Numerical Methods for Identification of Induction Motor Parameters

by

Steven Robert Shaw

S.B. Massachusetts Institute of Technology (1995)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Electrical Engineer

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1997

Author .....
Department of Electrical Engineering and Computer Science
January 24, 1997

Certified by .....
Steven B. Leeb
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

Numerical Methods for Identification of Induction Motor Parameters

by

Steven Robert Shaw

Submitted to the Department of Electrical Engineering and Computer Science
on January 24, 1997, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Electrical Engineer

Abstract

This thesis presents two methods for determining the parameters of a lumped induction motor model given stator current and voltage measurements during a startup transient. The first method extrapolates a series of biased parameter estimates obtained from reduced order models to an unbiased estimate using rational functions. The second method uses part of the lumped parameter model as a rotor current estimator. The estimated rotor currents are used to identify the mechanical subsystem and to predict the rotor voltages. Errors in the predicted rotor voltages are minimized using standard non-linear least squares techniques. Both methods are demonstrated on simulated and measured induction motor transient data.

Thesis Supervisor: Steven B. Leeb
Title: Assistant Professor

Acknowledgments

I would like to thank Professor Steven Leeb for his support, guidance and patience. Professor Leeb's enthusiasm is both unwavering and inspiring.

Tektronix and Intel generously donated equipment essential to this work.

This project was supported by ORD/EPG and Lincoln Labs, ACC Project No. 182A administered by Marc Bernstein.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work describes numerical techniques for finding the lumped circuit model parameters of an induction motor from its electrical startup transient. This work is motivated by the desire to add diagnostic capabilities to the non-intrusive load monitor (NILM) developed in [12],[13],[14]. The NILM can detect the operation of electrical loads in a building by making measurements at the electric utility service entry of a building. By combining system identification capabilities with the transient recognition and acquisition ability of the NILM, a non-intrusive diagnostic system might be possible. The situation is indicated schematically in Figure 1-1.

The premise of non-intrusive diagnostics is that pending problems in electrical loads manifest themselves in the electrical startup transient detected by the NILM. If electrical transients could be interpreted in terms of parameters of a physical model, then the cause of the impending failure might be identified. For example, a cracking rotor bar in an induction motor would correspond to an increasing rotor resistance, which might be detected from the startup transient of the defective motor [5],[32]. As a first step in adding non-intrusive diagnostics to the NILM, this thesis considers the problem of identifying the parameters of an induction motor from an observed startup transient.

This thesis develops two methods for determining induction motor parameters. The first is an extrapolative technique employing reduced order models. The extrapolative method is philosophically similar to Richardson extrapolation or Stoer-Bulirsch integ-

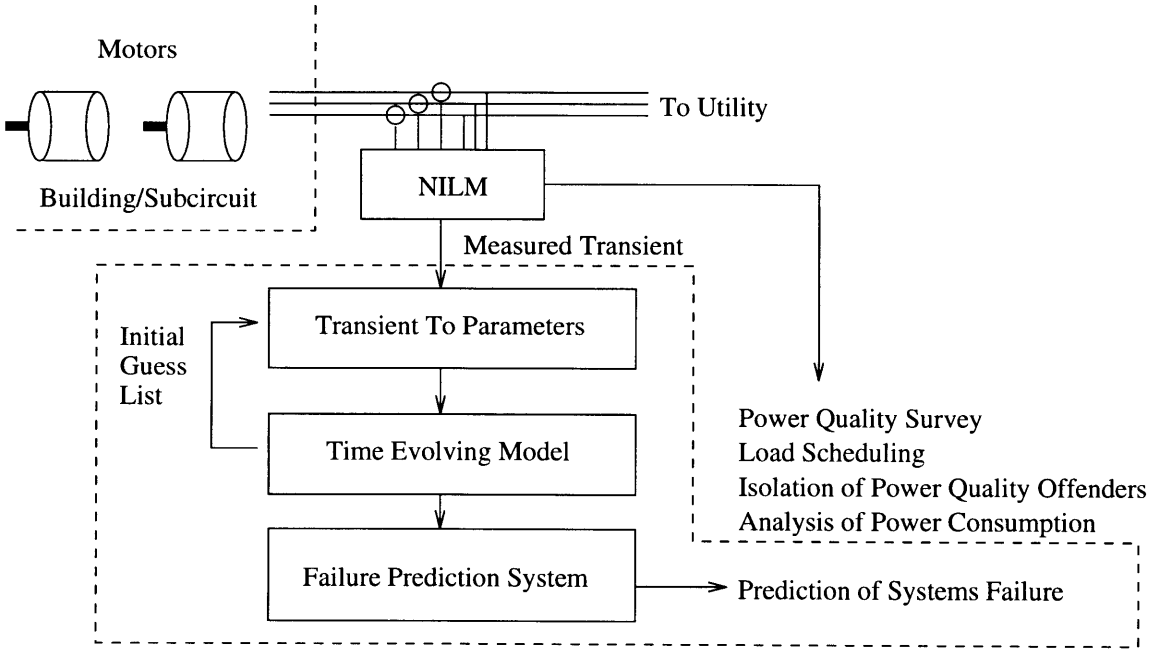

Figure 1-1: Context of the non-intrusive diagnostic system.

ration, in that the bias in a series of estimates is extrapolated to zero. The second method is a more classically oriented modified least-squares method using an observer extracted from the model. Both of these methods assume the lumped parameter induction motor model introduced by Krause in [11].

1.1 Thesis Outline

This chapter is an brief summary of some of the existing work relevant to induction motor identification and a review of some essential issues in system identification. The mathematical framework on which Chapters 3 and 4 are built is presented here, beginning with a practical discussion of least squares. Methods for solving discrete and continuous time system identification problems are discussed next, using examples. In Appendix A, the tools presented are put to work in a detailed example of power quality prediction based on modeling, identification, and simulation of the electric utility.

In Chapter 2, arbitrary reference frame transformations and the induction motor

model given in [11] are introduced. Alternate forms of the induction motor model, useful in the later chapters, are also presented. Simulation of the induction motor model is considered, and plots of simulated induction motor transients are given. Source codes for induction motor simulation are listed in Appendices B and C.

In Chapter 3, the extrapolative system identification procedure is developed as means of obtaining quick estimates of the parameters of a complicated system. Reduced order models are selectively applied to the data, and rational functions are used to extrapolate to approximately unbiased estimates. Source code for the extrapolative method can be found in Appendix D.

Chapter 4 develops a non-linear modified least-squares loss function for finding the parameters of the induction motor. Source listings for the method can be found in Appendix E. General purpose support routines, needed for both the extrapolative and least-squares methods can be found in Appendix F. In Chapter 5, parameter estimation results for the two techniques are evaluated and compared, using both real and simulated data. Data handling utilities and scripts used in Chapter 5 are in Appendix G. Finally, Chapter 6 concludes with a summary and analysis of the observations made.

1.2 Overview of related work

The following are summaries of some related work in the area of induction machine parameter and state estimation. This is by no means a comprehensive review of the literature. Rather, these summaries provide an overview of some of the perspectives adopted in induction motor estimation and fault detection.

In [5] and [4] the authors consider the application of estimation techniques to prediction of induction motor failure via rotor bar cracking. The induction motor model used is similar to Krause's steady state model [11]. A single phase of the motor is instrumented and identified, assuming balanced conditions. Single phase stator side electrical quantities are measured, and using a dynamometer, torque and speed measurements were made. The motor was also instrumented with thermocouples. Three

linear-least squares estimators are presented[1]. Of these estimators, one was found to be unstable, and the others were found to be incapable of satisfactorily estimating the parameter r_s. To work around the problem, r_s was measured directly. Ultimately, by taking thermal effects into account and by performing steady state experiments at a succession of operating speeds, the authors were able to detect changes in rotor resistance with sufficient accuracy to detect one broken rotor bar out of 45. In [5] the author concludes that "While R_r is easily estimated electrically, R_s is not." An accurate estimate of r_s was deemed necessary for the thermal compensation of r_r.

In [7] an estimation scheme is presented for determination of induction motor parameters. The model presumed is a linearized model using the stator currents and rotor fluxes as state variables. Except for the linearization, it appears to be equivalent to the model used in this thesis. In the paper, the 10 hp test induction motor is fully instrumented; measurements of stator currents, stator voltage, speed and torque are made. Torque excitation is available via a computer controlled DC motor connected to the induction motor. To identify parameters, the linearized machine model is manipulated into small signal transfer functions. The parameters of these transfer functions are obtained by exciting the induction motor with a pseudo-random binary torque signal generated by the DC motor. Measured stator current response to a torque step is compared to the simulated stator current response using parameters determined by the authors' methods, and "conventional test deduced parameters." It appears from the description of the torque source that the characterization of the motor and validation of the parameters was performed at one operating point.

In [29] a time-scale separation argument is used for simultaneous estimation of the slip and parameters. The argument is that the slip changes on a time scale much longer than the dynamics of the differential equations formed by the electrical subsystem. Using this observation, the slip can be regarded as "almost constant" and identified as if it were a parameter. Also, the electrical system can be regarded a approximately

[1]Although the estimators take the form $\hat{x} = (A^T A)^{-1} A^T b$, they do not satisfy the principle of least squares as stated in [28], i.e. the estimates do not minimize the errors in the observations in a least-squares sense.

LTI and identified as such. Of course, the success of this scheme requires that the recursive estimator converge on the parameters fast enough to track the slip. Although the slip estimation results for measured and simulated data in [29] are quite good, the parameter estimation did not perform as well. Note that the model presented in [29] is equivalent to the model used in this thesis, however, the assumption that the slip's derivative is negligible changes the character of the estimation problem dramatically. Essentially the same information is presented by the authors in [31].

In [30] "decomposed" algorithms are proposed for system identification. A decomposed algorithm proceeds in stages, minimizing the loss function over a sequence of subspaces which together span the parameter space. There is a clear analogy to relaxation methods for solving sparse linear systems. Recognizing this analogy, the properties of decomposed estimation using minimizations patterned on the classical relaxation methods are investigated. The methods are applied to on-line and batch parameter and speed estimation of induction machines. The model used in [30] is the same as is used in this thesis, although presented in a different form. However, in contrast to this thesis, a separation of electrical and mechanical time constants is assumed. Using the separation of time scales, the derivative of the slip is neglected, the electrical equations are "essentially LTI" and the slip is viewed as a slowly varying parameter. This treatment is identical to the treatment in [29], and according to the author, limits the potential usefulness of the algorithm to motors of under one hundred horsepower. Performance of the various proposed algorithms with the full parameter set was characterized by the author as unsatisfactory, and was attributed to ill-conditioning of the problem. To solve the conditioning problem, the author fixes the parameter r_s, reducing the dimension of the parameter space. With r_s fixed the various estimators showed good performance in estimating speed, and, depending on the estimator, one or two of the remaining free parameters. The bibliography is extensive.

In [16] speed and parameter estimation are considered. The model and parameterization are identical to the model used in [30]. The data from [16] appears in [31]. The same assumptions of time-scale separation are made throughout. The properties of time scale separation are exploited in the same way as in [29] and [30]. Similar

observations are made about the parameter r_s, i.e. that it is difficult to estimate. According, an estimator incorporating "slow stage" r_s estimation is considered. A comparison between measured/simulated currents and speed appears in the thesis, using parameters obtained from blocked rotor and no-load tests. Parameters from the no-load and blocked rotor tests, although they appear to be quite accurate from comparison of measured and simulated transients, were used as an initial guess for the estimation routines. Source code is included in the appendices.

In [32] the author attempts to establish the feasibility of using electrical stator measurements to detect broken rotor bars. Parameters were obtained from intact and damaged motors by standard tests; locked rotor, etc. Through a simulation study, the author determined that detection of broken rotor bars from spectral analysis of stator side current measurements was plausible. In actual tests with real motors, spectral information appeared to be dominated by other effects and rotor defects could not be detected. Source code in FORTRAN is included.

In [22] various observers for electromagnetic quantities, such as rotor flux, are given. The author presumes a model which is equivalent to the model used in this thesis. In Chapter 5 the author proposes for system identification the parameter set later used in [29], [30], [16] and [31]. The author correctly notes that the speed and machine papers might be estimated if the speed were essentially constant compared to both the convergence rate of the estimator and the electrical dynamics.

1.3 Methods for system identification

The problem of finding the parameters of an induction motor from its startup transient is a specific subset of the very general class of system identification problems. The subset is the class of identification problems where the input is deterministic, not controllable, and a model of the system is presumed. While alternative exist [10],[28],[20] the principle of least squares, stated below by Åström, is generally applicable to this type of problem:

> Postulate a mathematical model for the observations which gives them as a function of the unknown parameters θ. Choose the parameters θ such that the sum of squares of the differences between the observations and the model is as small as possible. [28]

Solution of linear and non-linear least squares problems, therefore, is considered first.

1.3.1 Linear least squares

Consider the system

$$Ax = b \tag{1.1}$$

where A has more rows than linearly independent columns. Generally there is no solution x, but perhaps there are x's that *almost* solve the system. For example, there exists

$$\bar{x} = \arg\min_x \|b - Ax\|^2 \tag{1.2}$$

where $\bar{x}$ is the "least-squares" solution. Note that the least squares solution is also the solution that minimizes the 2-norm of the residual $b - Ax$, as

$$\arg\min_x \|b - Ax\|^2 = \arg\min_x \|b - Ax\|. \tag{1.3}$$

There are other reasonable "solutions" to (1.1). For example,

$$\begin{aligned} x_\infty &= \arg\min_x \max_i |b_i - \sum_j A_{i,j}x_j| \\ &= \arg\min_x \|b - Ax\|_\infty \end{aligned} \tag{1.4}$$

is a solution that minimizes the maximum modulus of the components of the residual $b - Ax$. In general, the solution of (1.1) takes the form of a minimization over x,

$$x_V = \arg\min_x V(x) \tag{1.5}$$

dependent on the situation at hand. The solution given by (1.4), for example, might be preferable to the solution of (1.2) under certain circumstances.

