c.      //
//                                                                  //
//    matrix.transposed();                                          //
//       Returns a matrix that is the transposed version of matrix. //
//                                                                  //
//    exp(matrix);                                                  //
//       Returns e^matrix.  Make sure that the matrix is square.    //
//                                                                  //
//    inv(matrix);                                                  //
//       Returns the inverse matrix.  Identical to                  //
//       matrix.inverse(), provided for convenience.                //
//                                                                  //
//    pow(matrix, n);                                               //
//       Returns matrix^n.  Make sure that the matrix is square.    //
//                                                                  //
//    factorial(n);                                                 //
//       Returns n!.  Used in the exp() function.                   //
//                                                                  //
//////////////////////////////////////////////////////////////////////


#if !defined(AFX_MATRIX_H__16AAFD45_255C_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_MATRIX_H__16AAFD45_255C_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


//////////////////////////////////////////////////////////////////////
// Definitions of types:                                            //
//////////////////////////////////////////////////////////////////////

// Type of data held in matrix elements:
typedef double MATRIXDATA;


// Type of variable used as matrix index.
typedef unsigned int MATRIXINDEX;


// Type of variable used as exponent:
typedef __int64 EXPONENT;


//////////////////////////////////////////////////////////////////////
// Definitions of constants:                                        //
//////////////////////////////////////////////////////////////////////

// Maximum number of iterations for calculating matrix exponential.
//    Choose such that (MAXITERATIONS)! <= max value of EXPONENT:
#define MAXITERATIONS 20


// Tolerance for comparing floating points for equality:
#define EPSILON DBL_EPSILON


// Special factor used in calculating matrixexponential:
#define EXPREDUCTIONFACTOR 64


//////////////////////////////////////////////////////////////////////
// Declaration of CMatrix class:                                    //
//////////////////////////////////////////////////////////////////////
```

```
class CMatrix
{
private:
// Attributes:
    MATRIXINDEX row;
    MATRIXINDEX col;
    MATRIXDATA* data;

public:
// Construction/Destruction:
    CMatrix();
    CMatrix(MATRIXINDEX r, MATRIXINDEX c);
    CMatrix(const CMatrix& matrix);
    void resize(MATRIXINDEX r, MATRIXINDEX c);
    virtual ~CMatrix();

// Array operator:
    MATRIXDATA* operator[](MATRIXINDEX i) const;

// Assignment operators:
    CMatrix& operator=(const CMatrix& mat);
    CMatrix& operator+=(const CMatrix& mat);
    CMatrix& operator-=(const CMatrix& mat);
    CMatrix& operator*=(const CMatrix& matrix);
    CMatrix& operator*=(const MATRIXDATA scalar);
    CMatrix& operator/=(const MATRIXDATA scalar);

// Conditional operators:
    bool CMatrix::operator ==(const CMatrix& mat2) const;
    bool CMatrix::operator !=(const CMatrix& mat2) const;

// Arithmetic operators:
    CMatrix operator+(const CMatrix& mat) const;
    CMatrix operator-(const CMatrix& mat) const;
    CMatrix operator*(const CMatrix& mat) const;
    CMatrix operator/(const MATRIXDATA scalar) const;
    CMatrix operator/(const EXPONENT scalar) const;

// Member functions:
    MATRIXDATA cofactor(MATRIXINDEX r, MATRIXINDEX c) const;
    MATRIXDATA determinant() const;
    MATRIXDATA GetMaxAbsElement() const;
    CMatrix identity() const;
    CMatrix inverse() const;
    MATRIXDATA minor(MATRIXINDEX r, MATRIXINDEX c) const;
    CMatrix transposed() const;

private:
// Friend operators:
    friend CMatrix operator*(const CMatrix& matrix, MATRIXDATA scalar);
    friend CMatrix operator*(MATRIXDATA scalar, const CMatrix& matrix);

// Friend functions:
    friend CMatrix exp(const CMatrix& matrix);
    friend CMatrix inv(const CMatrix& matrix);
    friend CMatrix pow(const CMatrix&, EXPONENT pow);
};


//////////////////////////////////////////////////////////////////////
// External operators:                                              //
//////////////////////////////////////////////////////////////////////

CMatrix operator*(const CMatrix& matrix, MATRIXDATA scalar);
CMatrix operator*(MATRIXDATA scalar, const CMatrix& matrix);
```

```
//////////////////////////////////////////////////////////////////////
// External functions:                                              //
//////////////////////////////////////////////////////////////////////

CMatrix pow(const CMatrix& matrix, EXPONENT pow);
CMatrix exp(const CMatrix& matrix);
EXPONENT factorial(EXPONENT argument);
CMatrix inv(const CMatrix& matrix);


//////////////////////////////////////////////////////////////////////
// Inline functions:                                                //
//////////////////////////////////////////////////////////////////////

// Must be defined in header file to be inlined.
inline MATRIXDATA* CMatrix::operator [](MATRIXINDEX i) const
{
// Array operator: Allows matrix[i][j] to allow direct access to
//                 elements.

    // Make sure we aren't out-of-bounds:
    ASSERT(i <= row);  // NOTE:  We cannot check the column index!!!!

    // Return a pointer to the beginning of the desired row:
    return &data[i*col];
}

#endif // !defined(AFX_MATRIX_H__16AAFD45_255C_11D3_B226_0050041CB445__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// End of Matrix.h                                                  //
//////////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////////////////////////
// Matrix.cpp: implementation of the CMatrix class.                 //
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Matrix.h"
#include <math.h>   // For fabs() and pow() functions
#include <float.h>  // For definition of DBL_EPSILON used in header


//////////////////////////////////////////////////////////////////////
// Construction/Destruction                                         //
//////////////////////////////////////////////////////////////////////

CMatrix::CMatrix()
{
// Default constructor:  Constructs a CMatrix object with 0 rows and
//                       0 columns.  Can be made useful by calling
//                       resize(r, c).

    // Assign 0 to row and col, null pointer to data:
    row = 0;
    col = 0;
    data = 0;

    // Return:
    return;
}

CMatrix::CMatrix(MATRIXINDEX r, MATRIXINDEX c)
{
// Constructor:  Constructs a CMatrix object with r rows and c
//               columns.  Initializes elements to zero.

    // If we get invalid or zero dimensions, set row and col to zero,
    //    set data to null pointer, and quit:
    if(r <= 0 || c <= 0)
    {
        row = 0;
        col = 0;
        data = 0;
        return;
    }

    // Assign row and col attributes:
    row = r;
    col = c;

    // Allocate space for the matrix elements, throw exception if we
    //    fail:
    if( !(data = new MATRIXDATA[row*col]) )
        throw("Out of Memory");

    // Initialize all elements to zero (several member functions rely
    //    on this):
    for(MATRIXINDEX i = 0; i < row; i++)
        for(MATRIXINDEX j = 0; j < col; j++)
            (*this)[i][j] = 0;

    // Return:
    return;
}

CMatrix::CMatrix(const CMatrix& matrix)
{
// Copy constructor:  Called when CMatrix objects are duplicated.

    // Copy row and column attributes:
```

```
    row = matrix.row;
    col = matrix.col;

    // Allocate space for copy matrix elements, throw exception if we
    //    fail:
    if( !(data = new MATRIXDATA[row*col]) )
        throw("Out of Memory");

    // Copy elements one by one:
    for(MATRIXINDEX i = 0; i < row; i++)
        for(MATRIXINDEX j = 0; j < col; j++)
            (*this)[i][j] = matrix[i][j];
    // Return:
    return;
}

void CMatrix::resize(MATRIXINDEX r, MATRIXINDEX c)
{
// resize Reconstructor:  Destroys the matrix and recreates it with
//                        dimensions r and c.  Initializes elements to
//                        zero.

    // Destroy the current data:
    this->~CMatrix();

    // Reconstruct with new values:
    this->CMatrix::CMatrix(r, c);

    // Return:
    return;
}

CMatrix::~CMatrix()
{
// Destructor:  cleans up CMatrix object when it is no longer needed.

    // Free the space we allocated for the matrix elements:
    delete [] data;

    // Return:
    return;
}


//////////////////////////////////////////////////////////////////////
// Assignment Operators                                             //
//////////////////////////////////////////////////////////////////////

CMatrix& CMatrix::operator =(const CMatrix& mat)
{
// Assignment operator:  Assigns the value of one CMatrix to another.

    // Check that matrices have the same dimensions:
    ASSERT(this->row == mat.row && this->col == mat.col);

    // Assign elements one at a time:
    for(MATRIXINDEX i = 0; i < row; i++)
        for(MATRIXINDEX j = 0; j < col; j++)
            (*this)[i][j] = mat[i][j];

    // Return reference to this:
    return *this;
}

CMatrix& CMatrix::operator+=(const CMatrix& mat)
{
// Augmented addition operator:  Adds the two matrices and assigns the
//                               result to the first.
```

```
    // Check that matrices have the same dimensions:
    ASSERT(this->row == mat.row && this->col == mat.col);

    // Add and assign:
    *this = *this + mat;

    // Return reference to this:
    return *this;
}

CMatrix& CMatrix::operator-=(const CMatrix& mat)
{
// Augmented subtraction operator:  Subtracts the two matrices and
//                                  assigns the result to the first.

    // Check that matrices have the same dimensions:
    ASSERT(this->row == mat.row && this->col == mat.col);

    // Subtract and assign:
    *this = *this - mat;

    // Return reference to this:
    return *this;
}

CMatrix& CMatrix::operator *=(const CMatrix& matrix)
{
// Augmented multiplication operator:  Multiplies the two matrices and
//                                     assigns the result to the
//                                     first.

    // Check that matrices have appropriate dimensions:
    ASSERT(this->col == matrix.row);

    // Multiply and assign:
    *this = *this * matrix;

    // Return reference to this:
    return *this;
}

CMatrix& CMatrix::operator *=(const MATRIXDATA scalar)
{
// Augmented multiplication operator:  Multiplies the matrix by a
//                                     scalar and assigns the result
//                                     to the matrix.

    // Multiply and assign:
    *this = *this * scalar;

    // Return reference to this:
    return *this;
}

CMatrix& CMatrix::operator /=(const MATRIXDATA scalar)
{
// Augmented division operator:  Divides the matrix by a scalar and
//                               assigns the result to the matrix.

    // Divide and assign:
    *this = *this / scalar;

    // Return reference to this:
    return *this;
}

/////////////////////////////////////////////////////////////////////
```

```
// Conditional Operators                                            //
//////////////////////////////////////////////////////////////////////

bool CMatrix::operator ==(const CMatrix& mat2) const
{
// == operator:  Returns true if matrices are equal within the
//               defined tolerance EPSILON (in Matrix.h).

    // First check the dimensions:
    if( !(this->row == mat2.row && this->col == mat2.col) )
        return false;

    // Since we cannot directly compare floating-point numbers for
    //    equality, we subtract the matrices and get the largest
    //    element of the result:
    MATRIXDATA difference = (*this - mat2).GetMaxAbsElement();

    // Compare the difference to the tolerance we have defined and
    //    return the result:
    return fabs(difference) < EPSILON;
}

bool CMatrix::operator !=(const CMatrix& mat2) const
{
// != operator:  Returns true if matrices are not equal within the
//               defined tolerance EPSILON (in Matrix.h).

    // Return the opposite of the == operator:
    return !(*this == mat2);
}


//////////////////////////////////////////////////////////////////////
// Arithmetic Operators                                             //
//////////////////////////////////////////////////////////////////////

CMatrix CMatrix::operator +(const CMatrix& mat) const
{
// Addition operator:  Returns the sum of two matrices.

    // Check that matrices have the same dimensions:
    ASSERT(this->row == mat.row && this->col == mat.col);

    // Create a temporary matrix to hold the result:
    CMatrix result(row, col);

    // Add elements one at a time:
    for(MATRIXINDEX i = 0; i < row; i++)
        for(MATRIXINDEX j = 0; j < col; j++)
            result[i][j] = (*this)[i][j] + mat[i][j];

    // Return the result:
    return result;
}

CMatrix CMatrix::operator -(const CMatrix& mat) const
{
// Subtraction operator:  Returns the difference of two matrices.

    // Check that matrices have the same dimensions:
    ASSERT(this->row == mat.row && this->col == mat.col);

    // Create a temporary matrix to hold the result:
    CMatrix result(row, col);

    // Subtract elements one at a time:
    for(MATRIXINDEX i = 0; i < row; i++)
        for(MATRIXINDEX j = 0; j < col; j++)
```

```
            result[i][j] = (*this)[i][j] - mat[i][j];

    // Return the result:
    return result;
}

CMatrix CMatrix::operator *(const CMatrix& mat) const
{
// Multiplication operator:  Returns the product of two matrices.

    // Check that matrices have appropriate dimensions:
    ASSERT(this->col == mat.row);

    // Create a temporary matrix to hold the result:
    CMatrix result(this->row, mat.col);

    // Multiply, remembering that the constructor initializes result
    //    to all zeroes for us:
    for(MATRIXINDEX i = 0; i < this->row; i++)
        for(MATRIXINDEX j = 0; j < mat.col; j++)
            for(MATRIXINDEX k = 0; k < this->col; k++)
                result[i][j] += (*this)[i][k] * mat[k][j];

    // Return the result:
    return result;
}

CMatrix CMatrix::operator /(const MATRIXDATA scalar) const
{
// Division operator:  Returns the matrix divided by a scalar.

    // Create a temporary matrix to hold the result:
    CMatrix result(this->row, this->col);

    // Divide each element by the scalar:
    for(MATRIXINDEX i=0; i<this->row; i++)
        for(MATRIXINDEX j=0; j<this->col; j++)
            result[i][j] = (*this)[i][j] / scalar;

    // Return the result:
    return result;
}

CMatrix CMatrix::operator /(const EXPONENT scalar) const
{
// Division operator:  Returns the matrix divided by a scalar.