Solutions x_V of the minimization problem (1.5) are not easy to compute, in general. However, in the case of least squares the solution $\bar{x}$ given in (1.2) can be found in closed form. At the minimum of the loss function, there is no perturbation in x that reduces the value of $V(x)$, i.e.

$$\nabla V(x) = 0. \tag{1.6}$$

Note that the loss function $V(x)$ is convex up everywhere; it has a zero divergence only at the minimum. Either by direct evaluation of the divergence of $V(x)$ or by the geometric interpretation that the error is orthogonal to the column space of A, it can be shown that for least squares solution of (1.1),

$$A^T(b - A\bar{x}) = 0. \tag{1.7}$$

Therefore,

$$A^T A\bar{x} = A^T b. \tag{1.8}$$

If the square and symmetric matrix $A^T A$ is invertible, then

$$\bar{x} = (A^T A)^{-1} A^T b. \tag{1.9}$$

The set of equations (1.8) is sometimes called the normal equations [27].

1.3.2 Weighted least squares

An examination of the statistical properties of the least squares procedure motivates the consideration of weighted least squares. Suppose that the β_i are a series of measurements corrupted by some independent, identically distributed zero-mean processes e_i so that $\beta_i = b_i + e_i$. For example, β might be the output of a noisy sensor measuring

some physical quantity b. If the linear least squares estimator

$$A^T A\hat{x} = A^T \beta \tag{1.10}$$

is used in an attempt to find x, it makes sense to ask what effect using the corrupted $\beta = b + e$ will have on the estimate $\hat{x}$. Equation (1.10) can be rewritten

$$\sum_{i=1}^{N} a_{j,i} a_{i,j} \hat{x}_j = \sum_{i=1}^{N} a_{j,i} \beta_i. \tag{1.11}$$

Taking the expectation,

$$\sum_{i=1}^{N} a_{j,i} a_{i,j} E(\hat{x}_j) = E(\sum_{i=1}^{N} a_{j,i} \beta_i), \tag{1.12}$$

using the property that $E(cX) = cE(X)$ if X is a random variable and c is a constant. Provided

$$E(\sum_{i=1}^{N} a_{j,i} \beta_i) = \sum_{i=1}^{N} E(a_{j,i} \beta_i), \tag{1.13}$$

then

$$\sum_{i=1}^{N} a_{j,i} a_{i,j} E(\hat{x}_j) = \sum_{i=1}^{N} a_{j,i} E(\beta_i), \tag{1.14}$$

since $E(\beta_i) = b_i$,

$$E(\hat{x}) = (A^T A)^{-1} A^T b \tag{1.15}$$

which is an unbiased estimate. The condition in (1.13) is satisfied, generally, in the limit that N is large and the columns of A are statistically independent of b [21],[10]. Therefore, if the columns of A are deterministic quantities like t, t^2, $\sin(t)$ etc, and the sequence b_i is corrupted with zero-mean white noise, least squares will produce an unbiased estimate.

The noise properties of least squares can be easily extended to two more general statistical situations. If the zero-mean disturbances e_i have individual variances

$$\sigma_i^2 = E(e_i^2) \tag{1.16}$$

then the unbiased "weighted least squares" estimate is

$$\hat{x} = (A^T R^{-1} A)^{-1} A^T R^{-1} b, \tag{1.17}$$

where $R_{i,j} = \delta_{i,j}\sigma_i^2$, and $\delta_{i,j}$ is the Kronecker delta. The estimator (1.17) is also unbiased in the more general case where R is the covariance matrix $E(e^T e)$. If the covariance matrix has off diagonal terms, (1.17) is the "BLUE" or best linear unbiased estimator [10]. BLUE is equivalent to minimization of the weighted loss function

$$V(x) = (b - Ax)^T R^{-1} (b - Ax). \tag{1.18}$$

1.3.3 Numerical methods for finding $(A^T A)^{-1}$

The normal equations (1.8) are not always easy to solve. The direct approach, computation and inversion of $A^T A$, is not favored because the condition number of $A^T A$ may be large. If A has condition number q then $A^T A$ has condition number q^2. The condition number q is defined as the ratio of the maximum eigenvalue to the minimum eigenvalue [27] which is also the ratio of the maximum singular value to the minimum singular value [19]. Also, in cases where $A^T A$ is a large matrix, it may be desirable to exploit its symmetry when solving. In the system identification context where the input is not controllable, the most likely contingency is that the excitation is insufficient to identify the parameters of the assumed model. In terms of the solution of (1.9) this implies that $A^T A$ will be badly conditioned, i.e. $A^T A$ may be rank deficient or very nearly rank deficient.

For reasons of simplicity, equation (1.9) was solved by LU decomposition and a few steps of iterative improvement in this thesis. However, it is quite likely that further attempts in this area will encounter the situation where a model, based on physics, has more parameters than can be found by examination of the transient. In cases where the ordinary LU decomposition solution technique does not work well, a very

stable method of solution is to compute the singular value decomposition

$$A = USV^T, \tag{1.19}$$

where U and V^T are orthogonal and S is the diagonal matrix of singular values [27]. Then an x is given by

$$x = VS^{-1}U^T b, \tag{1.20}$$

where the diagonal matrix S^{-1} is given by

$$S_{i,i}^{-1} = \begin{cases} 0 & \text{if } S_{i,i} \leq \epsilon \\ 1/S_{i,i} & \text{otherwise} \end{cases} \tag{1.21}$$

By choosing ϵ appropriately, singularities or near singularities in A can be eliminated [19]. If $\epsilon = 0$ and A is singular, the resulting x has minimal norm, in the L_2 sense. It follows that the singular value decomposition method of solution with $\epsilon = 0$ is equivalent to minimization of the loss function

$$V(x) = (b - Ax)^T(b - Ax) + \delta(\|b - Ax\|)x^T x \tag{1.22}$$

where

$$\delta(t) = \begin{cases} 1 & \text{if t} = 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.23}$$

is the unit impulse function. Discussion of the properties of the singular value decomposition can be found in [27]. C Code to compute the singular value decomposition can be found in [19], and a presentation at the algorithmic level can be found in [26] and [8].

1.3.4 Non-linear least squares

Linear least squares is concerned with problems of the form $Ax = b$. Non-linear least squares is applicable to problems of the form[2] $F(x) = b$. The loss function is then

$$\begin{aligned} V(x) &= (b - F(x))^T (b - F(x)) \\ &= \|b - F(x)\|^2 \end{aligned} \tag{1.24}$$

and the desired estimate is

$$\hat{x} = \arg\min_x \|b - F(x)\|^2. \tag{1.25}$$

Clearly, the solution of (1.25) is almost as difficult as the general non-linear minimization problem. There is no guarantee of a unique solution, depending on $F(x)$. However, minimization problems have such broad application that many routines are available [26], [19], [2]. There are also routines specifically intended for minimization of forms like (1.25) that are more efficient than application of a general purpose minimization routine to the loss function. The following is a simple method based on Newton's procedure that performs well with well-designed loss functions and good initial guesses [9].

Newton's method finds roots of the function $g : \Re \rightarrow \Re$ by a series of first order approximations. In particular, iteration of

$$\frac{\partial g(x_i)}{\partial x} \Delta = -g(x_i) \tag{1.26}$$

and

$$x_{i+1} = x_i + \Delta \tag{1.27}$$

sometimes converges on a root of g [19]. If Newton's method is applied to $V : \Re^M \rightarrow \Re^N$, we obtain the Gauss-Newton method. The individual components of the non-

[2]It should be noted that certain loss functions require minimizations similar in form to non-linear least squares even when the system identification problem is linear in the parameters [10].

linear least squares loss function are

$$v_i = (b_i - f_i(x))^2. \tag{1.28}$$

Applying Newton's method, the individual increment Δ is given by

$$2(b_i - f_i(x))\nabla f_i(x)\Delta = v_i. \tag{1.29}$$

Equivalently,

$$\nabla f_i(x)\Delta = \frac{b_i - f_i(x)}{2}. \tag{1.30}$$

The fortuitous cancelation of the term $b_i - f_i(x)$ is why it is less efficient to apply a general purpose minimization routine directly to the least-squares loss function. The cancelation reduces the curvature of the problem – the effective curvature in (1.30) is f, not f^2. Combining the individual increments into matrix form, Δ is determined by the Jacobian

$$J_{i,j} = \frac{\partial f_i(x)}{\partial x_j} \tag{1.31}$$

and

$$J\Delta = \frac{b - F(x)}{2}. \tag{1.32}$$

This over-determined system is linear in the increment Δ and has the least-squares solution

$$\Delta_i = (J^T J)^{-1} J^T \frac{b - F(x_i)}{2}. \tag{1.33}$$

If (1.33) is iterated with (1.27), a solution to the non-linear least squares problem may be found, provided that F is suitably well behaved. More advanced routines for solving the non-linear least squares problem primarily offer greater stability than this method. Newton's method has quadratic convergence near a minimum, but can behave poorly while approaching the minimum. For example, when the linearization effected by the Jacobian is bad, as it is when the loss function is evaluated at a local maximum, there is a tendency to make a huge step to an unreasonable set of parameters from which there is no recovery. Generally speaking, more advanced routines have a method

of switching between a slow, stable method far from the minimum and a fast, near quadratic method close to the minimum [2], [19]. This is the character of the method used in Chapter 4.

1.4 Discrete time identification

The mathematical formalism of least-squares can be applied directly to the identification of discrete time systems. The essence of the problem is to transform the difference equation describing the system so that it is expressed in the matrix form $Ax = b$, where x is the vector (or matrix) of desired parameters.

Consider the system

$$y[n] = \alpha w[n] + \beta. \tag{1.34}$$

This system is non-linear in the input-output sense, but supposing $w[n]$ and $y[n]$ are available, it is linear in the parameters. The input, system, and response can be put in matrix form;

$$\begin{pmatrix} w[1] & 1 \\ w[2] & 1 \\ \downarrow & \downarrow \\ w[n] & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} y[1] \\ y[2] \\ \downarrow \\ y[n] \end{pmatrix}. \tag{1.35}$$

Provided that $w[n]$ has sufficient richness, the parameters can be estimated. To distinguish between α and β in this case, $w[n]$ cannot be a constant. Identifying (1.35) as the form $Ax = b$, the least squares estimate $\hat{x}$ is given by (1.9).

The parameters of higher order difference equations are easily found by transforming the system appropriately. For example, suppose that a state space difference equation is prepared with some initial state $q[0]$. We would like to extract information about the system given the resulting transient in $q[k]$. Define B so that the $q[k]$ are given by the recursion

$$q[k+1] = Bq[k]. \tag{1.36}$$

Then the tableau to be solved is

$$\begin{pmatrix} q^T[1] \\ q^T[2] \\ \downarrow \\ q^T[n-1] \end{pmatrix} B^T = \begin{pmatrix} q^T[2] \\ q^T[3] \\ \downarrow \\ q^T[n] \end{pmatrix}. \tag{1.37}$$

If

$$A = \begin{pmatrix} q^T[1] \\ q^T[2] \\ \downarrow \\ q^T[n-1] \end{pmatrix} \tag{1.38}$$

and

$$C = \begin{pmatrix} q^T[2] \\ q^T[3] \\ \downarrow \\ q^T[n] \end{pmatrix} \tag{1.39}$$

then, as expected, a least squares estimate is given by

$$\hat{B}^T = (A^T A)^{-1} A^T C. \tag{1.40}$$

The technique is trivially extended to the case where there is an input $v[k]$

$$q[k+1] = Bq[k] + v[k]. \tag{1.41}$$

We need only redefine C

$$C = \begin{pmatrix} q^T[2] - v^T[1] \\ q^T[3] - v^T[2] \\ \downarrow \\ q^T[n] - v^T[n-1] \end{pmatrix}. \tag{1.42}$$

Then a least squares estimate for the evolution matrix B is then given by

$$\hat{B}^T = (A^T A)^{-1} A^T C. \tag{1.43}$$

1.5 Continuous time identification

The continuous time analog to the difference equation $x[k+1] = Ax[k]$ is

$$\frac{dx(t)}{dt} = Ax(t). \tag{1.44}$$

The usual situation is that samples $x[n] = x(nT)$ are available for analysis. Assuming that sampling rate considerations are met and that the signal is properly sampled, the relationship between the $x[n]$ is

$$x[n+1] = x[n]e^{AT}. \tag{1.45}$$

Equation (1.45) reveals a fundamental problem; the relationship between the available data points is non-linear in the parameters. Short of solving the non-linear problem directly, there are a number of techniques for identifying the continuous time system. These techniques are given primarily to demonstrate the attractiveness of the operator transformation technique presented last and used in Chapters 3 and 4.

1.5.1 Identification of an RC Circuit from samples of the step response

The RC circuit in Figure (1-2) is a good example for continuous time system identification. It is interesting to note, however, that even the RC circuit can be intractable. For example, consider the situation where some voltage excitation is introduced on the left hand port, and the voltage on the right hand port is measured. The system parameters R and C cannot be individually determined. The character of the system identification problem is completely dependent on the data available.