    // Create a temporary matrix to hold the result:
    CMatrix result(this->row, this->col);

    // Divide each element by the scalar:
    for(MATRIXINDEX i=0; i<this->row; i++)
        for(MATRIXINDEX j=0; j<this->col; j++)
            result[i][j] = (*this)[i][j] / scalar;

    // Return the result:
    return result;
}


//////////////////////////////////////////////////////////////////////
// Member functions:                                                //
//////////////////////////////////////////////////////////////////////

MATRIXDATA CMatrix::cofactor(MATRIXINDEX r, MATRIXINDEX c) const
{
// cofactor function:  Returns the cofactor associated with row r and
//                     column c.
```

```
    // Check that the matrix is square:
    ASSERT(this->row == this->col);

    // Return the cofactor, which is (-1)^(r+c) times the minor:
    return pow(-1, r+c) * minor(r, c);
}

MATRIXDATA CMatrix::determinant() const
{
// determinant function:  Returns the determinant of the matrix.

    // Check that the matrix is square:
    ASSERT(this->row == this->col);

    // Create a variable to hold the result:
    MATRIXDATA result = 0;

    // If we are down to a single element matrix, the element is the
    //    determinant:
    if(row == 1 && col == 1)
        result = data[0];
    else
        // Calculate the determinant by cofactors about the first
        //    column (arbitrary, we could have used any row or
        //    column). Determinant = sum of (element * cofactor) over
        //    the column.  Remember that the constructor initializes
        //    result to all zeroes:
        for(MATRIXINDEX i = 0; i < row; i++)
            result += (*this)[i][0] * cofactor(i, 0);

    // Return the result;
    return result;
}

MATRIXDATA CMatrix::GetMaxAbsElement() const
{
// GetMaxAbsElement function:  Returns the absolute value of the
//                             matrix element with the largest
//                             absolute value.

    // Create a variable to hold the result:
    MATRIXDATA result = 0;

    // Check each element and keep the largest absolute value:
    for(MATRIXINDEX i=0; i<this->row; i++)
        for(MATRIXINDEX j=0; j<this->col; j++)
            if(fabs((*this)[i][j]) > result)
                result = fabs((*this)[i][j]);

    // Return the result:
    return result;
}

CMatrix CMatrix::identity() const
{
// identity function:  Returns an identity matrix of the same size as
//                     the current matrix.

    // Check that the matrix is square:
    ASSERT(this->row == this->col);

    // Create a temporary matrix to hold the result:
    CMatrix result(this->row, this->col);

    // Assign the identity matrix values to elements of result,
    //    remembering that the constructor initializes result to all
    //    zeroes:
```

```
    for(MATRIXINDEX i=0; i<this->row; i++)
        result[i][i] = 1;
    return result;
}

CMatrix CMatrix::inverse() const
{
// inverse function:  Returns the inverse of the matrix.

    // Check that the matrix is square:
    ASSERT(this->row == this->col);

    // Calculate the inverse matrix, using the formula:
    //    M^-1 = C^T / det(M)
    //    where C is a matrix whose elements are the corresponding
    //    cofactors of M, ^T represents transposition, and det(M)
    //    represents the determinant of M.

    // Create the cofactor matrix:
    CMatrix cofactorMatrix(row, col);
    for(MATRIXINDEX i = 0; i < row; i++)
        for(MATRIXINDEX j=0; j < col; j++)
            cofactorMatrix[i][j] = cofactor(i,j);

    // Return the inverse matrix, which is the cofactor matrix
    //    transposed over the original determinant:
    return cofactorMatrix.transposed() / this->determinant();
}

MATRIXDATA CMatrix::minor(MATRIXINDEX r, MATRIXINDEX c) const
{
// minor function:  Returns the minor associated with row r and
//                  column c.

    // Check that matrix is square:
    ASSERT(this->row == this->col);

    // Create a variable to hold the result:
    MATRIXDATA result;

    // Create a temporary matrix one row and column smaller than the
    //    original:
    CMatrix tmpmatrix(this->row-1, this->col-1);

    // Fill tmpmatrix with original matrix, minus row r and column c:
    for(MATRIXINDEX i = 0; i < tmpmatrix.row; i++)
        for(MATRIXINDEX j = 0; j < tmpmatrix.col; j++)
            // There are 4 possible cases:
            if(i < r && j < c)
                // 1 - Haven't reached row r or column c yet, so just
                //     copy elements
                tmpmatrix[i][j] = (*this)[i][j];

            else if(i < r && j >= c)
                // 2 - At or past column c, but haven't reached row r.
                //     Skip one column.
                tmpmatrix[i][j] = (*this)[i][j+1];

            else if(i >= r && j < c)
                // 3 - At or past row r, but haven't reached column c.
                //     Skip one row.
                tmpmatrix[i][j] = (*this)[i+1][j];

            else // (i > r && j > c)
                // 4 - Past row r and column c. Skip one row and one
                //     column.
                tmpmatrix[i][j] = (*this)[i+1][j+1];
    // The minor is the determinant of our temporary matrix:
    result = tmpmatrix.determinant();

    // Return the result:
    return result;
}

CMatrix CMatrix::transposed() const
{
//  transposed function:  Returns the matrix transposed (rows and
//                        columns interchanged).

    // Create a temporary matrix to hold the result:
    CMatrix result(col, row);

    // Fill result with the transposition of the original matrix:
    for(MATRIXINDEX i=0; i < this->col; i++)
        for(MATRIXINDEX j=0; j < this->row; j++)
            result[i][j] = (*this)[j][i];

    // Return the result:
    return result;
}

//////////////////////////////////////////////////////////////////////
// External operators:                                              //
//////////////////////////////////////////////////////////////////////

CMatrix operator *(const CMatrix& matrix, const MATRIXDATA scalar)
{
// Multiplication operator:  Returns the product of a matrix and a
//                           scalar.

    // Create a temporary matrix to hold the result:
    CMatrix result(matrix.row, matrix.col);

    // Multiply each element by the scalar:
    for(MATRIXINDEX i=0; i<matrix.row; i++)
        for(MATRIXINDEX j=0; j<matrix.col; j++)
            result[i][j] = matrix[i][j] * scalar;

    // Return the result:
    return result;
}

CMatrix operator *(const MATRIXDATA scalar, const CMatrix& matrix)
{
// Multiplication operator:  Returns the product of a matrix and a
//                           scalar.

    // Create a temporary matrix to hold the result:
    CMatrix result(matrix.row, matrix.col);

    // Multiply each element by the scalar:
    for(MATRIXINDEX i=0; i<matrix.row; i++)
        for(MATRIXINDEX j=0; j<matrix.col; j++)
            result[i][j] = matrix[i][j] * scalar;

    // Return the result:
    return result;
}

//////////////////////////////////////////////////////////////////////
// External Functions                                               //
//////////////////////////////////////////////////////////////////////

CMatrix pow(const CMatrix& matrix, EXPONENT pow)
{
```

```
// pow function:  Returns the value of the matrix to the pow power.

    // Check that the matrix is square:
    ASSERT(matrix.row == matrix.col);

    // Create a temporary matrix to hold the result:
    CMatrix result(matrix.row, matrix.col);

    if(pow >= 0)
    {
        // Start with matrix^0 (identity matrix):
        result = matrix.identity();

        // Multiply by the matrix pow times:
        for(EXPONENT n = 0; n < pow; n++)
            result *= matrix;
    }
    else // (pow < 0)
    {
        // Start with matrix^-1 (inverse matrix):
        result = matrix.inverse();

        // Multiply by matrix^-1 pow - 1 times:
        for(EXPONENT n = -1; n > pow; n--)
            result *= matrix.inverse();
    }

    // Return the result:
    return result;
}

CMatrix exp(const CMatrix& matrix)
{
// exp function:  Returns the value of e^matrix.

    // Check that the matrix is square:
    ASSERT(matrix.row == matrix.col);

    // We calculate [e^(matrix/m)]^m because it converges faster than
    //    e^matrix:
    const EXPONENT m = EXPREDUCTIONFACTOR;
    const CMatrix arg = matrix / m;

    // Calculate using Taylor series:
    //    e^matrix = sum(n=0 to infinity) of matrix^n / n!:

    // Create temporary matrices to hold the result of the current and
    //    last iterations:
    CMatrix result(matrix.row, matrix.col);
    CMatrix lastresult(matrix.row, matrix.col);

    // Start at term 1 (we know that matrix^0 == identity matrix):
    EXPONENT n = 1;
    result = matrix.identity();
    do
    {
        // Save the previous result:
        lastresult = result;

        // Add the next term in the series:
        result += pow(arg, n) / factorial(n);
        n++;

        // Quit when we are accurate to the desired precision or we
        //    have taken too many iterations (n! may overflow if we
        //    do not stop at MAXITERATIONS):
    } while(result != lastresult && n <= MAXITERATIONS);
    // Raise the result to the m power to finish the [e^(matrix/m)]^m
    //    computation:
    result = pow(result, m);

    // Return the result:
    return result;
}

EXPONENT factorial(EXPONENT arg)
{
// factorial function:  Returns argument!, where argument! is equal to
//                      arg * (arg - 1) * (arg - 2) * ... * 1.

    // Make sure arg is non-negative:
    ASSERT(arg >= 0);

    // Catch 0!, which is defined to be 1:
    if(arg == 0)
        return 1;

    // Start at 1:
    EXPONENT result = 1;

    // Take 1 * 2 * 3 * ... * arg:
    for(EXPONENT n = 2; n <= arg; n++)
        result *= n;

    // Return the result:
    return result;
}

CMatrix inv(const CMatrix& matrix)
{
// inv function:  Returns the inverse of the matrix.

    // Check that the matrix is square:
    ASSERT(matrix.row == matrix.col);

    // Return the inverse matrix:
    return matrix.inverse();
}

//////////////////////////////////////////////////////////////////////
// End of Matrix.cpp                                                //
//////////////////////////////////////////////////////////////////////
```

A.6 CProfileData Class

```
// ProfileData.h: interface for the CProfileData class.
//
//////////////////////////////////////////////////////////////////////

// The CProfileData class is the parent class for all profile data stored
// in the Road Profile Analyzer program.  It contains a CProfileArray object,
// which in turn holds a CIRIProfile object for each channel in the data.
// This class holds the data that is universal across channels, such as
// sample interval and number of samples.  It also contains the number of
// channels.  CProfileData is serializable, making it simple to save from
// the CRPADoc class.

#if !defined(AFX_PROFILEDATA_H__C35B0AE2_4F09_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_PROFILEDATA_H__C35B0AE2_4F09_11D3_B226_0050041CB445__INCLUDED_

#include "IRIProfileArray.h"    // Added by ClassView
#include "IRIProfile.h"    // Added by ClassView
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CProfileData : public CObject
{
public:
    CProfileData& operator =(const CProfileData &rProfileData);
    CProfileData(const CProfileData& srcProfileData);
    double GetSampleInterval();
    double GetSamplePoint(int chan, int samp);
    int GetNumberOfSamples();
    double GetEffLength() const;
    double GetChannelIRI(int channel, double& start, double& stop);
    int GetNumberOfChannels();
    void SetProfile(int chan, const CIRIProfile& prof);
    void SetNumberOfChannels(int nChannels);
    virtual void Serialize(CArchive& ar);
    void SetSamplePoint(int channel, int index, double value);
    void SetNumberOfSamples(int nSamples);
    void SetSampleInterval(double interval);
    CProfileData();
    virtual ~CProfileData();

protected:
    DECLARE_SERIAL(CProfileData);

private:
    CIRIProfileArray profile;
    int numChannels, numSamples;
    double sampleInterval;
};

#endif // !defined(AFX_PROFILEDATA_H__C35B0AE2_4F09_11D3_B226_0050041CB445__INCLUDED_)
```

```
// ProfileData.cpp: implementation of the CProfileData class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "ProfileData.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

IMPLEMENT_SERIAL(CProfileData, CObject, 0);

CProfileData::CProfileData() : numChannels(0), numSamples(0), sampleInterval(0)
{
    // We will typically have 2 channels:
    profile.SetSize(2);
}

CProfileData::~CProfileData()
{
}

CProfileData::CProfileData(const CProfileData &srcProfileData)
{
    *this = srcProfileData;
}

void CProfileData::SetSampleInterval(double interval)
{
    sampleInterval = interval;

    for(int i = 0; i < numChannels; i++)
        profile[i].SetDelta(sampleInterval);
}

void CProfileData::SetNumberOfSamples(int nSamples)
{
    numSamples = nSamples;

    for(int i = 0; i < numChannels; i++)
        profile[i].SetNSamp(numSamples);
}

void CProfileData::SetNumberOfChannels(int nChannels)
{
    numChannels = nChannels;

    if(profile.GetSize() != numChannels)
        profile.SetSize(numChannels);

    int i;

    for(i = 0; i < nChannels; i++)
        profile[i].SetNSamp(numSamples);

    for(i = 0; i < numChannels; i++)
        profile[i].SetDelta(sampleInterval);
}

void CProfileData::SetSamplePoint(int channel, int index, double value)
{
    profile[channel].SetProfile(index, value);
}

void CProfileData::Serialize(CArchive& ar)
{
    profile.Serialize(ar);

    if(ar.IsStoring())
    {
        // Storing Code:
        ar << numChannels << numSamples << sampleInterval;
    }
    else
    {
        // Loading Code:
        ar >> numChannels >> numSamples >> sampleInterval;
    }

    CObject::Serialize(ar);
}

void CProfileData::SetProfile(int chan, const CIRIProfile &prof)
{
    profile[chan] = prof;
}

int CProfileData::GetNumberOfChannels()
{
    return numChannels;
}

double CProfileData::GetChannelIRI(int channel, double& start, double& stop)
{
    return profile[channel].IRI(start, stop);
}

double CProfileData::GetEffLength() const
{
    const double L_b = 0.250;
    const int k = MAX(1, ROUND(L_b / sampleInterval));

    return (numSamples - k) * sampleInterval;
}

int CProfileData::GetNumberOfSamples()
{
    return numSamples;
}

double CProfileData::GetSamplePoint(int chan, int samp)
{
    return profile[chan].GetProfile(samp);
}

double CProfileData::GetSampleInterval()
{
    return sampleInterval;
}

CProfileData& CProfileData::operator =(const CProfileData &rProfileData)
{
    numChannels = rProfileData.numChannels;
    numSamples = rProfileData.numSamples;
    profile = rProfileData.profile;
    sampleInterval = rProfileData.sampleInterval;

    return *this;
}
```

A.7 CIRIProfile Class

```
/////////////////////////////////////////////////////////////////////
// IRIProfile.h: interface for the CIRIProfile class.              //
/////////////////////////////////////////////////////////////////////

// The IRIProfile class contains the data describing a profile, including
// sample interval, profile height vs. distance, and number of samples.
// It contains the functions necessary to calculate IRI on the profile.
// All length values are treated as meters, and all slope values are
// treated as m/km.  The CLength and CSlope classes can be used for
// conversion to other units.