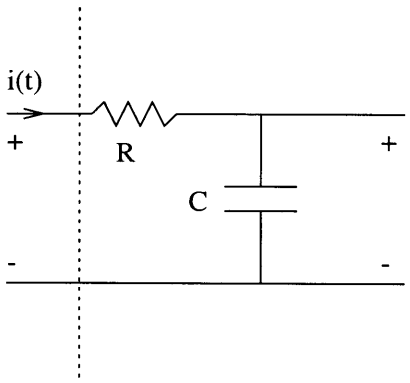

Figure 1-2: RC circuit for system identification

To parallel the motor identification problem, assume that the RC circuit is stimulated by a step in voltage on the left hand port and that the current entering the left hand port is measured. From circuit theory, the current is

$$i(t) = \frac{V}{R} e^{-t/RC} u(t). \tag{1.46}$$

The goal is to fit a solution of the form of (1.46) to the observed data. The preferred method, like least squares applied to a discrete time system, would be well behaved and have known properties with respect to noise in the data. Equation (1.46) is not, however, linear in time. There are at least three ways to pose the problem of fitting equation (1.46) to the observed data.

Normal equations from estimation of the integral or derivative

One technique for finding the parameters is to express the system as a integral or differential equation, and then to estimate the required integrals or derivatives from the data. For example,

$$v(t) = Ri(t) + \frac{1}{C} \int_0^t i(\tau) d\tau. \tag{1.47}$$

Alternatively, one could estimate the derivative of the current to obtain the time constant RC and then solve for R in a second step. Both of these strategies are restricted by the fact that the data is available at discrete times only. The integral or derivative

must be estimated by some kind of finite difference formula. Some of these are given below.

Forward Euler:

$$g(t+T) = g(t) + T\dot{g}(t) \tag{1.48}$$

Backward Euler:

$$g(t+T) = g(t) + T\dot{g}(t+T) \tag{1.49}$$

Trapezoidal Rule:

$$g(t+T) = g(t) + \frac{T}{2}(\dot{g}(t) + \dot{g}(t+T)) \tag{1.50}$$

There are numerous other forms [19]. All of these forms can be found by manipulation of a truncated Taylor series. For example, the first order methods are derived by ignoring the second order and higher terms in a Taylor series of a presumed e^{at} solution. For example, $x(t+T) = e^{AT}x(t)$ is represented by substitution of the truncated Taylor series $x(t+T) = (I+AT)x(t)$. Since $Ax = \dot{x}$ by definition, $x(t+T) = x(t)+T\dot{x}(t)$. These finite difference formulae also have frequency domain counterparts; for example, "impulse invariance" amounts to a first order integration method, and the bilinear or Möbius transform is equivalent mathematically to the trapezoidal rule. Clearly, rejection of higher order terms introduces error. Any system with an infinitely differentiable continuous time output, like e^{at}, cannot be precisely simulated with a method that neglects higher order derivatives. These problems can be addressed in a number of ways. One is to interpolate and upsample the signal, making the effective time increment smaller. This improves things because the whole Taylor series is effectively evaluated closer to the point of expansion.

Linearization of $e^{t/\tau}$ by Logarithm

Another means of solving for the parameters of the RC circuit is to use the logarithm to linearize the RC response. Consider this method for a discretized version of equation (1.46), equation (1.51). Equation (1.51) is simply equation (1.46) evaluated at discrete

points in time nT.

$$i[n] = \frac{V}{R}(e^{T/RC})^n u[n] \tag{1.51}$$

Taking the logarithm will "linearize" (1.51) with respect to n

$$\log(i[n]) = \log(\frac{V}{R}) + \frac{T}{RC}n. \tag{1.52}$$

To solve (1.52) given a set of sampled data, the following over-determined system can be written.

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ \downarrow & \downarrow \\ 1 & n-1 \end{pmatrix} \begin{pmatrix} \log(\frac{V}{R}) \\ \frac{T}{RC} \end{pmatrix} = \begin{pmatrix} i[0] \\ i[1] \\ \downarrow \\ i[n] \end{pmatrix} \tag{1.53}$$

Equation (1.53) can be solved by the method of normal equations.

The logarithm method has a certain elegance, but there are still problems. In particular, the noise performance is not obvious. If the noise is white and there are many samples, the estimates should be unbiased since the regressors are uncorrelated to the noise. However, any error incurred while solving the normal equations will appear in an *exponential* way in terms of the parameter estimates.

The "logarithm" technique can be extended to higher order systems. Consider a state-space continuous time system of the form

$$\frac{d}{dt}x = Ax. \tag{1.54}$$

Samples $x[k] = x(kT)$ are described by

$$x[k+1] = Bx[k], \tag{1.55}$$

where

$$B = e^{AT}. \tag{1.56}$$

Using discrete time techniques, an estimate $\hat{B}$ could be found from the samples $x[k]$.

The problem is then to evaluate the "matrix logarithm" of (1.56) to find an estimate of A from $\hat{B}$. Assuming that A and B are diagonalizable, let A and B have the following eigenvalue decompositions:

$$\begin{aligned} A &= S\Lambda S^{-1} \\ B &= R\Gamma R^{-1}. \end{aligned} \tag{1.57}$$

In Equation (1.57) Γ and Λ are diagonal matrices of eigenvalues and S and R are matrices of eigenvectors [27], [17]. Referring to Equation (1.56), B can be expanded according to the definition of the matrix exponential [27]:

$$B = I + AT + \frac{(AT)^2}{2} + \cdots + \frac{(AT)^N}{N!} + \cdots. \tag{1.58}$$

Since $[A, I] = AI - IA = 0$ and $[A, A] = 0$ according to the basic properties of the commutator [6], it is obvious from the series expansion of B in (1.58) that

$$[A, B] = 0. \tag{1.59}$$

A consequence of (1.59) is, according to Theorem 5F in [27], that A and B share the same eigenvector matrix S. Thus, (1.57) may be rewritten

$$\begin{aligned} A &= S\Lambda S^{-1} \\ B &= S\Gamma S^{-1}. \end{aligned} \tag{1.60}$$

Since

$$\begin{aligned} B &= e^{AT} \\ &= Se^{\Lambda T}S^{-1}, \end{aligned} \tag{1.61}$$

it follows that

$$\Gamma = e^{\Lambda T}. \tag{1.62}$$

Since Γ and Λ are diagonal, the logarithm can be applied directly, i.e.

$$\Lambda_{i,i} = \frac{\log \Gamma_{i,i}}{T}. \tag{1.63}$$

Therefore,

$$A = S \begin{pmatrix} \frac{\log \Gamma_{1,1}}{T} & & & \\ & \frac{\log \Gamma_{2,2}}{T} & & \\ & & \ddots & \\ & & & \frac{\log \Gamma_{n,n}}{T} \end{pmatrix} S^{-1}. \tag{1.64}$$

Note that (1.64) is not unexpected. S is a similarity transform that decouples (1.54). In the decoupled basis the state space system acts like N independent, non-interacting first order systems. There are, however, many contingencies which would have to be addressed in implementation. For example, the diagonalizability of the matrix B is presumed – this could be problematic, particularly since B is an estimate. This analysis merely supports the plausibility and indicates the complexity of applying the logarithm method to state space systems. A numerical discussion of the eigenvalue problem can be found in [26].

Operator transformation for RC system identification

One way of avoiding the noise and other difficulties associated with estimating the derivative of a continuous time signal is to replace a differential equation model with a model expressed in terms of some other operator. Ideally, this substitute operator would be easy to compute. Consider the causal, "low-pass" operator λ, with $\tau > 0$, [10]:

$$\lambda = \frac{1}{1 + p\tau}. \tag{1.65}$$

To eliminate the derivatives in a differential equation model, we isolate $p = \frac{d}{dt}$ in (1.65):

$$p = \frac{1 - \lambda}{\lambda \tau}. \tag{1.66}$$

Substituting p in the RC system relation

$$(R + \frac{1}{Cp})i(t) = v(t) \tag{1.67}$$

yields, with some manipulation,

$$(RC(1-\lambda) + \lambda\tau)i(t) = C(1-\lambda)v(t). \tag{1.68}$$

A set of normal equations in this new operator can be trivially derived.

$$\begin{pmatrix} (1-\lambda)i[1] & \lambda\tau i[1] \\ (1-\lambda)i[2] & \lambda\tau i[2] \\ \downarrow & \downarrow \\ (1-\lambda)i[N] & \lambda\tau i[N] \end{pmatrix} \begin{pmatrix} R \\ \frac{1}{C} \end{pmatrix} = \begin{pmatrix} (1-\lambda)v[1] \\ (1-\lambda)v[2] \\ \downarrow \\ (1-\lambda)v[N] \end{pmatrix} \tag{1.69}$$

It should be noted that transformation of the differential equation model to a model expressed in terms of the λ operator does not eliminate the truncation problem associated with finite difference approximations to the derivative. The action of λ on the observed quantities must be computed, which requires a finite difference scheme of some kind. Even if λ is computed by the FFT, the various methods of mapping the continuous time transfer function to a discrete time transfer function are equivalent to various approximations of the derivative by finite difference methods. For example, creating a discrete time λ by application of the bilinear transform and applying this discrete time λ via the FFT is equivalent to integrating using the trapezoidal rule[3]. There is an advantage, however. The derivative that must be approximated when applying λ is the derivative of the *output* of the operator, as opposed to the derivative of the noisy observations. In effect, the sensitivity of the terms involving λ to the approximation involved in computing the derivative can be determined by selection of τ. Also, it is often possible to arrange the system identification problem so that

[3]The equivalence of the integration methods and continuous to discrete time transforms is true in an analytical sense. However, the properties of error propagation and the ease with which initial conditions can be constrained are quite different.

the regression matrix consists only of filtered observations and the right hand side contains all the "noisy" unfiltered observations. Under these conditions, it might be argued that the regressors would be substantially uncorrelated to the right hand side, producing unbiased estimates.

1.5.2 Computation of λ

The λ operator makes the continuous time identification problem relatively straightforward. The penalty for simple analysis of the transformed system is the computation of λ.

One attractive possibility for computing λ is the "hybrid" scheme suggested in [10]. Rolf observes that for signals that can be represented by linear combinations of exponentials e^{st}, the λ operator (1.65) is precisely the transfer function of an RC circuit

$$\frac{V_{out}(s)}{V_{in}(s)} = \frac{1}{RCs+1}. \tag{1.70}$$

Assuming that the continuous time analog waveforms were available, the response of a precisely calibrated RC circuit to these waveforms could be sampled. The typical situation, however, is that only the sampled waveforms are available.

In standard references, e.g. [18], techniques are given for implementing IIR filters like λ. The main step is to select a mapping from continuous to discrete time. This mapping can be specified in the time domain via a finite difference approximation for the continuous time derivative like

$$\frac{dx}{dt} \approx \frac{x[k+1]-x[k]}{T} \tag{1.71}$$

or in the frequency domain via a mapping of continuous time frequencies Ω to discrete time frequencies ω, i.e.

$$\omega = T\Omega. \tag{1.72}$$

Alternatively, one can select a mapping that is based not on a transformation of the model but rather on some aspect of its response. For example, define a discrete time

transfer function such that the response of the filter to some important signal (like a step) is conserved.

The mapping used to translate the continuous time filter to a discrete time system suggests the method used to actually apply the discrete time filter to the data. The filter transformed with the time-domain mapping (1.71) is a difference equation that can be iterated, while the filter resulting from (1.72) is most conveniently implemented using the DFT. In the particular case of the λ operator, a reasonably accurate implementation via the DFT requires more operations and storage than the difference equation approach. The module `lambda.cc` in Appendix F implements the lambda operator using a variable step size equivalent of the finite difference equation.

1.6 Summary

This chapter presents mathematical techniques for the solution of some least-squares problems often encountered in system identification. This mathematical background is essential to Chapters 3 and 4. The principle tools reviewed in this chapter include solution of

- the over-determined linear least squares problem
- the over-determined non-linear least squares problem
- under-determined or badly conditioned problem via SVD

and application to discrete and continuous time system identification problems. A broad range of system identification problems are susceptible to the techniques of this chapter, as illustrated by the example of power quality prediction given in Appendix A. Note that all the mathematical results of this chapter could be expressed in the form of a minimization of a loss function over the parameter space. For reasons of computational simplicity, however, explicit minimization of the loss function using numerical techniques is generally avoided if possible.

Chapter 2

Induction Motor Model

The three phase induction motor model introduced in this chapter is the lumped parameter model given in [11]. Transformations to an arbitrary, rotating frame are introduced to interpret the induction motor model. The model is expressed in the synchronously rotating "dq" frame. Finally, simulation of the induction motor is considered, and it is seen that simulation is most efficiently accomplished using flux linkages as state variables rather than currents.

2.1 Arbitrary reference frame transformations

The stator windings in an induction motor are arranged so that applied three phase currents form a rotating magnetic field. This rotating field induces currents in the rotor, which usually rotates at a lesser angular velocity. The analysis of machinery of this sort, where there are rotating fields, structures and circuits, is greatly simplified by the introduction of a transform that can take sets of variables from the fixed laboratory frame to an arbitrary rotating frame. Transformations of this type are sometimes called Parks transformations [11]. The transformation to an arbitrary reference frame at angle $\beta(t)$ is

$$K = \frac{2}{3}\begin{pmatrix} \cos\beta & \cos(\beta - \frac{2\pi}{3}) & \cos(\beta + \frac{2\pi}{3}) \\ \sin\beta & \sin(\beta - \frac{2\pi}{3}) & \sin(\beta + \frac{2\pi}{3}) \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{pmatrix}. \tag{2.1}$$

The inverse transformation is

$$K^{-1} = \begin{pmatrix} \cos\beta & \sin\beta & 1 \\ \cos(\beta - \frac{2\pi}{3}) & \sin(\beta - \frac{2\pi}{3}) & 1 \\ \cos(\beta + \frac{2\pi}{3}) & \sin(\beta + \frac{2\pi}{3}) & 1 \end{pmatrix}. \tag{2.2}$$

Note that the transform and its inverse are time dependent through $\beta(t)$. A particular and important example is the transformation to the synchronously rotating frame. Here

$$\beta = \omega t \tag{2.3}$$

where ω is the base electrical frequency. Typically, $\omega = 2\pi 60$ rad/s. Equation (2.1) taken with (2.3) define a transformation to a frame that rotates synchronously with three phase sources in the laboratory frame. For example, the lab frame three phase voltage source given by

$$v_{abc} = V_0 \begin{pmatrix} \cos(\omega t) \\ \cos(\omega t - \frac{2\pi}{3}) \\ \cos(\omega t + \frac{2\pi}{3}) \end{pmatrix} u(t) \tag{2.4}$$

is, in the synchronously rotating frame,

$$v_{dq0} = V_0 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} u(t) \tag{2.5}$$

according to the transformation

$$v_{dq0} = K v_{abc}. \tag{2.6}$$

This is an important simplification of the drive typically applied to an induction motor. The synchronously rotating frame is often referred to as the "dq" frame. Note that under balanced conditions, where

$$i_a + i_b + i_c = 0 \tag{2.7}$$

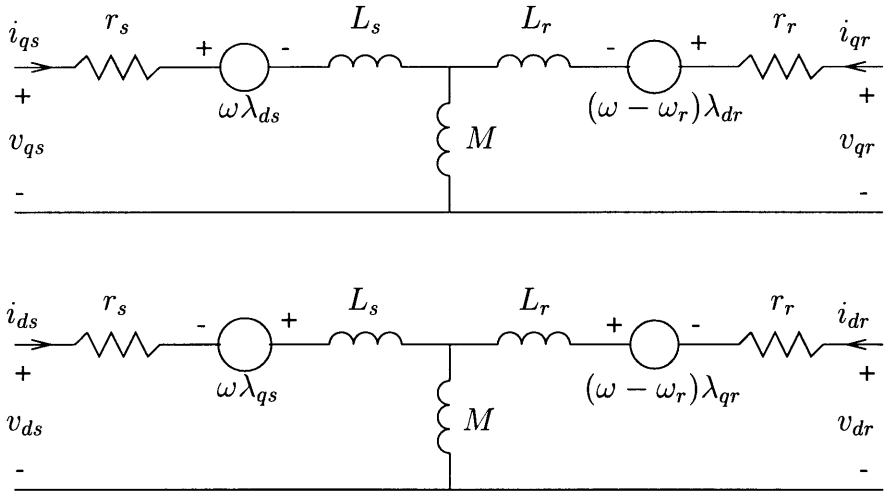

Figure 2-1: Induction motor circuit model.

and

$$v_a + v_b + v_c = 0 \tag{2.8}$$

only the currents i_q and i_d (v_q and v_d) need be specified. In the following work, balanced conditions are assumed. Other important frames include the laboratory frame, where β is constant, and the frame fixed in the rotor, where

$$\beta(t) = \int_0^t \omega_r(\tau)d\tau. \tag{2.9}$$

Codes to calculate the arbitrary reference frame transformations, including codes to translate data in files, can be found in Appendix G.

2.2 Model of Induction Motor (After Krause)

The formulation of the induction motor used here is the same as is used by Krause. A three-phase, balanced machine is assumed, i.e., (2.7) and (2.8) hold. Figure 2-1 (after Krause, Figure 4.5-1) shows the lumped-parameter model used. The variables indicated are in the synchronously rotating dq frame. Since the T network of the rotor and stator leakage inductances L and the magnetizing inductance M form a cut set,

no attempt is made to determine rotor and stator leakages independently. Rather, L_r is presumed equal to L_s. Also, in the equations that follow, the inductances appear as impedances at the base electrical frequency of 60 Hz (377 rad/s). For example,

$$X_m = \omega M = 120\pi \cdot M. \tag{2.10}$$

The equations that encapsulate the lumped parameter model above are given in terms of the dq currents and voltages in the following matrix form

$$\begin{pmatrix} r_s + X_{rr}\frac{p}{\omega} & X_{rr} & X_m\frac{p}{\omega} & X_m \\ -X_{rr} & r_s + X_{rr}\frac{p}{\omega} & -X_m & X_m\frac{p}{\omega} \\ X_m\frac{p}{\omega} & sX_m & r_r + X_{rr}\frac{p}{\omega} & sX_{rr} \\ -sX_m & X_m\frac{p}{\omega} & -sX_{rr} & r_r + X_{rr}\frac{p}{\omega} \end{pmatrix} \begin{pmatrix} i_{qs} \\ i_{ds} \\ i_{qr} \\ i_{dr} \end{pmatrix} = \begin{pmatrix} v_{qs} \\ v_{ds} \\ v_{qr} \\ v_{dr} \end{pmatrix}. \tag{2.11}$$

In (2.11) $X_{rr} = X_m + X_l$ and p is the operator $\frac{d}{dt}$. It should be noted that while the stator currents can be associated with the physical currents in the wires coming out of a motor, the rotor currents are not as easy to localize. The rotor currents in the model above are expressed in the synchronously rotating frame, which is not the same frame as the rotor itself. Furthermore, it is often impossible to make connections of any kind to the "rotor circuits." This is because the rotor moves and because the conducting material is often cast aluminum, not individual wires.

The mechanical part of the induction motor equations involve the reaction of the rotor and mechanical load to the electrical torque induced by the currents above. The rotation of the rotor enters the electrical dynamics through the slip s in (2.11). The currents affect the mechanical system through the torque of electric origin, which is given in [11] as

$$T_e = \frac{3}{2}\frac{P}{2}M(i_{qs}i_{dr} - i_{ds}i_{qr}). \tag{2.12}$$

Here, P is the number of poles, M is the magnetizing inductance (not X_m), and the rotor currents are as reflected to the stator. From basic mechanics, the action of a torque τ is to produce an angular acceleration $\frac{d\omega}{dt}$. The slip, which enters directly into

the above equations, is a normalized measure of the rotor's angular velocity ω_r.

$$s = \frac{\omega_s - \omega_r}{\omega_s} \tag{2.13}$$

Here ω_s is the synchronous angular frequency, which corresponds to the rotational frequency of the MMF wave induced by the stator.

Since the only interaction between the electrical system and the mechanical system is through the slip s, it is advantageous to "recast" the mechanical parameters. The reaction torque of a moment of inertia J and a friction B is

$$T_m = J\frac{d\omega}{dt} - B\omega. \tag{2.14}$$

Since

$$\frac{ds}{dt} = -\frac{1}{\omega_s}\frac{d\omega_r}{dt}, \tag{2.15}$$

it follows that for an induction machine loaded by an inertia and damping only,

$$\frac{ds}{dt} = \gamma(i_{ds}i_{qr} - i_{qs}i_{dr}) + \beta(1 - s), \tag{2.16}$$

where γ and β absorb the inertia and other parameters from equations (2.12) to (2.14). In particular, note that if the slip is used as a mechanical state variable, the number of poles can be discarded.

Clearly, it is necessary to limit the complexity of the mechanical load model for identification purposes. Otherwise, one could imagine a contrived mechanical load that could create an almost arbitrary signal at the electrical terminals. For the purposes of this thesis, the electro-mechanical interaction will be limited to (2.16).

2.3 Induction Machine Equations in Complex Variables

The symmetry in the induction machine equations can be exploited to obtain an expression equivalent to but more compact than (2.11) using complex variables. This is accomplished with the following definitions:

$$i_s = i_{qs} + j i_{ds} \tag{2.17}$$

$$i_r = i_{qr} + j i_{dr} \tag{2.18}$$

$$v_s = v_{qs} + j v_{ds} \tag{2.19}$$

$$v_r = v_{qr} + j v_{dr} \tag{2.20}$$

where $j = \sqrt{-1}$.

The induction motor model (2.11) can then be rewritten as

$$\begin{pmatrix} v_s \\ v_r \end{pmatrix} = \begin{pmatrix} r_s + (X_m + X_l)(\frac{p}{\omega} - j) & X_m(\frac{p}{\omega} - j) \\ (X_m)(\frac{p}{\omega} - sj) & r_r + (X_m + X_l)(\frac{p}{\omega} - sj) \end{pmatrix} \begin{pmatrix} i_s \\ i_r \end{pmatrix}. \tag{2.21}$$

The economization of notation achieved with complex variables is extremely helpful. In the routines in Chapter 4, the currents are actually stored as complex pairs because much of the calculation takes advantage of the complex fast Fourier transform.

2.4 Simulation of Induction Motor Model

An expression providing the derivative of the state as a function of the state and inputs is necessary to simulate the induction motor. Direct use of (2.11) or (2.21) is not very efficient because a matrix must be inverted at each time step to find the derivatives. The matrix inversion can be avoided by using a different set of state variables. Define the stator and rotor flux linkages per second as

$$\Psi_s = (X_l + X_m) i_s + X_m i_r \tag{2.22}$$

$$\Psi_r = (X_l + X_m)i_r + X_m i_s. \tag{2.23}$$

Using the new state variables Ψ_s and Ψ_r the induction motor model can be expressed in complex form as

$$v_s = r_s i_s + (\frac{p}{\omega} - j)\Psi_s \tag{2.24}$$

$$v_r = r_r i_r + (\frac{p}{\omega} - sj)\Psi_r. \tag{2.25}$$

Note that the required derivative appears in (2.25) in a simple way. However, it is necessary to find i_s and i_r at each step from the evolved states Ψ_s and Ψ_r. In practice, simulations of the induction motor using currents as state variables and flux linkages per second as state variables were found to yield identical results. The simulations using flux linkages were somewhat faster, as expected.

Using the parameters $X_m = 26.13\Omega$, $X_l = .754\Omega$, $r_r = .816\Omega$, $r_s = .435\Omega$ and an inertial load $J = .089\text{kg m}^2$ with an excitation of 220 V line to line, the results in Figures 2-2 through 2-7 were obtained by simulating (2.11). The parameters above describe a 3 hp rated motor and can be found in [11].

The simulation code, listed in Appendix C, is a fifth order Runge Kutta method with monitoring of the local truncation error using the fourth order embedded method, as described in [19]. Local truncation error estimates are used to control the step-size and bound the errors. This code is substantially derived from *Numerical Recipes in C*, although adopted for use with the C++ matrix and vector handling routines used for this work. Appendix B contains a similar, general purpose simulation code written in C.