#if !defined(AFX_IRIPROFILE_H__43420E69_2E45_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_IRIPROFILE_H__43420E69_2E45_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


/////////////////////////////////////////////////////////////////////
// Required includes:                                              //
/////////////////////////////////////////////////////////////////////

#include "Matrix.h"            // for CMatrix class
#include "DoubleArray.h"
#include "ObjectArray.h"
#include <float.h>            // for DBL_EPSILON constant


/////////////////////////////////////////////////////////////////////
// Definitions of types:                                           //
/////////////////////////////////////////////////////////////////////

// Type of variable used as index for profile and state matrix arrays:
typedef int IRIINDEX;


// Type of variable used for profile height values:
typedef double IRIDATA;


/////////////////////////////////////////////////////////////////////
// Definitions of constants:                                       //
/////////////////////////////////////////////////////////////////////

// Flags that indicate what we need to recalculate:
#define NEED_COMPUTESLOPE 0x01
#define NEED_QUARTERCAR   0x02
#define NEED_IRI          0x04


// Tolerance for comparing real numbers:
#define IRI_EPSILON DBL_EPSILON


/////////////////////////////////////////////////////////////////////
// Declaration of CIRIProfile class:                               //
/////////////////////////////////////////////////////////////////////

class CIRIProfile : public CObject
{
private:
// Attributes:
    IRIDATA delta, length;
    IRIINDEX nSamp, nSampAdj;
    CDoubleArray h_p, s_ps;
    CMatrix A, B, I, M1, M2;
    CObjectArray<CMatrix> x;
    byte flags;

protected:
    DECLARE_SERIAL(CIRIProfile);

public:
    enum Units
    {
        m, km, in, ft, mi
    };

public:
// Construction/Destruction:
    CIRIProfile();
    CIRIProfile(const CIRIProfile& srcIRIProfile);
    virtual ~CIRIProfile();

// Assignment operator:
    CIRIProfile& operator=(const CIRIProfile& rIRIProfile);

// Attribute functions:
    IRIDATA GetLength() const;
    IRIDATA GetAdjLength();
    IRIDATA GetProfile(IRIINDEX index) const;
    void SetDelta(IRIDATA newDelta);
    void SetNSamp(IRIINDEX newNSamp);
    void SetProfile(IRIINDEX index, IRIDATA value);

// Serialization:
    virtual void Serialize(CArchive& ar);

// Public member functions:
    IRIDATA IRI(IRIDATA& start, IRIDATA& stop);

// Private member functions:
private:
    void ComputeSlope();
    void QuarterCar();
};

#endif // !defined(AFX_IRIPROFILE_H__43420E69_2E45_11D3_B226_0050041CB445__INCLUDED_)


//////////////////////////////////////////////////////////////////////
// End of IRIProfile.h                                              //
//////////////////////////////////////////////////////////////////////
```

```
/////////////////////////////////////////////////////////////////////
// IRIProfile.cpp: implementation of the CIRIProfile class.        //
/////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "IRIProfile.h"
//#include <math.h>        // For fabs() function and __max() macro


/////////////////////////////////////////////////////////////////////
// Construction/Destruction                                        //
/////////////////////////////////////////////////////////////////////

IMPLEMENT_SERIAL(CIRIProfile, CObject, 0)

CIRIProfile::CIRIProfile() :
// Constructor:  Sets up member variables for performing IRI
//               computations.

    // Initialize members:
    A(4, 4),    // Set dimensions on matrices
    B(4, 1),
    I(4, 4),
    M1(4, 4),
    M2(4, 1),
    flags(0xff)    // Set all flags because nothing has been computed
{

    // Initialize A and B matrices with "golden car" values:
    // "Golden Car" constants:
    const double c = 6.0;
    const double k1 = 653.0;
    const double k2 = 63.3;
    const double mu = 0.15;

    // Initialize A, B and I matrices (A and B are defined in IRI
    //    algorithm).  (Recall CMatrix constructor initializes
    //    matrices to all zeroes):
    A[0][1] = 1;        //        0      1        0         0
    A[1][0] = -k2;        // A =   -k2    -c       k2         c
    A[1][1] = -c;        //        0      0        0         1
    A[1][2] = k2;        //      k2/mu  c/mu  -(k1+k2)/mu  -c/mu
    A[1][3] = c;
    A[2][3] = 1;
    A[3][0] = k2 / mu;
    A[3][1] = c / mu;
    A[3][2] = -(k1 + k2) / mu;
    A[3][3] = -c / mu;
    B[3][0] = k1 / mu;    //       0
                        // B =   0
                        //       0
                        //     k1/mu
    I = I.identity();    //      1 0 0 0
                        // I =  0 1 0 0
                        //      0 0 1 0
                        //      0 0 0 1

    // Return:
    return;
}

CIRIProfile::CIRIProfile(const CIRIProfile& srcIRIProfile)
{
// Copy Constructor:  Constructs a copy of a CIRIProfile object.

    // Call Default Constructor:
    this->CIRIProfile::CIRIProfile();
    // Copy essential attributes:
    delta = srcIRIProfile.delta;
    h_p.SetSize(srcIRIProfile.h_p.GetSize());
    for(int i = 0; i < h_p.GetSize(); i++)
        h_p[i] = srcIRIProfile.h_p[i];
    nSamp = srcIRIProfile.nSamp;
    flags = 0xff;    // need to recalculate everything
}

CIRIProfile::~CIRIProfile()
{
// Destructor:  Does nothing, since we have no dynamic allocation in
//              the CIRIProfile class.

    // Return:
    return;
}


//////////////////////////////////////////////////////////////////////
// Assignment operator:                                             //
//////////////////////////////////////////////////////////////////////

CIRIProfile& CIRIProfile::operator=(const CIRIProfile& rIRIProfile)
{
// Assignment operator:  Makes this CIRIProfile the same as

    // Copy essential attributes:
    delta = rIRIProfile.delta;
    h_p.SetSize(rIRIProfile.h_p.GetSize());
    for(int i = 0; i < h_p.GetSize(); i++)
    {
        h_p[i] = rIRIProfile.h_p[i];
    }
    nSamp = rIRIProfile.nSamp;
    flags = 0xff;    // need to recalculate everything

    // Return a reference to this:
    return *this;
}


//////////////////////////////////////////////////////////////////////
// Attribute functions:                                             //
//////////////////////////////////////////////////////////////////////

IRIDATA CIRIProfile::GetLength() const
{
    return nSamp * delta;
}

IRIDATA CIRIProfile::GetAdjLength()
{
    if(flags & NEED_COMPUTESLOPE)
        ComputeSlope();
    return nSampAdj * delta;
}

IRIDATA CIRIProfile::GetProfile(IRIINDEX index) const
{
// GetProfile function:  Returns the profile height h_p at the given
//                       index.

    // Return h_p[index]:
    return h_p[index];
}

void CIRIProfile::SetDelta(IRIDATA newDelta)
```

```
{
// SetDelta function:  Sets the delta attribute of the profile and
//                     updates flags so we know which simulations to
//                     run again.

    // Set flags if needed:
    if(ABS(delta - newDelta) > IRI_EPSILON)
        flags |= NEED_COMPUTESLOPE & NEED_QUARTERCAR & NEED_IRI;

    // Set delta attribute:
    delta = newDelta;

    // Return:
    return;
}

void CIRIProfile::SetNSamp(IRIINDEX newNSamp)
{
// SetNSamp function:  Sets the nSamp attribute of the profile and
//                     updates flags so we know which simulations to
//                     run again.

    // Set flags if needed:
    if(newNSamp != nSamp)
        flags |= NEED_COMPUTESLOPE & NEED_QUARTERCAR & NEED_IRI;

    // Set nSamp attribute:
    nSamp = newNSamp;

    // Return:
    return;
}

void CIRIProfile::SetProfile(IRIINDEX index, IRIDATA value)
{
// SetProfile function:  Sets the height h_p of the profile at the
//                       given index and updates flags so we know
//                       which simulations to run again.

    // Set flags:
    flags |= NEED_COMPUTESLOPE & NEED_QUARTERCAR & NEED_IRI;

    // Set profile attribute:
    h_p[index] = value;

    // Return:
    return;
}

//////////////////////////////////////////////////////////////////////
// Serialization:                                                   //
//////////////////////////////////////////////////////////////////////

void CIRIProfile::Serialize(CArchive& ar)
{
    if(ar.IsStoring())
    {
        // Storing Code:
        ar << delta << nSamp;
        h_p.Serialize(ar);
    }
    else
    {
        // Loading Code:
        ar >> delta >> nSamp;
        h_p.Serialize(ar);
        flags = 0xff; // need to recalculate everything
    }
    CObject::Serialize(ar);
}


//////////////////////////////////////////////////////////////////////
// Public member functions:                                         //
//////////////////////////////////////////////////////////////////////

IRIDATA CIRIProfile::IRI(IRIDATA& start, IRIDATA& stop)
{
// IRI function:  Returns the IRI of the profile from position start
//                to position stop.  Stop value 0 indicates the end of
//                the profile.  If not specified, both start and stop
//                default to 0.  Will only return the correct result
//                if nSamp and delta have been set correctly.

    // Create a variable to hold the result:
    IRIDATA result = 0.0;

    // Check whether we need to apply the moving average and/or
    //    quarter-car filters:
    if(flags & NEED_COMPUTESLOPE)
        ComputeSlope();
    if(flags & NEED_QUARTERCAR)
        QuarterCar();

    // Check for start < 0 or start too large, which means to use the
    //    beginning of the profile for starting point:
    if(start < 0 || start > nSampAdj * delta)
        start = 0.0;

    // Check for stop < 0 or stop too large, which means to use the
    //    end of the (useable) profile for stopping point:
    if(stop < 0 || stop > nSampAdj * delta)
        stop = nSampAdj * delta;

    // Accumulate the IRI according to:
    //
    //    IRI = (1/(stop - start)) *  sum(i=start to stop) of
    //          |(x[i])[0][0] - (x[i])[2][0]|:
    for(IRIINDEX i = ROUND(start / delta); i < ROUND(stop / delta); i++)
        result += ABS((x[i])[0][0] - (x[i])[2][0]);
    result /= ROUND(stop / delta) - ROUND(start / delta);

    // Return the result:
    return result;
}

//////////////////////////////////////////////////////////////////////
// Private member functions:                                        //
//////////////////////////////////////////////////////////////////////

void CIRIProfile::ComputeSlope()
{
// ComputeSlope function:  Generates the smoothed slope array s_ps as
//                         defined in the IRI algorithm.

    // Set up constants as defined in the IRI algorithm:
    //    k is the nearest integer to L_b/delta, or 1, whichever
    //    is larger; and L_b is 0.250m:
    const IRIDATA L_b = (IRIDATA)0.250;
    const IRIINDEX k = MAX(1, ROUND(L_b / delta));

    // The process reduces the number of useable samples by k, so
    //    set nSampAdj accordingly:
    nSampAdj = nSamp - k;

    // Calculate s_ps according to:
```

```
    //
    //    s_ps[i] = (h_p[i+k] - h_p[i]) / (k * delta):
    for(IRIINDEX i = 0; i < nSampAdj; i++)
        s_ps[i] = (h_p[i+k] - h_p[i]) / (k * delta);

    // Free any extra elements in the s_ps array:
    s_ps.SetSize(nSampAdj);

    // Unset the NEED_COMPUTESLOPE flag:
    flags &= ~NEED_COMPUTESLOPE;

    // Return:
    return;
}

void CIRIProfile::QuarterCar()
{
// QuarterCar function:  Computes the state variable array x as
//                       defined in the IRI algorithm.

    // Set up constants as defined in the IRI algorithm:
    //    L_0 is 11m, v is 80km/h (converted here to m/s), and iL_0 is
    //    the index of the nearest height measurement to L_0:
    const IRIDATA L_0 = (IRIDATA)11.0;
    const IRIDATA v = (IRIDATA)80000.0 / (IRIDATA)3600.0;
    const IRIINDEX iL_0 = ROUND(L_0 / delta);

    // Redimension x[0] to 4x1, then initialize x to the values at L_0
    //    as defined by IRI:
    x[0].resize(4, 1);
    (x[0])[0][0] = h_p[iL_0] / L_0;
    (x[0])[1][0] = 0;
    (x[0])[2][0] = h_p[iL_0] / L_0;
    (x[0])[3][0] = 0;

    // Redimension each x element to 4x1, then compute its value
    //    according to:
    //
    //    x[i] = e^(A*delta/v)*x[i-1] + A^-1*e^(A*delta/v)*B*s_ps[i]
    //
    //    But first abbreviate the expression to:
    //
    //    x[i] = M1*x[i-1] + M2*s_ps[i]
    //
    //    where M1 = e^(A*delta/v) and M2 = A^-1*(e^(A*delta/v)-I)*B:
    M1 = exp(A * (delta/v));
    M2 = (inv(A) * (exp(A * (delta/v)) - I)) * B;
    for(IRIINDEX i = 1; i < nSampAdj; i++)
    {
        x[i].resize(4, 1);
        x[i] = M1*x[i-1] + M2*s_ps[i];
    }

    // Free any unused items at the end of the list (they will exist
    //    if we have reduced the size of the profile during the life
    //    of the object):
    x.SetSize(i);


    // Unset the NEED_QUARTERCAR flag:
    flags &= ~NEED_QUARTERCAR;

    // Return:
    return;
}

//////////////////////////////////////////////////////////////////////
// End of IRIProfile.cpp                                            //
```

```
//////////////////////////////////////////////////////////////////////
```

A.8 CLength and CSlope Classes

```
// Length.h: interface for the CLength class.
//
//////////////////////////////////////////////////////////////////////

// CLength stores the value and units for a length.  It returns the value
// in units that depend on how it is cast.  If cast as a double or double&
// (in a function call, for example), it will return its value in meters.
// If cast as a CString, it will return a value plus an appropriate label.

#if !defined(AFX_LENGTH_H__5672BA2B_3EAD_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_LENGTH_H__5672BA2B_3EAD_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CLength
{
public:
// Units enumeration:
    enum Units
    {
        m, km, in, ft, mi
    };

private:
// Attributes:
    double value;
    Units units;
    double valueInMeters;
    bool valueIsCurrent;

public:
// Construction/Destruction:
    CLength();
    CLength(const CLength& srcLength);
    CLength(double length, enum Units units = m);
    void SetLength(double newValue, enum Units newUnits);
    virtual ~CLength();

// Attribute Functions:
    double GetValue();
    void SetValue(double newValue);
    enum Units GetUnits() const;
    void SetUnits(enum Units newUnits);
    double GetValueInMeters() const;

// Casting operators:
    operator CString();
    operator double() const;
    operator double&();
    operator int() const;

// Arithmetic operators:
    CLength operator +(const CLength& rLength) const;
    CLength operator /(const double divisor) const;

// Assignment operators:
    CLength& operator =(const CLength& rLength);
    CLength& operator +=(const CLength& rLength);

// Comparison operators:
    bool operator >(const CLength& rLength) const;
    bool operator <(const CLength& rLength) const;

private:
```

```
// Private member functions:
    double ConvertToMeters(double length,
                           enum Units currentUnits) const;
    double ConvertFromMeters(double length,
                             enum Units newUnits) const;

// Friend classes:
    friend class CIRIDialog;
};

#endif // !defined(AFX_LENGTH_H__5672BA2B_3EAD_11D3_B226_0050041CB445__INCLUDED_)
```

```
// Length.cpp: implementation of the CLength class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Length.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CLength::CLength()
    : value(0.0), units(m), valueInMeters(0), valueIsCurrent(true)
{
    return;
}

CLength::CLength(const CLength& srcLength)
{
    // Set valueInMeters:

    valueInMeters = srcLength.valueInMeters;

    // Call set units, which will adjust value and valueIsCurrent
    //    appropriately, as well as setting units:

    SetUnits(srcLength.units);
}

CLength::CLength(double length, enum CLength::Units units)
    : value(length), units(units), valueIsCurrent(true)
{
    valueInMeters = ConvertToMeters(length, units);
    return;
}

void CLength::SetLength(double newValue, enum CLength::Units newUnits)
{
    this->CLength::CLength(newValue, newUnits);
}

CLength::~CLength()
{
    return;
}

enum CLength::Units CLength::GetUnits() const
{
    return units;
}

void CLength::SetUnits(enum CLength::Units newUnits)
{
    // Convert value to meters, then to appropriate units:
    value = ConvertFromMeters(valueInMeters, newUnits);
    valueIsCurrent = true;

    // Change units attribute:
    units = newUnits;
}

double CLength::GetValue()
```

```
{
    if(!valueIsCurrent)
    {
        value = ConvertFromMeters(valueInMeters, units);
        valueIsCurrent = true;
    }

    return value;
}

void CLength::SetValue(double newValue)
{
    value = newValue;
    valueIsCurrent = true;
    valueInMeters = ConvertToMeters(newValue, units);
}

CLength::operator CString()
{
    if(!valueIsCurrent)
    {
        value = ConvertFromMeters(valueInMeters, units);
        valueIsCurrent = true;
    }

    CString result;

    switch(units)
    {
    case m:
        result.Format("%lf m", value);
        break;
    case km:
        result.Format("%lf km", value);
        break;
    case in:
        result.Format("%lf in", value);
        break;
    case ft:
        result.Format("%lf ft", value);
        break;
    case mi:
        result.Format("%lf mi", value);
        break;
    default:
        result.Format("%lf", value);
        break;
    }

    return result;
}

CLength::operator double&()
{
    valueIsCurrent = false;
    return valueInMeters;
}

CLength::operator double() const
{
    return valueInMeters;
}

CLength::operator int() const
{
    return units;
}
```

```
CLength CLength::operator +(const CLength& rLength) const
{
    // Create a CLength (in meters by default) that is the sum of the
    //    two arguments in meters, then convert the result to the
    //    units of the first argument:


    CLength result(this->valueInMeters + rLength.valueInMeters);
    result.SetUnits(this->units);


    // Return the result:

    return result;
}

CLength CLength::operator /(const double divisor) const
{
    // Create a CLength object with same units as this, but value
    //    divided by the divisor.  Calling GetValue() rather than
    //    reading value directly ensures that the value is curent
    //    with the valueInMeters, which should always be current:

    return CLength(ConvertFromMeters(this->valueInMeters / divisor, this->units),
        this->units);
}

CLength& CLength::operator =(const CLength& rLength)
{
    this->CLength::CLength(rLength);

    return *this;
}

CLength& CLength::operator +=(const CLength& rLength)
{
    return *this = *this + rLength;
}

bool CLength::operator <(const CLength& rLength) const
{
    return this->valueInMeters < rLength.valueInMeters;
}

bool CLength::operator >(const CLength& rLength) const
{
    return this->valueInMeters > rLength.valueInMeters;
}

double CLength::GetValueInMeters() const
{
// GetValueInMeters function:  Returns the value of the given length
//                              in meters.

    // Call ConvertToMeters() function:

    //return ConvertToMeters(value, units);
    return valueInMeters;
}

double CLength::ConvertToMeters(double length,
                                enum CLength::Units currentUnits
                                ) const
{
// ConvertToMeters function:  Returns the value of the given length
//                             in meters.

    // Conversion factors below are from the National Institute of
    //    Standards and Technology and are exact.

    // Return value adjusted by the appropriate conversion factor:

    switch(currentUnits)
    {
    case m:
        return length;
        break;
    case km:
        return length * 1000.0;
        break;
    case in:
        return length * 0.0254;
        break;
    case ft:
        return length * 0.3048;
        break;
    case mi:
        return length * 1609.344;
        break;
    default:
        return length;
        break;
    }
}

double CLength::ConvertFromMeters(double length,
                                enum CLength::Units newUnits
                                ) const
{
// ConvertFromMeters function:  Returns the value of the given (m)
//                              length in newUnits.

    // Conversion factors below are from the National Institute of
    //    Standards and Technology and are exact.

    // Return value adjusted by the appropriate conversion factor:

    switch(newUnits)
    {
    case m:
        return length;
        break;
    case km:
        return length / 1000.0;
        break;
    case in:
        return length / 0.0254;
        break;
    case ft:
        return length / 0.3048;
        break;
    case mi:
        return length / 1609.344;
        break;
    default:
        return length;
        break;
    }
}
```

```
// Slope.h: interface for the CSlope class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_SLOPE_H__5672BA2F_3EAD_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_SLOPE_H__5672BA2F_3EAD_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// The CSlope class is used primarily for unit conversion.  It stores a value
// and returns it in units depending on the situation.  When cast as a double,
// it returns its value in m/km.  When cast as a CString, it returns a CString
// object consisting of it numerical value and a unit label.

class CSlope
{
public:
// Units enumeration:
    enum Units
    {
        m_km, in_mi, pureSlope
    };

private:
// Attributes:
    double value;
    Units units;

public:
// Construction/Destruction:
    CSlope();
    CSlope(double slope, enum Units units = m_km);
    virtual ~CSlope();

// Attribute functions:
    enum Units GetUnits() const;
    void SetUnits(enum Units newUnits);
    double GetValue() const;
    double GetValueInM_Km() const;
    void SetValue(double newValue);

// Casting operators:
    operator CString() const;
    operator double() const;
    operator int() const;

// Private member functions:
private:
    double ConvertToM_Km(double slope, enum Units currentUnits) const;
    double ConvertFromM_Km(double slope, enum Units newUnits) const;
};

#endif // !defined(AFX_SLOPE_H__5672BA2F_3EAD_11D3_B226_0050041CB445__INCLUDED_)
```

```
// Slope.cpp: implementation of the CSlope class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Slope.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CSlope::CSlope()
    : value(0), units(m_km)
{
    return;
}

CSlope::CSlope(double slope, enum CSlope::Units units)
    : value(slope), units(units)
{
    return;
}

CSlope::~CSlope()
{
    return;
}

enum CSlope::Units CSlope::GetUnits() const
{
    return units;
}

void CSlope::SetUnits(enum CSlope::Units newUnits)
{
    value = ConvertToM_Km(value, units);
    value = ConvertFromM_Km(value, newUnits);

    units = newUnits;
}

double CSlope::GetValue() const
{
    return value;
}

double CSlope::GetValueInM_Km() const
{
    return ConvertToM_Km(value, units);
}

void CSlope::SetValue(double newValue)
{
    value = newValue;
}

CSlope::operator CString() const
{
// CString operator:  Returns the numerical value and units label for the
//                    CSlope object.

    // Variable to hold the result:
```

```
    CString result;

    // Write the number and appropriate abbreviation to the string:
    switch(units)
    {
    case m_km:
        result.Format("%lf m/km", value);
        break;
    case in_mi:
        result.Format("%lf in/mi", value);
        break;
    case pureSlope:
        result.Format("%lf", value);
        break;
    default:
        result.Format("%lf", value);
        break;
    }

    // Return the result:
    return result;
}

CSlope::operator double() const
{
    // Return the value in m/km:
    return ConvertToM_Km(value, units);
}

CSlope::operator int() const
{
    // Give a number corresponding to the units (for use in switch statements):
    return units;
}

double CSlope::ConvertToM_Km(double slope,
                             enum CSlope::Units currentUnits) const
{
// ConvertToM_Km function:  Returns the value of the given slope
//                          in meters per kilometer.

    // Conversion factors below are from the National Institute of
    //    Standards and Technology and are exact.

    // Return value adjusted by the appropriate conversion factor:

    switch(currentUnits)
    {
    case m_km:
        return slope;
        break;
    case in_mi:
        return slope * 0.0254 / 1.609344;
        break;
    case pureSlope:
        return slope / 0.001;
        break;
    default:
        return slope;
        break;
    }
}

double CSlope::ConvertFromM_Km(double slope,
               enum CSlope::Units newUnits) const
{
// ConvertFromM_Km function:  Returns the value of the given (m/km)
//                            slope in newUnits.
```

```
    // Conversion factors below are from the National Institute of
    //    Standards and Technology and are exact.

    // Return value adjusted by the appropriate conversion factor:

    switch(newUnits)
    {
    case m_km:
        return slope;
        break;
    case in_mi:
        return slope * 1.609344 / 0.0254;
        break;
    case pureSlope:
        return slope * 0.001;
        break;
    default:
        return slope;
        break;
    }
}
```

A.9 Array Classes

```
// ObjectArray.h: interface for the CObjectArray class.
//
//////////////////////////////////////////////////////////////////////

// CObjectArray is a template that will encapsulate an array of any object
// type.  It grows automatically as elements are added.  For example, if
// the array has elements 0 to 5, and element 6 is accessed, a default
// object will be allocated for element 6.  All of the code below is adapted
// from Microsoft's CArray class template.

#if !defined(AFX_OBJECTARRAY_H__AC967DA5_4FF2_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_OBJECTARRAY_H__AC967DA5_4FF2_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <afxtempl.h>

template <class TYPE>
class CObjectArray : public CArray<TYPE, TYPE&>
{
public:
    CObjectArray(){};
    virtual ~CObjectArray(){};

// CArray overrides (NOT virtual):
    TYPE& ElementAtGrow(int nIndex);
    TYPE& operator[](int nIndex) const;
    TYPE& operator[](int nIndex);
};

template <class TYPE>
AFX_INLINE TYPE& CObjectArray<TYPE>::ElementAtGrow(int nIndex)
{
    ASSERT_VALID(this);
    ASSERT(nIndex >= 0);
    if (nIndex >= m_nSize)
        SetSize(nIndex+1, -1);
    return m_pData[nIndex];
}

template <class TYPE>
AFX_INLINE TYPE& CObjectArray<TYPE>::operator[](int nIndex) const
{
    ASSERT(nIndex >= 0 && nIndex < m_nSize);
    return m_pData[nIndex];
}

template <class TYPE>
AFX_INLINE TYPE& CObjectArray<TYPE>::operator[](int nIndex)
{
    return ElementAtGrow(nIndex);
}

#endif // !defined(AFX_OBJECTARRAY_H__AC967DA5_4FF2_11D3_B226_0050041CB445__INCLUDED_)
```

```
// IRIProfileArray.h: interface for the CIRIProfileArray class.
//
//////////////////////////////////////////////////////////////////////

// The CIRIProfileArray class is a modification of Microsoft's CArray that
// holds an array of type CIRIProfile.  It will automatically grow if
// elements beyond the end of the array are accessed.  It can be saved
// through serialization.