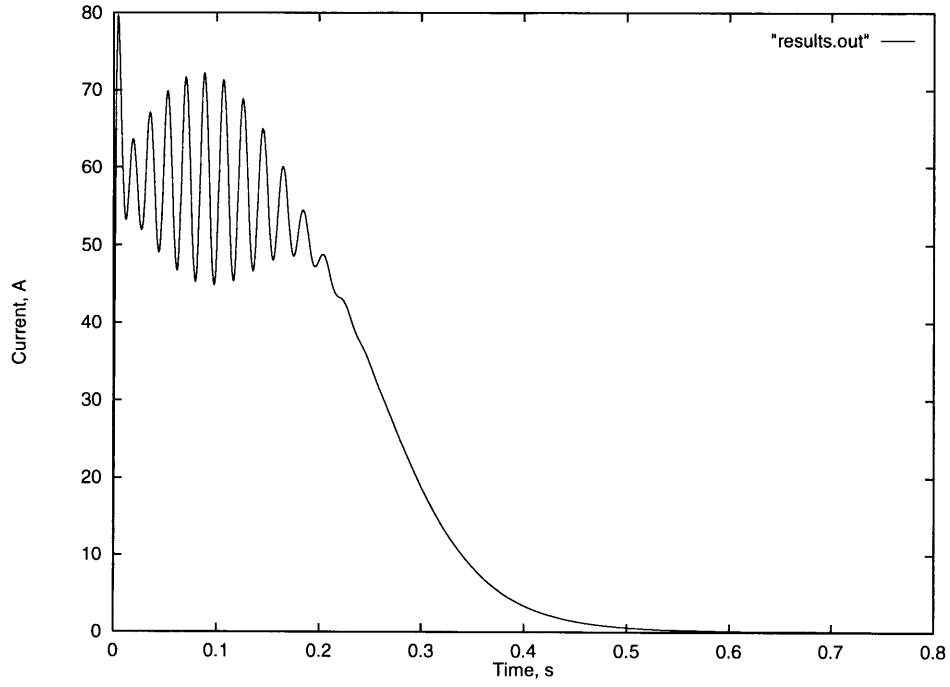

Figure 2-2: Motor starting transient, i_{qs}

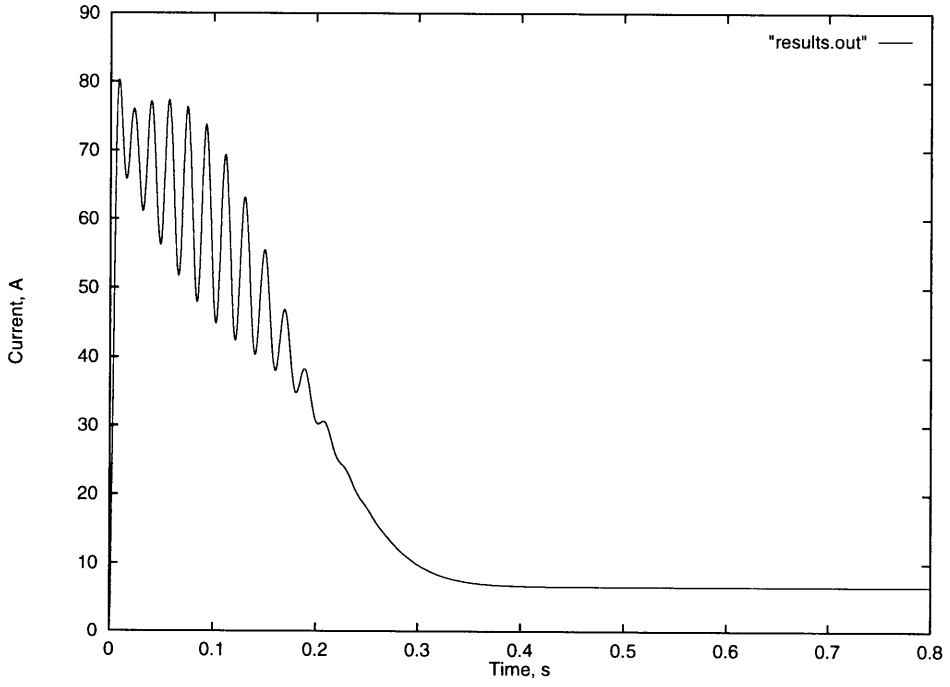

Figure 2-3: Motor starting transient, i_{ds}

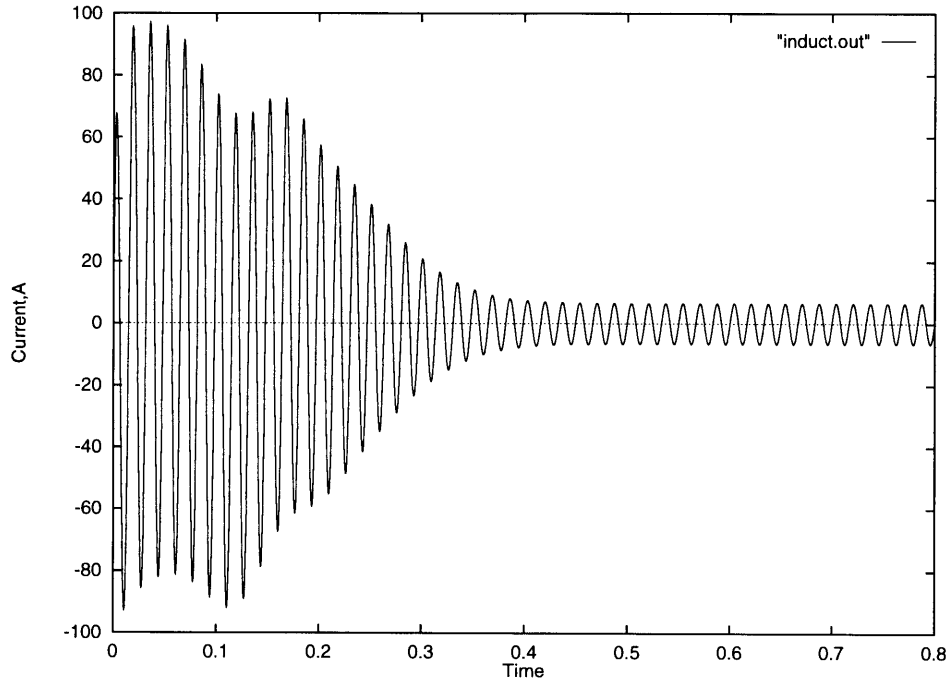

Figure 2-4: Motor starting transient, i_a

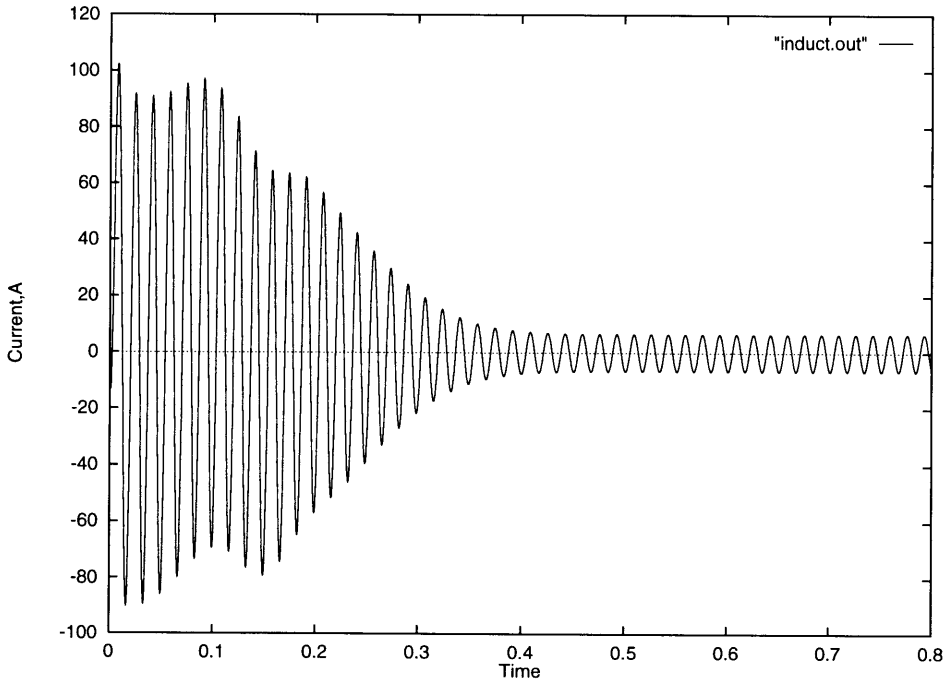

Figure 2-5: Motor starting transient, i_b

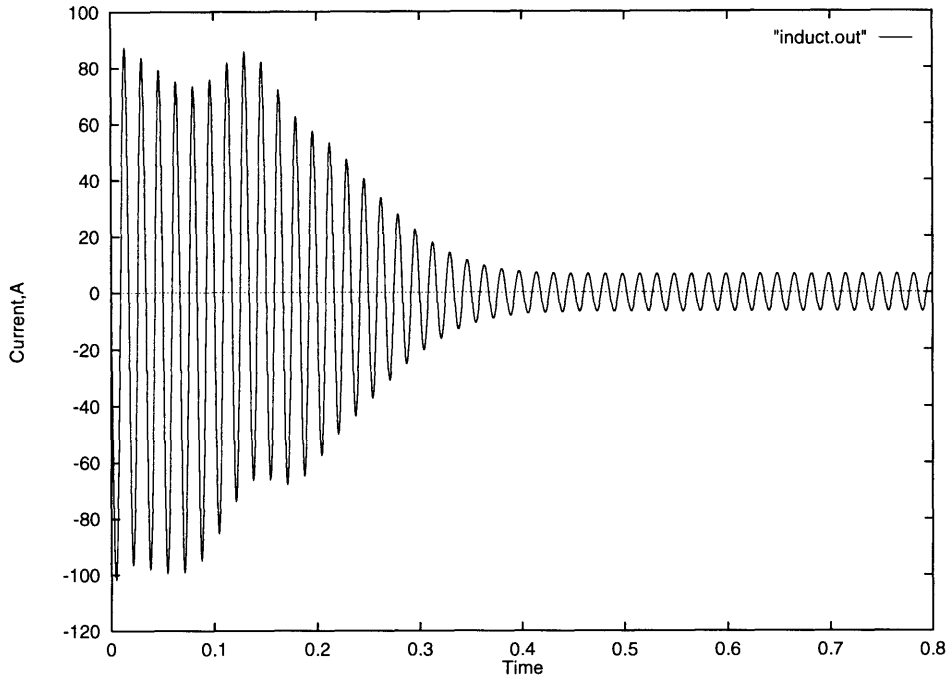

Figure 2-6: Motor starting transient, i_c

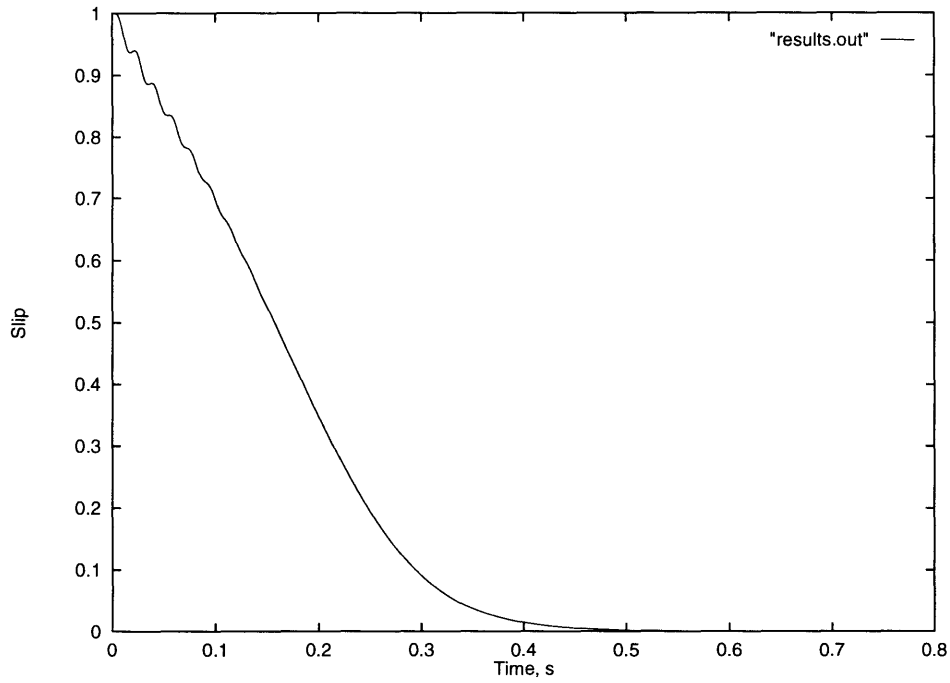

Figure 2-7: Motor starting transient, s

Chapter 3

Extrapolative System Identification

The extrapolative method developed here is motivated by the idea of quickly "eye-balling" data to obtain reasonably accurate parameter estimates. The parameter estimates obtained using this method are not likely to be least-squares solutions, but might be sufficiently accurate for some applications. In situations where a specific error criteria such as least squares must be minimized by an iterative routine, the methods presented here could dramatically increase computational efficiency by supplying a good initial guess.

The philosophy of the method is to decompose a transient described by a complicated model into smaller domains described by simple, easy to identify models. The desired parameters are obtained using standard system identification techniques for these simple models in their respective regions of validity. This situation is illustrated in Figure 3-1. The trajectory ξ is a transient in phase-space. The trajectory is entirely contained in a domain for which the full model M is valid. Instead of attempting to perform system identification given the complicated model M, regions are identified where the trajectory intersects the domains of simpler models μ_k that are easy to identify. The simpler models μ_k are then identified using the portions of the trajectory for which they are valid. The simple models constrain a subspace of the parameters of the entire model M, and the combined results of the identification of the simple models constrain the entire parameter space of M.

As stated thus far, the technique is impractical. This is because for conveniently

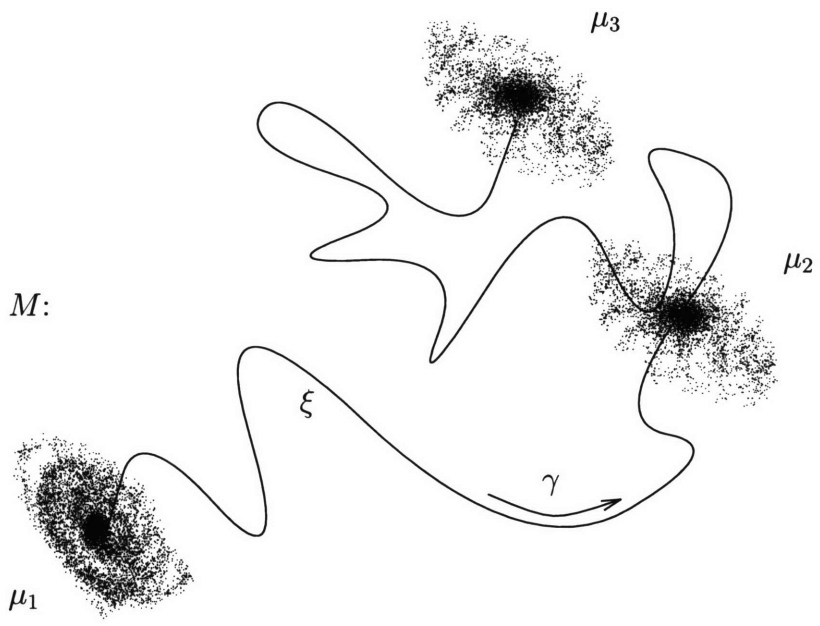

Figure 3-1: Phase space illustration of model decomposition.

simple μ_k, the domains of validity are likely to be vanishingly small. Thus the system identification of the individual simple models will be constrained to a few data points, and the results will be extremely sensitive to any noise. To reduce the effects of noise, it is necessary to identify the simple models over a window. However, the non-zero width of the window may prevent identification of the model close to its region of validity. For example, a model valid at the beginning of a set of data cannot identified over a window centered at $t = 0$, because there is no data for t less than zero. The observation that makes the extrapolative method practical is that *the region of support for identifying a model can be extended beyond the neighborhood where the model is valid.* The proviso is that the "model error" outside of the region of validity be reasonably well behaved. This is illustrated qualitatively in Figure 3-2. In Figure 3-2, the tangent line is proposed as a simple model of a circle valid around the area of intersection. The dashed line shows the hypothetical estimate of a tangent line based only on the "noisy" points of the circle around the area of intersection. Although the dashed line and the solid line are nearly indistinguishable near the point of intersection,

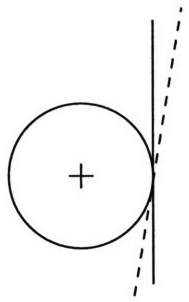

Figure 3-2: Tangent line model of a circle.