#if !defined(AFX_IRIPROFILEARRAY_H__AC967DA9_4FF2_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_IRIPROFILEARRAY_H__AC967DA9_4FF2_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <afxtempl.h>

class CIRIProfileArray : public CArray<CIRIProfile, CIRIProfile&>
{
public:
    CIRIProfileArray(){};
    virtual ~CIRIProfileArray(){};
    CIRIProfileArray(const CIRIProfileArray& srcProfileArray);

// Assignment operator:
    CIRIProfileArray& operator =(const CIRIProfileArray& rProfileArray);

// CArray overrides (NOT virtual):
    CIRIProfile& ElementAtGrow(int nIndex);
    CIRIProfile& operator[](int nIndex) const;
    CIRIProfile& operator[](int nIndex);
    void Serialize(CArchive& ar);
};

AFX_INLINE CIRIProfile& CIRIProfileArray::ElementAtGrow(int nIndex)
{
    ASSERT_VALID(this);
    ASSERT(nIndex >= 0);
    if (nIndex >= m_nSize)
        SetSize(nIndex+1, -1);
    return m_pData[nIndex];
}

AFX_INLINE CIRIProfile& CIRIProfileArray::operator[](int nIndex) const
{
    ASSERT(nIndex >= 0 && nIndex < m_nSize);
    return m_pData[nIndex];
}

AFX_INLINE CIRIProfile& CIRIProfileArray::operator[](int nIndex)
{
    return ElementAtGrow(nIndex);
}

AFX_INLINE void CIRIProfileArray::Serialize(CArchive& ar)
{
    ASSERT_VALID(this);

    CObject::Serialize(ar);

    if (ar.IsStoring())
    {
        ar.WriteCount(m_nSize);
        for (int i = 0; i < m_nSize; i++)
            m_pData[i].Serialize(ar);
    }
    else
    {
        DWORD nOldSize = ar.ReadCount();
        SetSize(nOldSize);
        for (int i = 0; i < m_nSize; i++)
            m_pData[i].Serialize(ar);
    }
}

CIRIProfileArray::CIRIProfileArray(const CIRIProfileArray& srcProfileArray)
{
    *this = srcProfileArray;
}

AFX_INLINE CIRIProfileArray& CIRIProfileArray::operator =
    (const CIRIProfileArray& rProfileArray)
{
    this->SetSize(rProfileArray.m_nSize);

    for(int i = 0; i < rProfileArray.m_nSize; i++)
        (*this)[i] = rProfileArray[i];

    return *this;
}


#endif
// !defined(AFX_IRIPROFILEARRAY_H__AC967DA9_4FF2_11D3_B226_0050041CB445__INCLUDED_)
// DoubleArray.h: interface for the CDoubleArray class.
//
//////////////////////////////////////////////////////////////////////

// CDoubleArray is a modified version of Microsoft's CArray, adapted for
// storing type double.  It grows automatically if elements are accessed
// beyond the endpoint of the array.  It is can be serialized, making it
// easy to save from the CRPADoc class.

#if !defined(AFX_DOUBLEARRAY_H__AC967DA6_4FF2_11D3_B226_0050041CB445__INCLUDED_)
#define AFX_DOUBLEARRAY_H__AC967DA6_4FF2_11D3_B226_0050041CB445__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <afxtempl.h>

class CDoubleArray : public CArray<double, double>
{
public:
    CDoubleArray(){};
    virtual ~CDoubleArray(){};

// CArray overrides (NOT virtual):
    double& ElementAtGrow(int nIndex);
    double& operator[](int nIndex) const;
    double& operator[](int nIndex);
};

AFX_INLINE double& CDoubleArray::ElementAtGrow(int nIndex)
{
    ASSERT_VALID(this);
    ASSERT(nIndex >= 0);
    if (nIndex >= m_nSize)
        SetSize(nIndex+1, -1);
    return m_pData[nIndex];
}

AFX_INLINE double& CDoubleArray::operator[](int nIndex) const
{
    ASSERT(nIndex >= 0 && nIndex < m_nSize);
    return m_pData[nIndex];
}

AFX_INLINE double& CDoubleArray::operator[](int nIndex)
{
    return ElementAtGrow(nIndex);
}

#endif // !defined(AFX_DOUBLEARRAY_H__AC967DA6_4FF2_11D3_B226_0050041CB445__INCLUDED_)
```

B Source Code for Filter-Response Evaluation

```
/***************************************************************
 *
 *  HEADER:         im.h
 *
 *
 *  DESCRIPTION:    Header file for I&M functions.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *                  Justin S. Dyer
 *
 *
 *  DATE CREATED:   24 January 1999     Version 1.01
 *
 *
 *  REVISIONS:      30Mar99 Add POLYN and RATIONAL data types,
 *                          with attendant #define's and
 *                          function declarations.
 *                   7Sep99 Add constants, declarations.
 *                   8Sep99 Add gen_freqs(), LINEAR_SWEEP,
 *                          LOG_SWEEP.
 *
 *
***************************************************************/

//--------------------------------------------------------------
//  Make sure that this header file is included only once.
//--------------------------------------------------------------
#ifndef im_h
#define im_h


//--------------------------------------------------------------
//  Do not use Microsoft  struct complex  found in math.h.
//  im.h must be #include'd before math.h if math.h is
//  needed and compilation is to occur using Microsoft C.
//--------------------------------------------------------------
#define _COMPLEX_DEFINED


//==============================================================
//  Type definitions.
//--------------------------------------------------------------

typedef struct
    {
    double re,                  // Real part.
           im;                  // Imaginary part.
    } COMPLEX;


                                // Polynomial having real-
                                // valued coefficients.
typedef struct
    {
    int     degree;             // Degree of the polynomial.
    double  *c;                 // Pointer to the coefficients
                                // of the polynomial.
    } POLYN;


                                // Rational function having
                                // real-valued coefficients.
typedef struct
```

```
    {
POLYN num;                   // Numerator polynomial.
POLYN denom;                 // Denominator polynomial.
    } RATIONAL;



                               /* Real-valued matrix.         */
typedef struct
    {
    int     rows;               /* Number of rows in the       */
                                /* matrix.                     */
    int     cols;               /* Number of columns in the    */
                                /* matrix.                     */
    double **a;                 /* Pointer to the components   */
                                /* of the matrix.              */
    } D_MATRIX;


//=================================================================
//  Return-error codes.
//-----------------------------------------------------------------

#define NORMAL          0       // No error detected.

#define CERR_CPH       10       // cphase(); cphased().


#define ERR_MKPOLYN  1010       // make_polyn()
#define ERR_FRPOLYN  1020       // free_polyn()

#define ERR_MKRAT    1030       // make_rational()
#define ERR_CPRAT    1035       // copy_rational()
#define ERR_FRRAT    1040       // free_rational()

#define ERR_MKMAT    1050       // make_matrix()
#define ERR_FRMAT    1060       // free_matrix()

#define ERR_MKDMAT   1070       // make_dmatrix()
#define ERR_FRDMAT   1080       // free_dmatrix()

#define ERR_CT       2010       // ct_response()
#define ERR_DT       2020       // dt_response()

#define ERR_FS       2030       // f_scale()

#define ERR_LP_HP    2040       // lp_to_hp()
#define ERR_LP_BP    2050       // lp_to_bp()
#define ERR_LP_BS    2060       // lp_to_bs()

#define ERR_BLT      2070       // blt()

#define ERR_CLUD     2080       // c_lud(); c_lud1
#define ERR_CSUBST   2090       // c_subst(); c_subst1


#define ERR_GENFREQS 9000       // gen_freqs()


//=================================================================
//  Constants.
//-----------------------------------------------------------------

#ifndef NORMAL
#define NORMAL      0
#endif
```

```
#ifndef NO
#define NO          0
#endif

#ifndef YES
#define YES         1
#endif

#define RAD         0           // Frequencies are in rad/sec.

#define HZ          1           // Frequencies are in Hz.



#define NULL_POLYN  -1          // "degree" of a null-polynomial.



#define PI          3.14159265358979323846264З
#define TWO_PI      6.283185307179586476925287


                                // 1 rad/deg.
#define RAD_DEG     0.0174532925199432957692З7

                                // 1 deg/rad.
#define DEG_RAD     57.2957795130823208767981З5


#define LINEAR_SWEEP    0       // Linearly spaced frequencies.
#define LOG_SWEEP       1       // Logarithmically (base-10)
                                //   spaced frequencies.


//============================================================//
//  Macros.                                                   //
//------------------------------------------------------------//

#define ABS(x)          ((x) > 0 ? (x) : -(x))
#define MAX(a, b)       ((a) > (b) ? (a) : (b))
#define MIN(a, b)       ((a) < (b) ? (a) : (b))




//==============================================================
//  Function declarations.
//--------------------------------------------------------------

COMPLEX     cadd            (COMPLEX x, COMPLEX y);
COMPLEX     cconj           (COMPLEX x);
COMPLEX     cdiv            (COMPLEX x, COMPLEX y);
COMPLEX     cexpon          (double theta);
double      cmag            (COMPLEX x);

double      cmagsq          (COMPLEX x);
COMPLEX     cmplx           (double x, double y);
COMPLEX     cmult           (COMPLEX x, COMPLEX y);
COMPLEX     cneg            (COMPLEX x);
double      cphase          (COMPLEX x, int *error_ptr);

double      cphased         (COMPLEX x, int *error_ptr);
COMPLEX     csqrt           (COMPLEX x);
COMPLEX     csub            (COMPLEX x, COMPLEX y);

int         ct_response     (RATIONAL xfer_fcn, int num_freqs,
                             int rad_or_hz, double frequencies[],
                              COMPLEX response[]);

int          dt_response      (double f_s, RATIONAL xfer_fcn,
                               int num_freqs, int rad_or_hz,
                               double frequencies[],
                               COMPLEX response[]);

COMPLEX      eval_polyn       (COMPLEX x, POLYN p_x);
COMPLEX      eval_rational    (COMPLEX x, RATIONAL r_x);

int          f_scale          (RATIONAL *xfer_fcn,
                               double sc_factor);

int          free_dmatrix     (D_MATRIX *ptr_dmat);

int          free_matrix      (int row_size, char **ptr_to_mat);

int          free_polyn       (POLYN *ptr_polyn);
int          free_rational    (RATIONAL *ptr_rat);


int          gen_freqs(int sweep, int num_freqs,
                      double first_freq, double last_freq,
                      double frequencies[]);



int          lp_to_bp         (RATIONAL *lp, RATIONAL *bp,
                               double bp_w_c, double bp_bw);
int          lp_to_bs         (RATIONAL *lp, RATIONAL *bs,
                               double bs_w_c, double bs_bw);
int          lp_to_hp         (RATIONAL *lp, RATIONAL *hp);

int          make_dmatrix     (int row_size, int col_size,
                               D_MATRIX *ptr_dmat);

char         **make_matrix    (int row_size, int col_size,
                               unsigned element_size);

int          make_polyn       (int degree, POLYN *ptr_polyn);
int          make_rational    (int num_degree, int denom_degree,
                               RATIONAL *ptr_rat);

int          copy_rational( RATIONAL* pFrom, RATIONAL* pTo );


#endif
```

```
/****************************************************************
 *
 *  SOURCE FILE:    complex.c
 *
 *
 *  DESCRIPTION:    Basic complex-arithmetic functions:
 *
 *                      cadd(x, y)
 *                      cconj(x)
 *                      cdiv(x, y)
 *                      cexpon(theta)
 *                      cmag(x)
 *                      cmagsq(x)
 *                      cmplx(x, y)
 *                      cmult(x, y)
 *                      cneg(x, y)
 *                      cphase(x, error_ptr)
 *                      cphased(x, error_ptr)
 *                      csqrt(x)
 *                      csub(x, y)
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer
 *                  Justin S. Dyer
 *
 *
 *  DATE CREATED:   24 January 1999     Version 1.00
 *
 *
 *  REVISIONS:
 *
 *
****************************************************************/

#include "im.h"
#include <math.h>


/*-------------------------------------------------------------*/
/*  cadd(x, y)      Returns the sum of two complex numbers     */
/*                  x and y.                                   */
/*-------------------------------------------------------------*/

COMPLEX cadd(COMPLEX x, COMPLEX y)
    {
    x.re += y.re;
    x.im += y.im;

    return (x);
    }



/*-------------------------------------------------------------*/
/*  cconj(x)        Returns the conjugate of x.                */
/*-------------------------------------------------------------*/

COMPLEX cconj(COMPLEX x)
    {
    x.im = - x.im;

    return (x);
    }
```

```
/*-----------------------------------------------------------*/
/*  cdiv(x, y)      Divides x by y.                          */
/*-----------------------------------------------------------*/

COMPLEX cdiv(COMPLEX x, COMPLEX y)
    {
    COMPLEX quotient;
    double  recip_denom;


    recip_denom = 1.0 / (y.re * y.re  +  y.im * y.im);

    quotient.re = recip_denom * (x.re * y.re  +  x.im * y.im);
    quotient.im = recip_denom * (x.im * y.re  -  x.re * y.im);

    return (quotient);
    }



/*-----------------------------------------------------------*/
/*  cexpon(theta)   Returns exp(j * theta).                  */
/*-----------------------------------------------------------*/

COMPLEX cexpon(double theta)
    {
    COMPLEX expon;

    expon.re = cos(theta);
    expon.im = sin(theta);

    return (expon);
    }



/*-----------------------------------------------------------*/
/*  cmag(x)         Returns the modulus (magnitude) of x.    */
/*-----------------------------------------------------------*/

double  cmag(COMPLEX x)
    {
    return (sqrt(x.re * x.re  +  x.im * x.im));
    }



/*-----------------------------------------------------------*/
/*  cmagsq(x)       Returns the square of the modulus        */
/*                  (magnitude) of x.                        */
/*-----------------------------------------------------------*/

double  cmagsq(COMPLEX x)
    {
    return (x.re * x.re  +  x.im * x.im);
    }



/*-----------------------------------------------------------*/
/*  cmplx(x, y)     Returns the complex number x + jy.       */
/*-----------------------------------------------------------*/

COMPLEX cmplx(double x, double y)
    {
    COMPLEX complex_no;
    complex_no.re = x;
    complex_no.im = y;

    return (complex_no);
    }



/*------------------------------------------------------------*/
/*  cmult(x, y)     Returns the product of x and y.           */
/*------------------------------------------------------------*/

COMPLEX cmult(COMPLEX x, COMPLEX y)
    {
    COMPLEX product;

    product.re = x.re * y.re  -  x.im * y.im;
    product.im = x.re * y.im  +  x.im * y.re;

    return (product);
    }



/*------------------------------------------------------------*/
/*  cneg(x)         Returns the negative of x.                */
/*------------------------------------------------------------*/

COMPLEX cneg(COMPLEX x)
    {
    x.re = - x.re;
    x.im = - x.im;

    return (x);
    }



/*------------------------------------------------------------*/
/*  cphase(x, error_ptr)                                      */
/*                  Returns the phase, in radians, of x.   If */
/*                  x = 0, then cphase() returns 0.           */
/*                                                            */
/*                  Value of *error_ptr:                      */
/*                      NORMAL:    No error.                  */
/*                      CERR_CPH:  Argument x = 0.            */
/*------------------------------------------------------------*/

double  cphase(COMPLEX x, int *error_ptr)
    {
    if (x.re == 0.0 && x.im == 0.0)
        {
        *error_ptr = CERR_CPH;
        return (0.0);
        }
    else
        {
        *error_ptr = NORMAL;
        return (atan2(x.im, x.re));
        }
    }



/*------------------------------------------------------------*/
/*  cphased(x, error_ptr)                                     */
/*                  Returns the phase, in degrees, of x.   If */
/*                  x = 0, then cphased() returns 0.           */
/*                                                             */
/*                  Value of *error_ptr:                       */
/*                      NORMAL:    No error.                   */
/*                      CERR_CPH:  Argument x = 0.             */
/*-------------------------------------------------------------*/


double  cphased(COMPLEX x, int *error_ptr)
    {
    if (x.re == 0.0 && x.im == 0.0)
        {
        *error_ptr = CERR_CPH;
        return (0.0);
        }
    else
        {
        *error_ptr = NORMAL;
        return (DEG_RAD * atan2(x.im, x.re));
        }
    }



/*-------------------------------------------------------------*/
/*  csqrt(x)        Returns the square root of the complex     */
/*                  number x.                                  */
/*                  The principal value having the real part   */
/*                  greater than zero is returned.  If the     */
/*                  real part equals zero, the principal value */
/*                  having the imaginary part greater than     */
/*                  zero is returned.                          */
/*-------------------------------------------------------------*/

COMPLEX csqrt(COMPLEX x)
    {
    double  r, theta;

    COMPLEX c_sqrt;

    if (x.re == 0.0 && x.im == 0.0)
        {
        c_sqrt.re = 0.0;
        c_sqrt.im = 0.0;
        return (c_sqrt);
        }

    r = pow(x.re * x.re + x.im * x.im, 0.25);
    theta = 0.5 * atan2(x.im, x.re);

    c_sqrt.re = r * cos(theta);
    c_sqrt.im = r * sin(theta);

    if (c_sqrt.re < 0.0)
        {
        c_sqrt.re = - c_sqrt.re;
        c_sqrt.im = - c_sqrt.im;
        }
    else
        if (c_sqrt.re == 0.0)
            c_sqrt.im = fabs(c_sqrt.im);