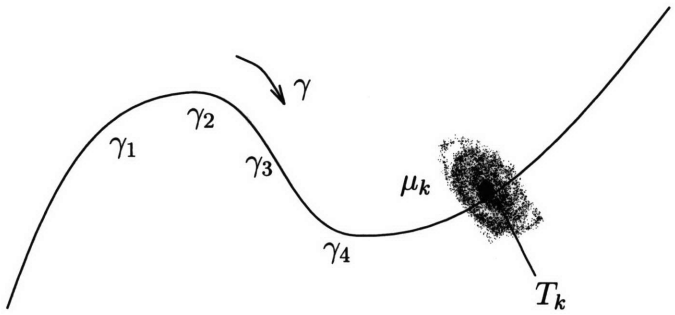

Figure 3-3: Parameterization in the neighborhood of μ_k.

it is clear from looking at the entire circle that the solid line is a better "fit." Figure 3-2 illustrates directly that a very simple model, valid only for a small region, may be supported by data in regions where the model is invalid. This is the observation on which the extrapolative method is based.

To extend the region of support for the simple model beyond its region of validity in a formal way, it is useful to introduce a parameter γ as indicated in Figure 3-3. Then define T_k such that $\mu_k \equiv M$ as $\gamma \to T_k$, as shown.

Then, the model μ_k can be applied to the data to obtain estimates of the parameters for a sequence of windows γ_1, γ_2, etc. tending towards T_k. A series of parameter estimates $\hat{x}(\gamma)$ will be found. Of course, since the model μ_k may not even be close for some γ, there will be a bias $\beta(\gamma)$ associated with the estimates $\hat{x}(\gamma)$. Generally,

$$\hat{x}(\gamma) = x(T_k) + \beta(\gamma) \tag{3.1}$$

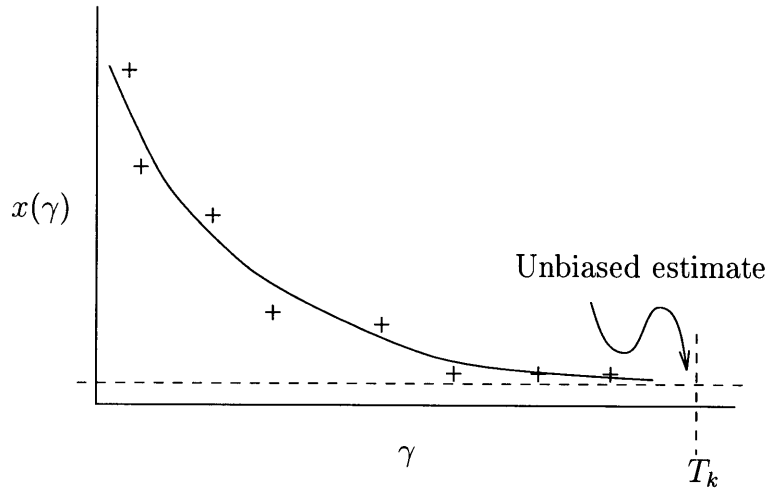

Figure 3-4: Extrapolation to an unbiased estimated.

where $\beta(\gamma) \to 0$ as $\gamma \to T_k$. The assumption is that not only will $\beta(\gamma)$ tend to zero, it will also tend to zero in a well-behaved manner. In this case well-behaved means that a appropriate class of extrapolation functions can be identified *a priori*. Fitting the $x(\gamma)$ with a suitable set of functions, we can extrapolate or interpolate to $x(T_k)$ as shown in Figure 3-4 [1]. This general idea, i.e. using an extrapolative method to evaluate some limit of a function, is called Richardson's deferred approach to the limit [19]. The method is often used in high precision numerical integration routines where either the limit as the step size goes to zero or the limit as the order of the method goes to infinity is of interest. The success of the scheme depicted in Figure 3-4 depends on three conditions. First, the biased estimates must be obtained over windows that are sufficiently long that the bias $\beta(\gamma)$ is a function of model mismatch and not a reflection of noise. Second, the simplified models μ_k must be chosen so that $\beta(\gamma)$ is well behaved approaching T_k. These two conditions are dependent on the model and data at hand. A third condition is that a suitable class of interpolating and extrapolating functions must be used to find the unbiased estimates.

[1] One method of handling colored noise in system identification problems is to add free parameters to "fit the noise" until the residuals are white. The assumption is that the colored noise is filtered white noise, and that the filter has some reasonable form. This is analogous to the extrapolative method, where we "fit the bias" assuming that the bias is reasonably behaved.

3.1 Rational function extrapolation

The class of rational functions has an uncanny ability to approximate well-behaved functions. This property can be understood rather simply. The general rational function takes the form of a ratio of polynomials,

$$R(x) = \frac{N(x)}{D(x)}. \tag{3.2}$$

By performing a partial fraction expansion and assuming no repeated roots in $D(x)$,

$$R(x) = \sum_{i=1}^{N} \frac{a_i x + b_i}{c_i x - d_i}. \tag{3.3}$$

Note that $R(x)$ is a superposition; its general properties can be understood by examination of the individual bi-linear terms in the sum. Each term is an analytic (except at the pole) mapping of $(0, 1, \infty) \mapsto (x_1, x_2, x_3)$. For example, if

$$r(x) = \frac{ax + b}{cx - d} \tag{3.4}$$

then

$$r(0) = -\frac{b}{d} \tag{3.5}$$

$$r(1) = \frac{a+b}{c-d} \tag{3.6}$$

$$r(\infty) = \frac{a}{c}. \tag{3.7}$$

If a single term describes a function of time, for instance, the function can be given particular values at 0, some arbitrary time t, and the limit of the function as $t \to \infty$ is perfectly well defined. The rational function expansion is particularly suitable for capturing overall behavior of functions. For example, the term

$$f(t) = \frac{1}{(e-1)t + 1}. \tag{3.8}$$

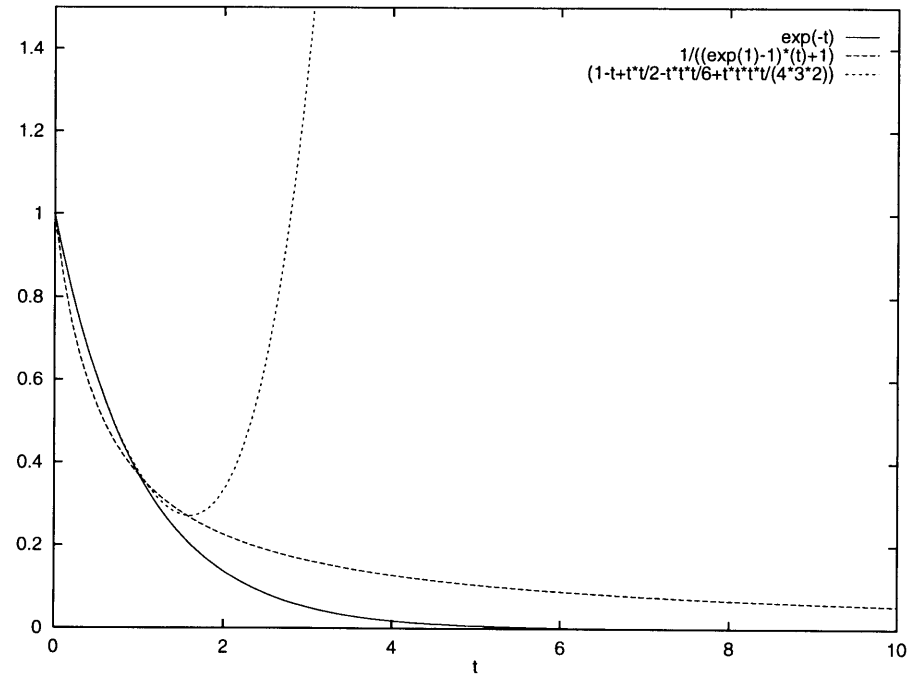

Figure 3-5: Truncated Taylor series and rational approximations to e^{-t}.

approximates $f(t) = e^{-t}$ reasonably well between the support points 0 and 1 and has the same limit as $t \to \infty$. In contrast, a finite order polynomial approximation for e^{-t} based on the Taylor series expansion

$$e^{-t} = 1 - t + \frac{t^2}{2!} - \frac{t^3}{3!} + \cdots + \frac{(-t)^n}{n!} + \cdots \tag{3.9}$$

is guaranteed to be unbounded as $t \to \infty$; adding more terms doesn't help. This is illustrated in Figure 3-5. In Figure 3-5, a fourth order truncated Taylor series and first order rational function approximations to e^{-t} are compared to the actual function. The consequence of the bi-linear term is that a rational function interpolator or extrapolator is likely to have reasonable properties approximating functions with common features; poles, well defined limits, etc. The drawback is that computation of the coefficients of a rational function interpolator given data is not trivial [19], [26]. Fortunately, because of the importance of rational function expansions in Richardson extrapolation and in established methods like Romberg integration and the Bulirsch-Stoer method, methods

are readily available [19].

3.2 RC Example

The extrapolative technique[2] is easily applied to the RC system identification example of Chapter 1. The RC response to a voltage step $v(t) = u(t)$ is

$$i(t) = \frac{1}{R} e^{-\frac{t}{RC}}. \tag{3.10}$$

In the neighborhood of $t = 0$,

$$e^{-\frac{t}{RC}} \approx 1 - \frac{t}{RC} \tag{3.11}$$

by truncation of the Taylor series. Therefore, a "low-time" model is

$$i(t) \approx \frac{1}{R}(1 - \frac{t}{RC}). \tag{3.12}$$

Equation (3.12) can be used as a model in a suitable domain to find R and C. Using the extrapolative technique, however, the model (3.12) is applied for a sequence of γ converging on 0. This is shown in Figure 3-6, where a series of possible "fits" of the linear, low time model are applied to the current transient $i(t)$ for $R = 1\Omega$ and $C = .3F$. In Figure 3-6 the parameter of each low-time fit is $\gamma = t$. In practice the fits in Figure 3-6 would be evaluated over windows to reduce the effects of noise. The low-time model fit at $\gamma = 1$, for example, might be based on data in a window extending from $t = 0$ to $t = 2$.

The low-time model fits shown in Figure 3-6 are interpreted as a sequence of biased estimates for R and C; the next step is to extrapolate the estimates to zero bias. Estimates at zero bias are found by assuming that a "reasonable curve", like a rational function, will capture the behavior of the bias in the parameter estimates as a function of the parameter γ. In this case, since the model is a "low-time" model,

[2]In the extrapolative method, the estimates of the system parameters (i.e. R and C) are themselves parameterized (by γ). To avoid confusion, in this example "parameter" refers to γ and "estimates" refers to R and C.