    return (c_sqrt);
    }


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

```
/*  csub(x, y)      Returns the difference (x - y) of two      */
/*                  complex numbers x and y.                   */
/*-------------------------------------------------------------*/

COMPLEX csub(COMPLEX x, COMPLEX y)
    {
    x.re -= y.re;
    x.im -= y.im;

    return (x);
    }
```

```
/***************************************************************
 *
 *  SOURCE:         mk_polyn.c
 *
 *
 *  FUNCTION:       make_polyn(degree, ptr_polyn)
 *
 *
 *  DESCRIPTION:    Allocates memory for the real-valued
 *                  coefficients of a polynomial of degree
 *                   degree .  Memory for  degree + 1
 *                  coefficients is allocated.
 *
 *
 *  ARGUMENTS:
 *
 *      degree      (input)  int
 *                  Degree of the polynomial to be
 *                  created.
 *
 *      ptr_polyn   (output) POLYN *
 *                  Pointer to the structure describing the
 *                  coefficients of the polynomial just made.
 *                  The coefficients are of type double.
 *
 *
 *  RETURN:         int
 *                  NORMAL     :  Normal return.
 *                  ERR_MKPOLYN:  An error has occurred in
 *                                allocating memory for the
 *                                polynomial.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D.
 * Justin S. Dyer
 *
 *
 *  DATE CREATED:   28 March 1999   Version 1.00
 *
 *
 *  REVISIONS:      7Sep99  Additional error-checking.
 *
 *
 ***************************************************************/

#include <stdio.h>
#include <malloc.h>
#include "im.h"


int make_polyn( int degree, POLYN *ptr_polyn)

    {

//  ------------------------------------------------------------
//  Initialize ptr_polyn in anticipation of error.
//  ------------------------------------------------------------
    ptr_polyn->degree  = NULL_POLYN;
    ptr_polyn->c       = NULL;


//  ------------------------------------------------------------
//  Make sure that  degree  is valid.
//  ------------------------------------------------------------
    if (degree < 0  && degree != NULL_POLYN)
        return (ERR_MKPOLYN);
//  -------------------------------------------------------------
//  Allocate memory for the coefficients of the polynomial.
//  -------------------------------------------------------------
    if (degree >= 0)
        {
        if ( (ptr_polyn->c = (double *)
            malloc((unsigned)((degree + 1) * sizeof(double))))
            == NULL)
        return (ERR_MKPOLYN);
        }

    ptr_polyn->degree = degree;
    return (NORMAL);
    }
```

```
/**************************************************************
 *
 *  SOURCE:         fr_polyn.c
 *
 *
 *  FUNCTION:       free_polyn(ptr_polyn)
 *
 *
 *  DESCRIPTION:    Deallocates memory associated with the
 *                  real-valued polynomial coefficients pointed
 *                  to by  ptr_polyn .
 *
 *
 *  ARGUMENTS:
 *
 *      ptr_polyn   (input)  POLYN *
 *                  Pointer to the structure that describes the
 *                  polynomial (coefficients) to be freed.
 *
 *
 *  RETURN:         int
 *                  NORMAL     :  Normal return.
 *                  ERR_FRPOLYN:  An error has occurred during
 *                                the attempt to free the
 *                                polynomial.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D.
 *                  Justin S. Dyer
 *
 *
 *  DATE CREATED:   28 March 1999   Version 1.00
 *
 *
 *  REVISIONS:      None.
 *
 *
 *************************************************************/

#include <stdio.h>
#include <malloc.h>
#include "im.h"


int         free_polyn(POLYN *ptr_polyn)
{
    int     err_code;

    err_code = NORMAL;

    if (ptr_polyn->degree < 0)
        err_code = ERR_FRPOLYN;


/*------------------------------------------------------------*/
/*  Deallocate memory associated with the elements of the      */
/*  coefficient array.                                         */
/*------------------------------------------------------------*/
    if (ptr_polyn->c == NULL)
        err_code = ERR_FRPOLYN;
    else
        free((char *)ptr_polyn->c);


    ptr_polyn->degree  = NULL_POLYN;
    ptr_polyn->c       = NULL;
    return (err_code);
}
```

```
/****************************************************************
 *
 *  SOURCE FILE:    eval_polyn.c
 *
 *
 *  FUNCTION:       eval_polyn(x, p_x)
 *
 *
 *  DESCRIPTION:    Evaluates the polynomial  p_x  at the value
 *                   x  of the independent variable.
 *
 *
 *  ARGUMENTS:
 *
 *      x           (input)  COMPLEX
 *                  Value, of the independent variable, at which
 *                  the polynomial  p_x  is to be evaluated.
 *
 *      p_x         (input)  POLYN
 *                  Structure containing the degree and
 *                  coefficients of the polynomial to be
 *                  evaluated.
 *
 *
 *  RETURN:         COMPLEX
 *                  Value of the polynomial  p_x  at  x .
 *
 *
 *  AUTHOR:         Copyright 1989
 *                  Stephen A. Dyer, Ph.D.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
****************************************************************/

#include "im.h"
#include <math.h>


//#include "polyn.h"


COMPLEX eval_polyn(COMPLEX x, POLYN p_x)

    {
    COMPLEX p;
    int     i;


    p = cmplx(p_x.c[p_x.degree], 0.0);

    for (i = p_x.degree - 1; i >= 0; i--)
        p = cadd(cmplx(p_x.c[i], 0.0), cmult(x, p));


    return (p);
    }
```

```
/***************************************************************
 *
 *  SOURCE:         make_rational.c
 *
 *
 *  FUNCTION:       make_rational(num_degree, denom_degree,
 *                      ptr_rat)
 *
 *
 *  DESCRIPTION:    Allocates memory for the real-valued
 *                  coefficients of a rational function whose
 *                  numerator is of degree  max_num_deg and
 *                  whose denominator is of degree
 *                   max_denom_deg.  Memory for  max_num_deg + 1
 *                  numerator coefficients and
 *                   max_denom_deg + 1 is allocated.
 *
 *
 *
 *  ARGUMENTS:
 *
 *      num_degree  (input)  int
 *                  Degree of the numerator of the rational
 *                  function to be created.
 *
 *      denom_degree
 *                  (input)  int
 *                  Degree of the denominator of the rational
 *                  function to be created.
 *
 *      ptr_rat     (output) RATIONAL *
 *                  Pointer to the structure describing the
 *                  coefficients of the rational function.
 *                  The coefficients are of type double.
 *
 *
 *  RETURN:         int
 *                  NORMAL     :  Normal return.
 *                  ERR_MKRAT  :  An error has occurred in
 *                                allocating memory for the
 *                                rational function.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D.
 *                  Justin S. Dyer
 *
 *
 *  DATE CREATED:   26 March 1999       Version 1.00
 *
 *
 *  REVISIONS:      7Sep99  Correct initialization.
 *
 *
***************************************************************/

#include <stdio.h>
#include <malloc.h>
#include "im.h"


int         make_rational(int num_degree,
                          int denom_degree,
                          RATIONAL *ptr_rat)
{
/*-----------------------------------------------------------*/
/*  Allocate memory for the coefficients of the numerator.     */
/*------------------------------------------------------------*/
    if (make_polyn(num_degree, &(ptr_rat->num)))
        return (ERR_MKRAT);


/*------------------------------------------------------------*/
/*  Allocate memory for the coefficients of the denominator.   */
/*------------------------------------------------------------*/
    if (make_polyn(denom_degree, &(ptr_rat->denom)))
        return (ERR_MKRAT);

    return (NORMAL);
}
```

```
/**************************************************************
 *
 *  SOURCE:         fr_rat.c
 *
 *
 *  FUNCTION:       free_rational(ptr_rat)
 *
 *
 *  DESCRIPTION:    Deallocates memory associated with the
 *                  real-valued numerator and denominator
 *                  coefficients pointed to by  ptr_rat .
 *
 *
 *  ARGUMENTS:
 *
 *      ptr_rat     (input)  RATIONAL *
 *                  Pointer to the structure that describes the
 *                  rational-function (coefficients) to be freed.
 *
 *
 *  RETURN:         int
 *                  NORMAL     :  Normal return.
 *                  ERR_FRRAT  :  An error has occurred during
 *                                the attempt to free the
 *                                rational function.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D.
 *                  Justin S. Dyer
 *
 *
 *  DATE CREATED:   28 March 1999       Version 1.00
 *
 *
 *  REVISIONS:      None.
 *
 *
**************************************************************/

#include <stdio.h>
#include <malloc.h>
#include "im.h"


int         free_rational(RATIONAL *ptr_rat)
{
    int     err_code;

    err_code = NORMAL;

/*------------------------------------------------------------*/
/*  Deallocate memory associated with the elements of the      */
/*  coefficient arrays.                                        */
/*------------------------------------------------------------*/
    if (free_polyn(&(ptr_rat->num)))
        err_code = ERR_FRRAT;


    if (free_polyn(&(ptr_rat->denom)))
        err_code = ERR_FRRAT;


    return (err_code);
}
```

```
// Copy a rational to another rational...

#include "im.h"

int copy_rational( RATIONAL* pFrom, RATIONAL* pTo )
    {

    int i = 0;

    free_rational( pTo );
    if( make_rational( (pFrom->num).degree, (pFrom->denom).degree,
        pTo ) != NORMAL )
        return (ERR_CPRAT);

    (pTo->num).degree = (pFrom->num).degree;
    (pTo->denom).degree = (pFrom->denom).degree;

    for( i = 0; i <= (pTo->num).degree; i++ )
        (pTo->num).c[i] = (pFrom->num).c[i];

    for( i = 0; i <= (pTo->denom).degree; i++ )
        (pTo->denom).c[i] = (pFrom->denom).c[i];

    return (NORMAL);
    }
```

```
/**************************************************************
 *
 *  SOURCE FILE:    eval_rational.c
 *
 *
 *  FUNCTION:       eval_rational(x, r_x)
 *
 *
 *  DESCRIPTION:    Evaluates the rational function r_x  at the
 *                  value  x  of the independent variable.
 *
 *
 *  ARGUMENTS:
 *
 *      x           (input)  COMPLEX
 *                  Value, of the independent variable, at which
 *                  the rational function  r_x  is to be
 *                  evaluated.
 *
 *
 *      r_x         (input)  RATIONAL
 *                  Structure containing the degrees and
 *                  coefficients of the rational function to be
 *                  evaluated.
 *
 *
 *  RETURN:         COMPLEX
 *                  Value of the rational function  r_x  at  x .
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.00
 *
 *
 *  REVISIONS:      None.
 *
 *
**************************************************************/

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


COMPLEX eval_rational(COMPLEX x, RATIONAL r_x)

    {
    COMPLEX denom, num;


//  ------------------------------------------------------------
//  Calculate numerator.
//  ------------------------------------------------------------
    num = eval_polyn(x, r_x.num);


//  ------------------------------------------------------------
//  Calculate denominator.
//  ------------------------------------------------------------
    denom = eval_polyn(x, r_x.denom);


//  ------------------------------------------------------------
//  Find quotient.
//  ------------------------------------------------------------
    if (denom.re == 0.0  &&  denom.im == 0.0)
//      A pole.
        return (cmplx(HUGE, HUGE));
    else
        return (cdiv(num, denom));
    }
```

```
/****************************************************************
 *
 *  SOURCE:         make_matrix.c
 *
 *
 *  FUNCTION:       make_matrix(row_size, col_size, element_size)
 *
 *
 *  DESCRIPTION:    This is the low-level make-matrix function
 *                  which is invoked by user-accessible make-
 *                  matrix functions.  This function allocates
 *                  memory for a matrix of size  row_size x
 *                  col_size , each of whose elements is of
 *                  size element_size.
 *
 *
 *  ARGUMENTS:
 *
 *      row_size    (input)  int
 *                  Maximum number of rows for the matrix to
 *                  be created.
 *
 *      col_size    (input)  int
 *                  Maximum number of columns for the matrix to
 *                  be created.
 *
 *      element_size
 *                  (input)  unsigned
 *                  Size, in bytes, of each element within the
 *                  matrix.  E.g., a matrix whose elements are
 *                  of type COMPLEX has element_size of
 *                  sizeof(COMPLEX) = 16 bytes.
 *
 *
 *  RETURN:         **char
 *                  A value of NULL is returned if an error has
 *                  occurred in allocating memory for the matrix.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
****************************************************************/


#include <stdio.h>
#include <malloc.h>


char        **make_matrix(int row_size, int col_size,
                       unsigned element_size)

{
    char    **ptr_mat;
    int     i, j;

//------------------------------------------------------------
//  Allocate memory to accommodate an array of pointers to
//  arrays.  A total of  row_size  arrays will need to be
//  set up.
//-------------------------------------------------------------
    if ( (ptr_mat =
        (char **)malloc((unsigned)(row_size * sizeof(char *))))
        == NULL)
        return (NULL);

//-------------------------------------------------------------
//  Allocate memory to accommodate  row_size  arrays of
//  col_size  elements each.
//-------------------------------------------------------------
    for (i = 0; i < row_size; i++)
        {
        if ( (ptr_mat[i] =
            (char *)malloc((unsigned)(col_size * element_size)))
            == NULL)
            break;
        }

//  If unable to allocate all memory required, deallocate and
//  return an error.
    if (i != row_size)
        {
        for (j = 0; j < i; j++)
            free(ptr_mat[j]);

        free((char *)ptr_mat);
        return (NULL);
        }


    return (ptr_mat);
    }
```

```
/***************************************************************
 *
 *  SOURCE:         free_matrix.c
 *
 *
 *  FUNCTION:       free_matrix(row_size, ptr_to_mat)
 *
 *
 *  DESCRIPTION:    This is the low-level free-matrix function
 *                  which is invoked by user-accessible free-
 *                  matrix functions.  This function deallocates
 *                  memory for the matrix pointed to by
 *                   ptr_to_mat .
 *
 *
 *  ARGUMENTS:
 *
 *      row_size    (input)  int
 *                  Number of rows in the matrix to be freed.
 *
 *      ptr_to_mat  (input)  char **
 *                  Pointer to the matrix to be freed.
 *
 *
 *  RETURN:         void
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
***************************************************************/

#include <stdio.h>
#include <malloc.h>
#include "im.h"


int     free_matrix(int row_size, char **ptr_to_mat)

    {
    int     err_code, i;


//  Return-error code:
    err_code = NORMAL;


//  -----------------------------------------------------------
//  Deallocate memory associated with the  row_size  arrays.
//  -----------------------------------------------------------
    for (i = 0; i < row_size; i++)
        {
        if (ptr_to_mat[i] == NULL)
            {
//          An unexpected null-pointer was encountered.
            err_code = ERR_FRMAT;
            break;
            }

        free(ptr_to_mat[i]);
        ptr_to_mat[i] = NULL;
        }


//  -------------------------------------------------------------
//  Deallocate memory that accommodates the array of pointers
//  to arrays.
//  -------------------------------------------------------------
    if ((char *)ptr_to_mat == NULL)
        {
//      An unexpected null-pointer was encountered.
        err_code = ERR_FRMAT;
        }
    else
        {
        free((char *)ptr_to_mat);
        ptr_to_mat = NULL;
        }

    return (err_code);
    }
```

```
/***************************************************************
 *
 *  SOURCE:         make_dmatrix.c
 *
 *
 *  FUNCTION:       make_dmatrix(row_size, col_size, ptr_dmat)
 *
 *
 *  DESCRIPTION:    Allocates memory for a real-valued matrix
 *                  of size  row_size x col_size , each of whose
 *                  elements is of type double.
 *
 *
 *  ARGUMENTS:
 *
 *      row_size    (input)  int
 *                  Maximum number of rows for the matrix to
 *                  be created.
 *
 *      col_size    (input)  int
 *                  Maximum number of columns for the matrix to
 *                  be created.
 *
 *      ptr_dmat    (output) D_MATRIX *
 *                  Pointer to the structure describing the
 *                  matrix just made, each of
 *                  whose elements is of type double.
 *
 *
 *  RETURN:         int
 *                  NORMAL    :  Normal return.
 *                  ERR_MKDMAT:  An error has occurred in
 *                               allocating memory for the
 *                               matrix.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
***************************************************************/

#include <stdio.h>
#include <malloc.h>
#include "im.h"


int         make_dmatrix(int row_size, int col_size,
                         D_MATRIX *ptr_dmat)

    {
//  -----------------------------------------------------------
//  Initialize ptr_dmat in anticipation of error.
//  -----------------------------------------------------------
    ptr_dmat->rows     = 0;
    ptr_dmat->cols     = 0;
    ptr_dmat->a        = NULL;

    if (row_size < 1 || col_size < 1)
        return (ERR_MKDMAT);
//  -----------------------------------------------------------
//  Allocate memory for the elements of the matrix.
//  -----------------------------------------------------------
    if ( (ptr_dmat->a = (double **)
        make_matrix(row_size, col_size, sizeof(double)) ) == NULL)
        return (ERR_MKDMAT);


    ptr_dmat->rows = row_size;
    ptr_dmat->cols = col_size;
    return (NORMAL);
    }
```

```
/****************************************************************
 *
 *  SOURCE:         free_dmatrix.c
 *
 *
 *  FUNCTION:       free_dmatrix(ptr_dmat)
 *
 *
 *  DESCRIPTION:    Deallocates memory associated with a
 *                  real-valued matrix pointed to by
 *                  ptr_dmat.
 *
 *
 *  ARGUMENTS:
 *
 *      ptr_dmat    (input)  D_MATRIX *
 *                  Pointer to the structure which describes the
 *                  matrix to be freed.
 *
 *
 *  RETURN:         int
 *                  NORMAL    :  Normal return.
 *                  ERR_FRDMAT:  An error has occurred during
 *                               the attempt to free the matrix.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
****************************************************************/

#include <stdio.h>
#include <malloc.h>
#include "im.h"


int         free_dmatrix(D_MATRIX *ptr_dmat)

    {
    int     err_code;


/*  Return-error code:                                          */
    err_code = NORMAL;


    if (ptr_dmat->rows < 1  ||  ptr_dmat->cols < 1)
        return (ERR_FRDMAT);


/*--------------------------------------------------------------*/
/*  Deallocate memory associated with the elements of the      */
/*  matrix.                                                     */
/*--------------------------------------------------------------*/
    if (ptr_dmat->a == NULL)
        err_code = ERR_FRDMAT;
    else if (free_matrix(ptr_dmat->rows, (char **)ptr_dmat->a))
        err_code = ERR_FRDMAT;
    ptr_dmat->rows     = 0;
    ptr_dmat->cols     = 0;
    ptr_dmat->a = NULL;


    return (err_code);
    }
```

```
/*****************************************************************
 *
 *  SOURCE:         f_scale.c
 *
 *
 *  FUNCTION:       f_scale(xfer_fcn, sc_factor)
 *
 *
 *  DESCRIPTION:    Scales the transfer function  xfer_fcn  in
 *                  frequency by  sc_factor .
 *
 *                  The  xfer_fcn  is normalized with the
 *                  coefficient of the highest-degree term in
 *                  the denominator set to unity.
 *
 *
 *  ARGUMENTS:
 *
 *      xfer_fcn    (i/o)    RATIONAL *
 *                  Coefficients of the transfer function to be
 *                  frequency-scaled.   xfer_fcn  contains the
 *                  scaled coefficients upon return.
 *
 *      sc_factor   (input)  double
 *                  Scaling factor to be applied.  For example,
 *                  if a critical frequency of  xfer_fcn  is