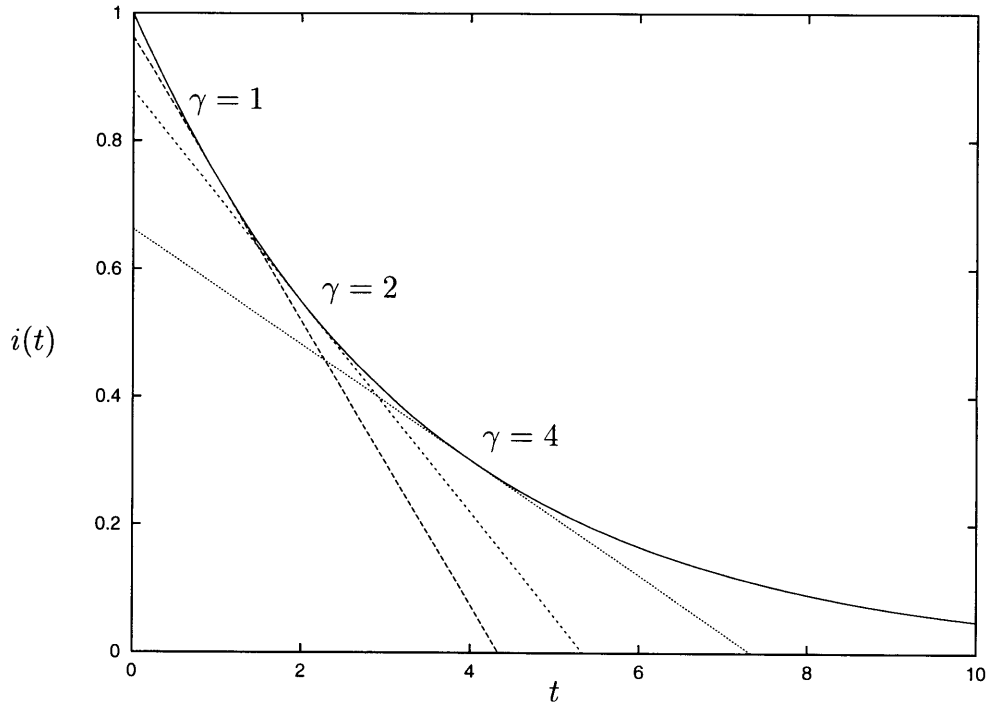

Figure 3-6: A sequence of fits of the "low-time" model to an RC transient.

the sequence of estimates are extrapolated to $\gamma = 0$. The extrapolation process for the estimates $\hat{R}$ and $\hat{C}$ is depicted in Figure 3-7. In Figure 3-7 rational function approximations of $\hat{R}(\gamma)$ and $\hat{C}(\gamma)$ based on the estimates at $\gamma = 1, 2, 4$ are shown. The rational function approximations are used to extrapolate the values of $\hat{R}(\gamma = 0)$ and $\hat{C}(\gamma = 0)$, which should be unbiased estimates of the system parameters $R = 1$ and $C = .3$. For comparison, continuous plots of

$$\tilde{R}(\gamma) = \frac{1}{i(\gamma) - \gamma i'(\gamma)} \tag{3.13}$$

and

$$\tilde{C}(\gamma) = \frac{-1}{\tilde{R}(\gamma) i'(\gamma)} \tag{3.14}$$

are also shown. Theoretically, $\tilde{R}$ and $\tilde{C}$ would result from the noiseless identification of the low-time model over differentially small windows centered on γ. These plots are shown to illustrate how the rational function extrapolator approximates the model error

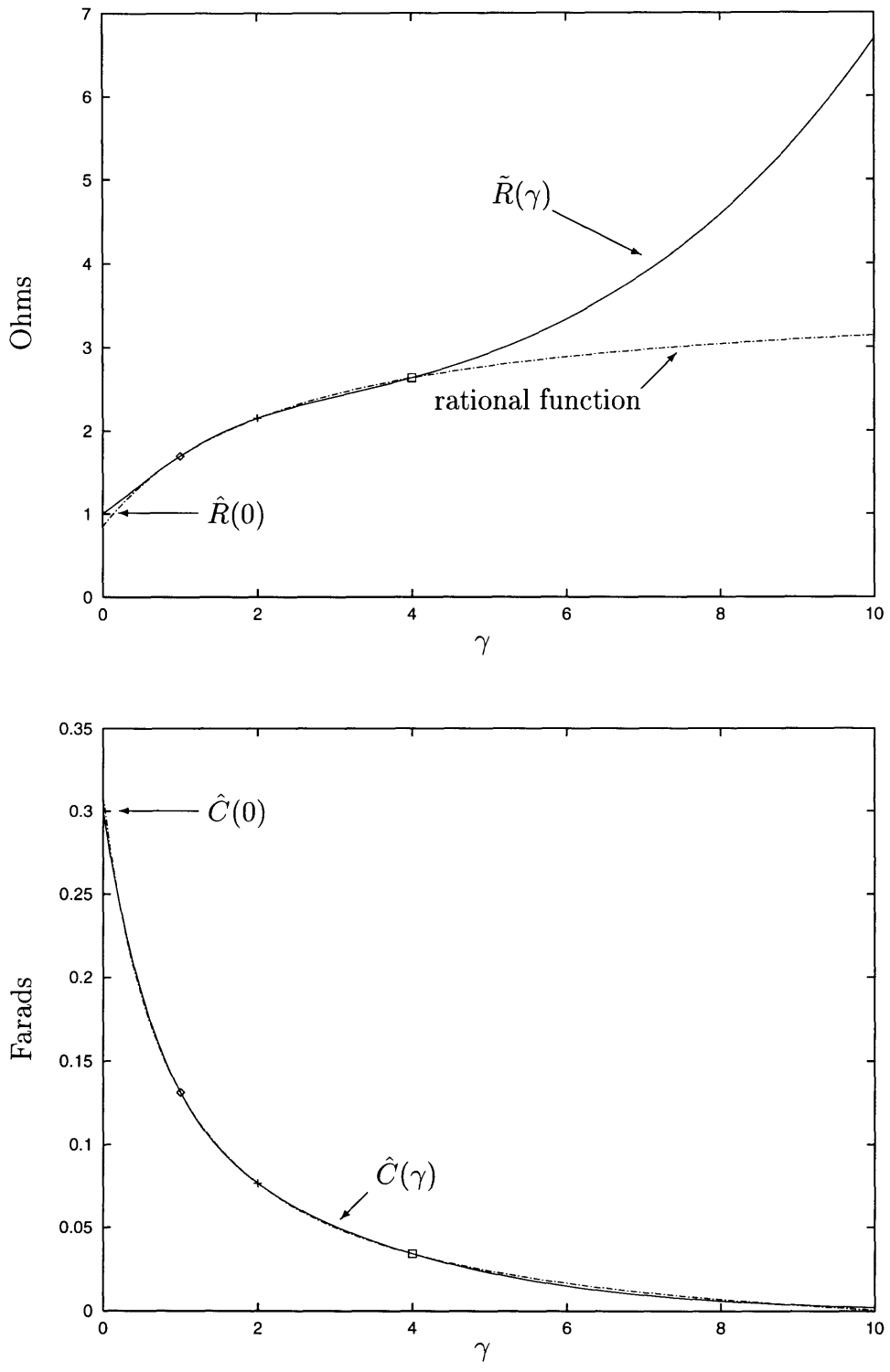

Figure 3-7: Extrapolation of the unbiased estimates $\hat{R}(0)$ and $\hat{C}(0)$ from estimates at $\gamma = 1, 2, 4$.

bias $\beta(\gamma)$. In practice the estimates at $\gamma = 1, 2, 4$ would be obtained from measured data and the plots of $\tilde{R}$ and $\tilde{C}$ would be unavailable. A slight error is evident in the extrapolation to $\hat{R}(0)$ in Figure 3-7. This error arises because the rational function extrapolation, given the three support points shown, does not precisely model the bias. The extrapolative method is approximate. Nevertheless, Figure 3-7 shows that a rational function expansion of model error bias allows a simple model (equation 3.12) to be used for the identification of a more complicated model given data for which the simple model is invalid.

3.3 Simplified induction motor models

To apply the extrapolative system identification technique, one need only find simple models of the more complicated system in question. For the RC circuit, and for any system with a known response of the form e^{At}, a simplified model can be extracted from the Taylor series. A simplified model that takes advantage of the Taylor series for e^{At} avoids the continuous time identification issues discussed in Chapter 1. In the case of the induction motor, we would like to avoid the complication that the rotor currents in the standard model (2.21) are not measured. This is accomplished by developing simple models for the induction motor in the neighborhood of $t = 0$, when the rotor is inertially confined, and $t = \infty$, when the motor and load are in steady state.

An implementation of the high and low slip models below applied using the extrapolative identification procedure can be found in Appendix D.

3.3.1 Model for high slip

Under the condition $s = 1$, which persists at $t = 0$ due to inertial confinement of the rotor,

$$i_r = -i_s. \tag{3.15}$$

The induction motor may be thought of as a transformer with a shorted secondary in this state. Substituting (3.15) into the complex induction motor model (2.21) yields

$$v_s = (r_s + r_r)i_s + 2X_l(\frac{p}{\omega} - j)i_s. \tag{3.16}$$

This is the simplified induction motor model, valid as $t \to 0$. In the first instants of induction motor operation, therefore, two degrees of freedom are eliminated. In particular, the parameter X_l and the sum of the parameters r_s and r_r can be found. Furthermore, (3.16) is valid for any realistic mechanical load.

3.3.2 Model for low slip

For an inertial load, the slip goes to zero in the steady state. In the limit of low slip, there is no torque, hence $i_r = 0$. Again substituting into (2.21),

$$v_s = r_s i_s + (X_l + X_m)(\frac{p}{\omega} - j)i_s. \tag{3.17}$$

The portion of the transient that approaches steady state therefore eliminates two additional degrees of freedom. Combined with the data given by the in-rush current, the four electrical parameters of the induction motor are completely specified.

Mechanical situations more complicated than a simple inertia are generally of interest. These problems can be solved by applying the extrapolative method to the "steady state" model in [11], provided that the mechanical excitation allows the condition

$$\frac{ds}{dt} \to 0. \tag{3.18}$$

In other words, the mechanical system must have a constant or slowly varying steady state relative to the electrical time constants. Note that this condition may be easier or more difficult to satisfy depending on the rating and design of the motor. If the steady state mechanical situation allows relatively error free use of the steady state induction motor model it can be used for identification of the remaining mechanical and electrical parameters in the same fashion as (3.17). System identification of induction motors

using the simple steady state model is discussed by [5] and [29].

3.4 Summary

The extrapolative estimation procedure outlined here obtains quick estimates of the parameters of complicated models by applying standard system identification techniques to reduced models in the temporal domains for which the reduced models are valid. Rational function extrapolation is critical both because it allows the region of support of the estimates to include more data and because it allows the use of models that are valid only "in the limit." To apply the method, one need only derive reduced order models.

The prime advantage of the extrapolative method is that it is fast and easy to implement. For example, in the case of the induction motor, the complexity of the full transient model is irrelevant; one need only consider the simple "blocked rotor" and "steady state" models. The extrapolative method implemented with rational functions is especially attractive because model simplifications that are true "in the limit" can be exploited. For example, in the case of the induction motor the approximation $i_r = -i_s$ is used even though this is true only as $t \to 0$.

Chapter 4

Modified Least Squares

In this chapter, a conventional approach to finding the induction motor parameters is developed. The basic idea is to formulate the system identification problem as a minimization and to solve that minimization problem. It is seen that finding *a* loss function is rather simple, but finding a loss function that is quickly minimized is not. The notation of the complex induction motor model (2.21) is used throughout.

The problem of finding the induction motor parameters can be stated mathematically in a simple way. Since the excitation v_s and the stator currents i_s are known or measured, an estimate of the parameters is given by a minimization of the form

$$\hat{x} = \arg\min_x V(x, i_s, v_s). \tag{4.1}$$

One candidate for the loss function $V(x, i_s, v_s)$ is to set i'_s to the results of a simulation using the parameters x and the excitation v_s. Then the loss function is the squared error between the observed and simulated currents, i.e.

$$V(x, i_s, v_s) = (i_s - i'_s(v_s, x))^T (i_s - i'_s(v_s, x)). \tag{4.2}$$

Theoretically (4.2) could be minimized and the resulting x would be the least-squares parameter solution. Unfortunately, the necessarily iterative procedure to minimize (4.2) requires an expensive simulation for every step. While implementation is simple,

assuming that a sufficiently sophisticated minimization routine exists, the time required makes this algorithm unacceptable. The challenge is to design a loss function that is computationally easy to minimize.

4.1 Eliminating the rotor currents

Incorporating the simulator at every stage of the minimization is a slow way of avoiding the difficulty that the rotor currents i_r are not measured. If the rotor currents were measured, then techniques of Chapter 1 could be used without modification – i.e. form a regression matrix based on the model and solve for the parameters. Another strategy for dealing with the unmeasured rotor currents is to algebraically eliminate those quantities from the model. The transformed model, expressed only in terms of measured or known quantities, could then be used to find the desired parameters. For example, if

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \tag{4.3}$$

and x_2 is not measured, then if b (in general, an operator) is invertible,

$$cx_1 + db^{-1}(r_1 - ax_1) = r_2. \tag{4.4}$$

Equation 4.4 has the desired property of containing only x_1, and the standard techniques from Chapter 1 could be applied to finding the parameters in the operators a through d. If b is not invertible, x_2 can still be eliminated. Applying b and d to (4.3) yields

$$bcx_1 + bdx_2 = br_1 \tag{4.5}$$

$$dax_1 + dbx_2 = dr_2. \tag{4.6}$$

If the commutator (Lie bracket) $[b, d] = bd - db$ is equal to zero, then the term bdx_2 can be isolated in one equation and substituted in the other, eliminating x_2. If $[b, d] = 0$,

then

$$dax_1 + br_1 - bcx_1 = dr_2. \tag{4.7}$$

The complex induction motor model

$$\begin{pmatrix} v_s \\ v_r \end{pmatrix} = \begin{pmatrix} r_s + (X_m + X_l)(\frac{p}{\omega} - j) & X_m(\frac{p}{\omega} - j) \\ (X_m)(\frac{p}{\omega} - sj) & r_r + (X_m + X_l)(\frac{p}{\omega} - sj) \end{pmatrix} \begin{pmatrix} i_s \\ i_r \end{pmatrix}. \tag{4.8}$$

has the same form as (4.3). Eliminating i_r by inverting one of the operators acting on i_r does not seem promising. Both operators acting on i_r have a zero; their corresponding inverses have a pole which could introduce internal stability issues. The operators acting on i_r also have a non-zero commutator due to the time variation of s. To eliminate i_r, a choice must be made between errors due to the pole in the inverses, or errors due to the non-zero commutator.