 *                  1 rad/s and the revised  xfer_fcn  is to have
 *                  that critical frequency at 10e6 rad/s, then
 *                   sc_factor  is 1.0e6.
 *
 *
 *  RETURN:         int
 *                  NORMAL  :  Normal return
 *                  ERR_FS  :  An error occurred in the scaling
 *                             operation.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
*****************************************************************/

#include "im.h"


int     f_scale(RATIONAL *xfer_fcn, double sc_factor)

    {
    int     i;
    double  one_sf, scale;


//  Check for positive scale factor.
    if (sc_factor <= 0.0)
        return (ERR_FS);

//  Check for "valid" rational function.
    if ((xfer_fcn->num).degree < 0 ||
        (xfer_fcn->denom).degree < 0)
        return (ERR_FS);
//  -------------------------------------------------------------
//  Frequency-scale.
//  -------------------------------------------------------------
    if (sc_factor != 1.0)
        {
        one_sf = 1.0 / sc_factor;

        scale  = 1.0;
        for (i = 1;
            i <= MAX((xfer_fcn->num).degree,
            (xfer_fcn->denom).degree); i++)
            {
            scale *= one_sf;

            if (i <= (xfer_fcn->num).degree)
                (xfer_fcn->num).c[i] *= scale;

            if (i <= (xfer_fcn->denom).degree)
                (xfer_fcn->denom).c[i] *= scale;
            }
        }


//  -------------------------------------------------------------
//  Normalize s.t. the coefficient of the highest-degree term
//  in the denominator is unity.
//  -------------------------------------------------------------
    scale = 1.0 / (xfer_fcn->denom).c[(xfer_fcn->denom).degree];

    for (i = 0; i <= (xfer_fcn->num).degree; i++)
        (xfer_fcn->num).c[i] *= scale;

    for (i = 0; i < (xfer_fcn->denom).degree; i++)
        (xfer_fcn->denom).c[i] *= scale;

    (xfer_fcn->denom).c[(xfer_fcn->denom).degree] = 1.0;


    return (NORMAL);
    }
```

```
/****************************************************************
 *
 *  SOURCE FILE:    ct_response.c
 *
 *
 *  FUNCTION:       ct_response(xfer_fcn, num_freqs, rad_or_hz,
 *                      frequencies, response)
 *
 *
 *  DESCRIPTION:    Computes the frequency response of the
 *                  transfer function  xfer_fcn  at the
 *                  frequencies  requested.
 *
 *
 *  DOCUMENTATION
 *  FILES:          None.
 *
 *
 *  ARGUMENTS:
 *
 *      xfer_fcn    (input)  RATIONAL
 *                  Coefficients of the transfer function
 *                  to be evaluated.
 *
 *      num_freqs   (input)  int
 *                  Number of frequencies at which to evaluate
 *                  the frequency response.
 *
 *      rad_or_hz   (input)  int
 *                  0        :  frequencies are in rad/sec;
 *                  otherwise:  frequencies are in Hz.
 *
 *      frequencies (input)  double []
 *                  Frequencies at which the response is to be
 *                  evaluated.
 *
 *      response    (output) COMPLEX []
 *                  The values (real and imaginary parts) of
 *                  the frequency response, evaluated at the
 *                  frequencies  specified.
 *
 *
 *
 *  RETURN:         int
 *                  Normal  :  NORMAL
 *                  On error:  ERR_CT
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 2.01
 *
 *
 *  REVISIONS:      None.
 *
 *
****************************************************************/

#include "im.h"
#include <math.h>


int     ct_response(RATIONAL xfer_fcn, int num_freqs,
                    int rad_or_hz, double frequencies[],
                    COMPLEX response[])
```

```
{
int     i;


for (i = 0; i < num_freqs; i++)
    {
    if (rad_or_hz == HZ)
        response[i] = eval_rational
            (cmplx(0.0, TWO_PI * frequencies[i]), xfer_fcn);
    else
        response[i] = eval_rational
            (cmplx(0.0, frequencies[i]), xfer_fcn);
    }

return (NORMAL);
}
```

```
/***************************************************************
 *
 *  SOURCE FILE:    dt_response.c
 *
 *
 *  FUNCTION:       dt_response(f_s, xfer_fcn, num_freqs,
 *                      rad_or_hz, frequencies, response)
 *
 *
 *  DESCRIPTION:    Computes the discrete-time (DT) frequency
 *                  response of the transfer function  xfer_fcn
 *                  at the frequencies requested.
 *
 *
 *  ARGUMENTS:
 *
 *      f_s         (input)  double
 *                  Sampling frequency, in Hz, for the DT
 *                  system.  (f_s is the reciprocal of the
 *                  sampling interval T.)
 *
 *      xfer_fcn    (input)  RATIONAL
 *                  Coefficients of the transfer function
 *                  to be evaluated.
 *
 *      num_freqs   (input)  int
 *                  Number of frequencies at which to evaluate
 *                  the frequency response.
 *
 *      rad_or_hz   (input)  int
 *                  0        :  frequencies are in rad/sec;
 *                  otherwise:  frequencies are in Hz.
 *
 *      frequencies (input)  double []
 *                  Frequencies at which the response is to be
 *                  evaluated.
 *
 *      response    (output) COMPLEX []
 *                  The values (real and imaginary parts) of
 *                  the frequency response, evaluated at the
 *                  frequencies  specified.
 *
 *
 *
 *  RETURN:         int
 *                  Normal  :  NORMAL
 *                  On error:  ERR_DT
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
***************************************************************/

#include "im.h"
#include <math.h>


int     dt_response(double f_s, RATIONAL xfer_fcn, int num_freqs,
                    int rad_or_hz, double frequencies[],
                    COMPLEX response[])

    {
    double  t_s, w_t;
    int     i;

    t_s = 1.0 / f_s;

    for (i = 0; i < num_freqs; i++)
        {
        if (rad_or_hz == HZ)
            w_t = TWO_PI * frequencies[i] * t_s;
        else
            w_t = frequencies[i] * t_s;


        response[i] = eval_rational(cexpon(w_t), xfer_fcn);
        }

    return (NORMAL);
    }
```

```
/*****************************************************************
 *
 *  SOURCE FILE:    lp_to_hp.c
 *
 *
 *  FUNCTION:       lp_to_hp(lp, hp)
 *
 *
 *  DESCRIPTION:    Performs a lowpass-to-highpass transforma-
 *                  tion.
 *
 *
 *                  NOTES:
 *
 *                  1.  *hp  must be distinct from  *lp .
 *
 *                  2.  *lp  is assumed to be a valid rational
 *                      function (i.e., has been "made" prior
 *                      to invoking lp_to_bp() ).  No error-
 *                      checking for this condition is done.
 *
 *
 *  ARGUMENTS:
 *
 *      lp          (input)  RATIONAL *
 *                  Normalized lowpass transfer function to
 *                  be transformed.
 *
 *      hp          (output) RATIONAL *
 *                  Resulting highpass transfer function.
 *
 *
 *  RETURN:         int
 *                  NORMAL      :  Normal return.
 *                  ERR_LP_HP   :  An error has occurred.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   28 October 1989     Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
*****************************************************************/

#include "im.h"


int     lp_to_hp(RATIONAL *lp, RATIONAL *hp)

    {
    double  constant;
    int     hp_degree, n;


//  -----------------------------------------------------------
//  Initialization.
//  -----------------------------------------------------------
//  Make sure that  *lp  and  *hp  are distinct.
    if (lp == hp)
        return (ERR_LP_HP);
    hp_degree = MAX(lp->num.degree, lp->denom.degree);


//  -------------------------------------------------------------
//  "Make" a highpass transfer function of the proper size.
//  -------------------------------------------------------------
//  Assure that the HP transfer function is made from scratch.
//  It is assumed to have already been "made."
    free_rational(hp);

//  Now do the "make."
    if (make_rational(hp_degree, hp_degree, hp))
        return (ERR_LP_HP);


//  -------------------------------------------------------------
//  Compute the highpass coefficients.
//  -------------------------------------------------------------
    for (n = hp_degree; n >= 0; n--)
        {
        if (n >= hp_degree - lp->num.degree)
            hp->num.c[n] = lp->num.c[hp_degree - n];
        else
            hp->num.c[n] = 0.0;


        if (n >= hp_degree - lp->denom.degree)
            hp->denom.c[n] = lp->denom.c[hp_degree - n];
        else
            hp->denom.c[n] = 0.0;
        }


//  -------------------------------------------------------------
//   Normalization.
//  -------------------------------------------------------------
    constant = 1.0 / hp->num.c[hp->num.degree];

    for (n = 0; n <= hp_degree; n++)
        {
        hp->num.c[n] *= constant;
        hp->denom.c[n] *= constant;
        }


    return (NORMAL);
    }
```

```
/****************************************************************
 *
 *  SOURCE FILE:    lp_to_bp.c
 *
 *
 *  FUNCTION:       lp_to_bp(lp, bp, bp_w_c, bp_bw)
 *
 *
 *  DESCRIPTION:    Performs a lowpass-to-bandpass transforma-
 *                  tion.  This function normalizes the bandpass
 *                  coefficients such that the coefficient of
 *                  the highest-degree term in the numerator is
 *                  equal to unity.
 *
 *                  This function uses a recursion formula to
 *                  generate the transformation coefficients
 *                  c(n,r).
 *
 *                  NOTES:
 *
 *                  1.  *bp  must be distinct from  *lp .
 *
 *                  2.  *lp  is assumed to be a valid rational
 *                      function (i.e., has been "made" prior
 *                      to invoking lp_to_bp() ).  No error-
 *                      checking for this condition is done.
 *
 *                  3.  This function is based on the algorithm
 *                      presented in
 *
 *                      S.A. Dyer, "An improved algorithm for
 *                        low-pass to bandpass transformations,"
 *                        Proc. IEEE, vol. 76, no. 10,
 *                        pp. 1378-1379, 1988.
 *
 *
 *  ARGUMENTS:
 *
 *      lp          (input)  RATIONAL *
 *                  Normalized lowpass transfer function to
 *                  be transformed.
 *
 *      bp          (output) RATIONAL *
 *                  Resulting bandpass transfer function.
 *
 *      bp_w_c      (input)  double
 *                  Desired center frequency, in rad/sec, of
 *                  the bandpass transfer function.
 *
 *      bp_bw       (input)  double
 *                  Desired bandwidth, in rad/sec, of the
 *                  bandpass transfer function.
 *
 *
 *  RETURN:         int
 *                  NORMAL      :  Normal return.
 *                  ERR_LP_BP   :  An error has occurred.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 2.01
 *
 *
 *  REVISIONS:      None.
 *
 *
****************************************************************/

#include "im.h"
#include <math.h>


int     lp_to_bp(RATIONAL *lp, RATIONAL *bp, double bp_w_c,
                 double bp_bw)

    {
    D_MATRIX    c;
    double      alpha, beta, constant;
    int         k, max_deg, max_deg_2, n;


//  -----------------------------------------------------------------
//  Compute constants.
//  -----------------------------------------------------------------
    alpha = 1.0 / bp_bw;
    beta  = alpha * bp_w_c * bp_w_c;



//  -----------------------------------------------------------------
//  Initialization.
//  -----------------------------------------------------------------
//  Make sure that  *lp  and  *bp  are distinct.
    if (lp == bp)
        return (ERR_LP_BP);

    max_deg = MAX(lp->num.degree, lp->denom.degree);
    max_deg_2 = 2 * max_deg;


//  -----------------------------------------------------------------
//  "Make" a bandpass transfer function of the proper size.
//  -----------------------------------------------------------------
//  Assure that the BP transfer function is made from scratch.
//  It is assumed to have already been "made."
    free_rational(bp);

//  Now do the "make."
    if (make_rational(max_deg + lp->num.degree,
        max_deg + lp->denom.degree, bp))
        return (ERR_LP_BP);



//  -----------------------------------------------------------------
//  Compute the {c( , )}.
//  -----------------------------------------------------------------
    if (make_dmatrix(max_deg_2 + 1, max_deg + 1, &c))
        {
//      *bp does not contain a valid rational function.
        free_rational(bp);
        return (ERR_LP_BP);
        }


//  Compute the {c(k,0)}.
    for (k = 0; k <= max_deg_2; k++)
            c.a[k][0] = 0.0;

    c.a[max_deg][0] = 1.0;
//  Compute the {c(k,n)}.
    for (n = 1; n <= max_deg; n++)
        {
        c.a[0][n] = beta * c.a[1][n-1];

        for (k = 1; k < max_deg_2; k++)
            c.a[k][n] = alpha * c.a[k-1][n-1]  +
                beta * c.a[k+1][n-1];

        c.a[max_deg_2][n] = alpha * c.a[max_deg_2-1][n-1];
        }


//  ------------------------------------------------------------
//  Compute the bandpass coefficients.
//  ------------------------------------------------------------
    for (k = 0; k <= bp->num.degree; k++)
        {
        bp->num.c[k] = 0.0;
        for (n = 0; n <= lp->num.degree; n++)
            bp->num.c[k] += lp->num.c[n] * c.a[k][n];
        }



    for (k = 0; k <= bp->denom.degree; k++)
        {
        bp->denom.c[k] = 0.0;
        for (n = 0; n <= lp->denom.degree; n++)
            bp->denom.c[k] += lp->denom.c[n] * c.a[k][n];
        }


//  ------------------------------------------------------------
//  Normalization.
//  ------------------------------------------------------------
    constant = 1.0 / bp->num.c[bp->num.degree];

    for (k = 0; k <= bp->num.degree; k++)
        bp->num.c[k] *= constant;
    for (k = 0; k <= bp->denom.degree; k++)
        bp->denom.c[k] *= constant;


//  ------------------------------------------------------------
//  Prepare for return.
//  ------------------------------------------------------------
    if (free_dmatrix(&c))
        return (ERR_LP_BP);

    return (NORMAL);
    }
```

```
/*****************************************************************
 *
 *  SOURCE FILE:    lp_to_bs.c
 *
 *
 *  FUNCTION:       lp_to_bs(lp, bs, bs_w_c, bs_bw)
 *
 *
 *  DESCRIPTION:    Performs a lowpass-to-bandstop transforma-
 *                  tion.
 *
 *
 *                  NOTES:
 *
 *                  1.  *bs  must be distinct from  *lp .
 *
 *                  2.  *lp  is assumed to be a valid rational
 *                      function (i.e., has been "made" prior
 *                      to invoking lp_to_bp() ).  No error-
 *                      checking for this condition is done.
 *
 *                  3.  The transformation is accomplished by
 *                      performing a lowpass-to-highpass
 *                      transformation, followed by a lowpass-
 *                      to-bandpass transformation.
 *
 *
 *  ARGUMENTS:
 *
 *      lp          (input)  RATIONAL *
 *                  Normalized lowpass transfer function to
 *                  be transformed.
 *
 *      bs          (output) RATIONAL *
 *                  Resulting bandstop transfer function.
 *
 *      bs_w_c      (input)  double
 *                  Desired center frequency, in rad/sec, of
 *                  the bandstop transfer function.
 *
 *      bs_bw       (input)  double
 *                  Desired bandwidth, in rad/sec, of the
 *                  bandstop transfer function.
 *
 *
 *  RETURN:         int
 *                  NORMAL      :  Normal return.
 *                  ERR_LP_BS   :  An error has occurred.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   7 September 1999    Version 1.01
 *
 *
 *  REVISIONS:      None.
 *
 *
*****************************************************************/

#include "im.h"


int     lp_to_bs(RATIONAL *lp, RATIONAL *bs, double bs_w_c,
                 double bs_bw)
```

```
    {
    RATIONAL    hp;
    int         max_deg, rtn_code;


//  ------------------------------------------------------------
//  Initialization.
//  ------------------------------------------------------------
//  Make sure that  *lp  and  *bs  are distinct.
    if (lp == bs)
        return (ERR_LP_BS);

//  Make  hp .
    max_deg = MAX(lp->num.degree, lp->denom.degree);

    if (make_rational(max_deg, max_deg, &hp))
        return (ERR_LP_BS);


//  ------------------------------------------------------------
//  "Make" a bandstop transfer function of the proper size.
//  ------------------------------------------------------------
//  Assure that the BS transfer function is made from scratch.
//  It is assumed to have already been "made."
    free_rational(bs);

//  Now do the "make."
    if (make_rational(2 * max_deg, 2 * max_deg, bs))
        return (ERR_LP_BS);



//  ------------------------------------------------------------
//  Perform lowpass-to-bandstop transformation.
//  ------------------------------------------------------------
//  Perform lowpass-to-highpass transformation.
    if (lp_to_hp(lp, &hp))
        {
        free_rational(&hp);
        return (ERR_LP_BS);
        }


//  Perform highpass-to-bandstop transformation via
//   lp_to_bp() .  Normalization of the coefficients is done
//  within  lp_to_bp() .
    if (lp_to_bp(&hp, bs, bs_w_c, bs_bw))
        rtn_code = ERR_LP_BS;
    else
        rtn_code = NORMAL;


//  ------------------------------------------------------------
//  Prepare to return.
//  ------------------------------------------------------------
//  Free  hp .
    if (free_rational(&hp))
        rtn_code = ERR_LP_BS;


    return (rtn_code);
    }
```

```
/****************************************************************
 *
 *  SOURCE FILE:    gen_freqs.c
 *
 *
 *  FUNCTION:       gen_freqs(sweep, num_freqs, first_freq,
 *                      last_freq, frequencies)
 *
 *
 *  DESCRIPTION:    Fills the first  num_freqs  elements of the
 *                  array  frequencies  with values  first_freq
 *                  through  last_freq , spaced as follows:
 *
 *                  Linearly,        if  sweep = LINEAR_SWEEP
 *                  Logarithmically, if  sweep = LOG_SWEEP
 *
 *
 *  ARGUMENTS:
 *
 *      sweep       (input)  int
 *                  Desired spacing of frequency-values, as
 *                  follows:
 *
 *                  Linear:              sweep = LINEAR_SWEEP
 *                  Logarithmic:         sweep = LOG_SWEEP
 *
 *      num_freqs   (input)  int
 *                  Number of frequency-values to be generated.
 *
 *      begin_freq  (input)  double
 *                  Desired value of the first frequency.
 *
 *      last_freq   (input)  double
 *                  Desired value of the last frequency.
 *
 *      frequencies (output) double[]
 *                  Array of frequencies.
 *
 *
 *  RETURN:         int
 *
 *                  Return-code:
 *                      NORMAL:     Normal return.
 *
 *
 *  AUTHOR:         Copyright 1999
 *                  Stephen A. Dyer, Ph.D., P.E.
 *
 *
 *  DATE CREATED:   8 September 1999    Version 1.00
 *
 *
 *  REVISIONS:      None.
 *
 *
 ****************************************************************/

#include "im.h"
#include <math.h>


int     gen_freqs(int sweep, int num_freqs, double first_freq,
                  double last_freq, double frequencies[])

    {
    int     i;
    double  delta_f;
    if ((first_freq < 0.0) || (last_freq < 0.0) ||
        (last_freq <= first_freq))
        return(ERR_GENFREQS);

//  ------------------------------------------------------------
//  Logarithmically spaced frequencies:
//  ------------------------------------------------------------
    if (sweep == LOG_SWEEP)
        {
//      Temporarily store the logs of the frequencies.
        frequencies[0] = log(first_freq);
        frequencies[num_freqs - 1] = log(last_freq);

        delta_f = (frequencies[num_freqs - 1] - frequencies[0]) /
            (double)(num_freqs - 1);

        for (i = 1; i < num_freqs - 1; i++)
            frequencies[i] = frequencies[0] +
            (double)i * delta_f;

//      Now exponentiate to get the frequencies.
        frequencies[0] = first_freq;
        frequencies[num_freqs - 1] = last_freq;

        for (i = 1; i < num_freqs - 1; i++)
            frequencies[i] = exp(frequencies[i]);
        }

//  ------------------------------------------------------------
//  Linearly spaced frequencies (this is the DEFAULT):
//  ------------------------------------------------------------
    else
        {
        delta_f = (last_freq - first_freq) /
            (double)(num_freqs - 1);

        for (i = 0; i < num_freqs; i++)
            frequencies[i] = first_freq + (double)i * delta_f;
        }


    return (NORMAL);
    }
```