4.2 Rotor current observer

Since the rotor currents cannot be eliminated algebraically, the next best thing is to estimate the currents and attempt to confine the resulting errors. The most stable method found, from the point of view of numerical errors, was to use the quantity Ψ_s to find i_r. In particular, (2.24) can be rewritten to obtain,

$$\hat{\Psi}_s = \frac{\omega}{p - \omega j}(v_s - r_s i_s) \tag{4.9}$$

which can be used to estimate Ψ_s from the measured quantities v_s and i_s. Recall that the base frequency $\omega = 2\pi 60$. Using the definition of Ψ_s,

$$\Psi_s = X_m(i_r + i_s) + X_l i_s \tag{4.10}$$

i_r can be estimated

$$\hat{i}_r = \frac{1}{X_m}(\Psi_s - X_m i_s - X_l i_s). \tag{4.11}$$

To the extent that (4.9) can be computed accurately, then, $\hat{i_r}$ is expressed in terms of the parameters and the observations i_s and v_s. In Appendix E, equation (4.9) is implemented using the FFT. Although the partial derivatives of this rotor current observer are of interest, considerable effort can be saved by postponing the calculation of partials until the error introduced by the pole in (4.9) is handled.

4.3 The loss function

Having used half the induction motor model to create an observer for the rotor currents, only the second half

$$v_r = X_m(\frac{p}{\omega} - sj)i_s + (r_r + (X_m + X_l)(\frac{p}{\omega} - sj))i_r \tag{4.12}$$

remains for use in the loss function. Note that under the assumed conditions of single excitation, $v_r = 0$. However, if the parameters are incorrect or if $\hat{i}_r$ is not equal to the unmeasurable i_r, Equation (4.12) will have some error ϵ,

$$\epsilon = X_m(\frac{p}{\omega} - sj)i_s + (r_r + (X_m + X_l)(\frac{p}{\omega} - sj))\hat{i}_r. \tag{4.13}$$

The error ϵ will be a combination of the errors due to the observer pole at ω and the errors due to incorrect parameter values [1]. However, the errors due to the observer pole will be in a small neighborhood in the frequency domain around the pole frequency ω. Although the errors due to the observer pole are modulated by the time variation in the slip, the slip is relatively close to DC (given reasonable mechanical loads) in comparison to the observer pole frequency. The errors due to parameter mismatch will be at lower frequencies. For example, if r_r is off by δ the error ϵ will be a "copy" of the relatively low-pass rotor current, i.e. $\epsilon = \delta\hat{i}_r$. The situation in the frequency domain is depicted schematically in Figure 4-1. The indicated solution is to minimize $\epsilon^T\epsilon$ in the frequency domain at those frequencies where the artifact introduced by the

[1] Note that according the definition of the error in (4.13), the estimated parameters will *not* be "least-square parameters" in terms of the observations i_s.

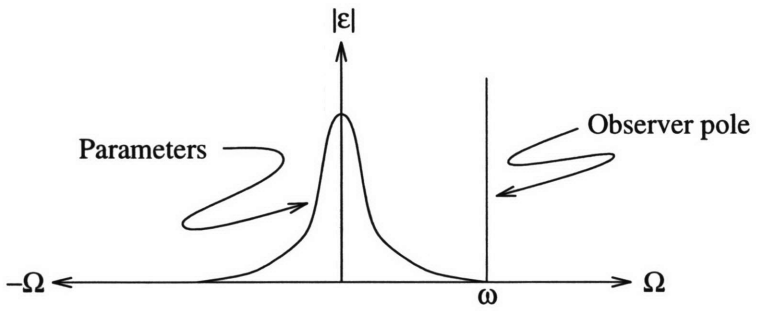

Figure 4-1: Sources of error in the frequency domain.

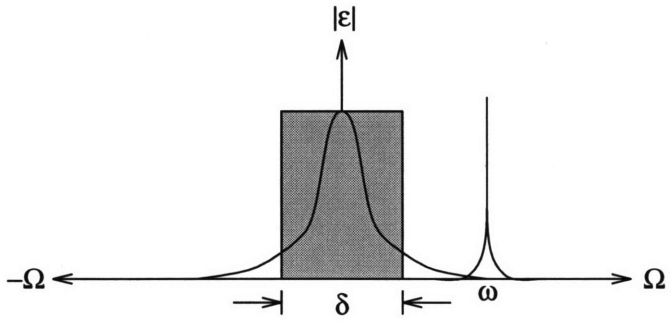

Figure 4-2: Errors are minimized in a selected band.

observer pole has no effect. Note that minimization can be performed in the frequency domain with the same effect as minimization in the time domain. This is guaranteed by Parseval's relation [24] which for a discrete time signal $x[n]$ and its DFT $X[k]$ is

$$\frac{1}{N}\|x\|^2 = \|X\|^2. \tag{4.14}$$

That is, anything that forces the error to zero in the frequency domain also forces the error to zero in the time domain. Furthermore, if the disturbances in the time domain are white, the power spectral density is uniform. The advantage of using the frequency domain is that the minimization can be applied selectively to the errors that are due to parameter mismatch, ignoring the observer pole artifacts, as indicated in Figure 4-2.

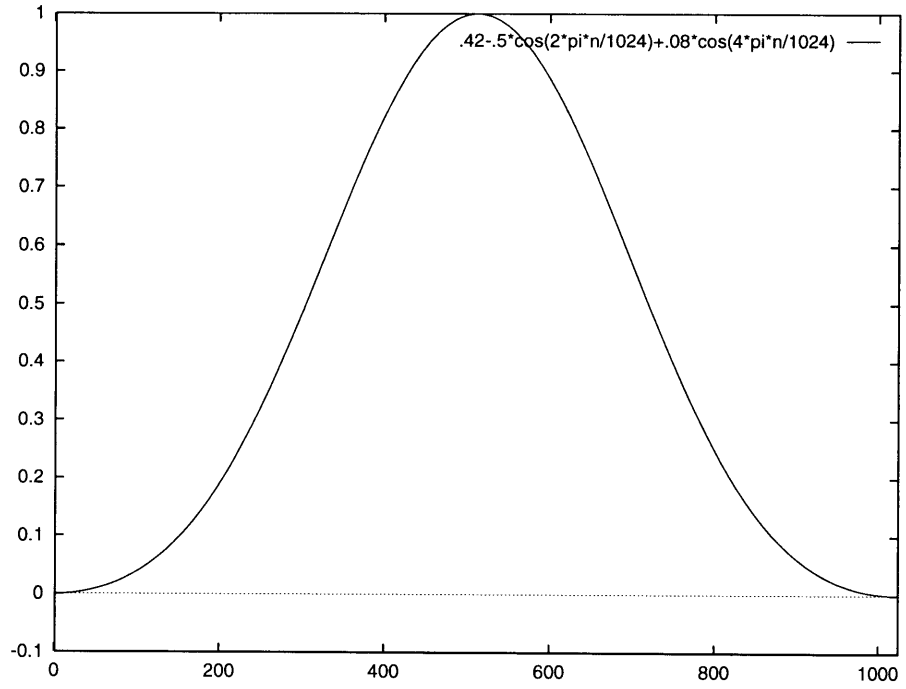

Figure 4-3: Blackman window.

4.3.1 Spectral Leakage

As a consequence of the finite-length of practical data sets, the error introduced by the observer pole spreads into nearby frequencies. This phenomenon is called spectral leakage [18], [10]. In the minimization problem, the difficulty is that spurious data due to the observer pole may "leak" to the frequency range where the error is being minimized. The effect of spectral leakage is shown in Figure 4-2 where the observer pole is "smeared." A practical measure for containing this spectral leakage is to window the time-domain errors ϵ. Various windows are available; any reasonable function that goes to zero smoothly at two points will do. For this work, a Blackman window

$$w[n] = \begin{cases} 0.42 - 0.5\cos(\frac{2\pi n}{M}) + 0.08\cos(\frac{4\pi n}{M}) & 0 \leq n \leq M \\ 0 & \text{otherwise} \end{cases} \tag{4.15}$$

was used [18]. Equation 4.15 is plotted in Figure 4-3 for $M = 1024$.

4.3.2 Partial Derivatives

Algorithms for minimization typically require the availability of partial derivatives of the error with respect to the parameters at various values of the independent variable. If the minimization is performed in the time domain, the desired partials are $\frac{\partial \epsilon}{\partial x_i}\Big|_t$ for the parameters x_i. The partials in the frequency domain are related via the DFT $\mathcal{F}$

$$\frac{\partial \epsilon}{\partial x_i}\Big|_f = \mathcal{F}(\frac{\partial \epsilon}{\partial x_i}\Big|_t). \tag{4.16}$$

For simplicity the partial derivatives are given here in the time domain, with the understanding that the partials must be windowed and transformed to minimize the windowed and transformed loss function.

First, consider the effect of the rotor current observer. Although the estimation of rotor currents is performed for numerical reasons via the flux linkages, the algebraically equivalent expression in currents can be used for computation of partial derivatives. The relevant equation is

$$v_s = r_s i_s + (X_m + X_l)(\frac{p}{\omega} - j)i_s + X_m(\frac{p}{\omega} - j)\hat{i}_r. \tag{4.17}$$

Since the minimization is confined to low frequencies as shown in Figure 4-2, the approximation

$$\frac{p}{\omega} \approx 0 \tag{4.18}$$

is valid. Setting $\frac{p}{\omega}$ to zero in (4.17),

$$v_s = r_s i_s - j(X_m + X_l)i_s - jX_m\hat{i}_r. \tag{4.19}$$

Using the fact that $\frac{\partial i_s}{\partial x}$ is zero for all parameters x, the partials $\frac{\partial \hat{i}_r}{\partial x}$ are, in the limit of low frequencies,

$$\frac{\partial \hat{i}_r}{\partial X_m} = -\frac{i_s + \hat{i}_r}{X_m} \tag{4.20}$$

$$\frac{\partial \hat{i}_r}{\partial X_l} = -\frac{i_s}{X_m} \tag{4.21}$$

$$\frac{\partial \hat{i}_r}{\partial r_r} = 0 \tag{4.22}$$

$$\frac{\partial \hat{i}_r}{\partial r_s} = -j\frac{i_s}{X_m}. \tag{4.23}$$

Ultimately the partials $\frac{\partial \epsilon}{\partial x}$ for all parameters x are desired. Applying the approximation (4.18) to (4.13) yields,

$$\epsilon = -sjX_m i_s + r_r i_r - sj(X_m + X_l)\hat{i}_r. \tag{4.24}$$

Application of the chain rule and appropriate substitutions yields

$$\frac{\partial \epsilon}{\partial X_m} = (r_r - sjX_l)\frac{\partial \hat{i}_r}{\partial X_m} \tag{4.25}$$

$$\frac{\partial \epsilon}{\partial X_l} = -sj(\hat{i}_r - i_s) + (r_r - sjX_l)\frac{\partial \hat{i}_r}{\partial X_l} \tag{4.26}$$

$$\frac{\partial \epsilon}{\partial r_r} = \hat{i}_r \tag{4.27}$$

$$\frac{\partial \epsilon}{\partial r_s} = (r_r - sj(X_m + X_l))\frac{\partial \hat{i}_r}{\partial r_s}. \tag{4.28}$$

That the partial derivatives of the error function can be expressed analytically makes the minimization of the associated loss function susceptible to a broad range of algorithms. The problem as stated is suitable for the Gauss-Newton procedure from Chapter 1, for example. While Gauss-Newton is attractive for its speed and simplicity, in situations where it is undesirable to repeatedly modify the initial guess to achieve convergence a more stable algorithm is preferable. The suggested algorithm is Levenburg-Marquardt, given in [2] and [19]. Minimization of (4.24) given the partial derivatives in 4.28 using the Levenburg-Marquardt algorithm is implemented in the files `identify.cc` and `estimate.cc` in Appendix E. The file `mrqmin.cc` is a slightly modified version of the like-named file from Numerical Recipes [19].

4.4 The mechanical interaction

The proceeding development neglects the mechanical system associated with the induction motor. The partial derivatives of the error function are written assuming that $\frac{\partial s}{\partial x} = 0$ for any parameter x, and therefore the equations above are useful for the case where $s(t)$ is actually measured. Measurement of $s(t)$ is a great advantage, because it effectively decouples the electrical and mechanical systems. Without $s(t)$ the parameters of the mechanical system must be identified in addition to electrical parameters from the electrical transient only. At some point, the complexity of the combined electrical and mechanical model might exceed the information content of the electrical transient.

In many cases, it might be possible to work around the limited information content of the transient. This is because the mechanical system is often simple during the electrical startup transient. For example, in typical operation of a machine tool like a lathe or mill, the induction motor is started with only an inertia and windage load. Then the cutter is applied to the material, introducing a mechanical disturbance and a substantially more complicated modeling problem. However, it is possible that during the initial startup transient, the induction motor parameters and the simple mechanical model could be identified and used in the subsequent operation of the tool to gather information about the machining process, for instance. Furthermore, if no slip information were required, this data could be gathered non-intrusively. Identification of the induction motor parameters and a relatively simple mechanical load model is therefore likely to be helpful even in complicated scenarios.

4.4.1 Slip estimation by reparametrization and shooting

The electro-mechanical interaction of the induction motor with an inertial and damping load is governed by Equation 2.16, reproduced here for convenience

$$\frac{ds}{dt} = \gamma(i_{ds}i_{qr} - i_{qs}i_{dr}) + \beta(1 - s). \tag{4.29}$$

Define a time t_0 when the system enters steady state such that

$$\frac{ds(t_0)}{dt} = 0 \tag{4.30}$$

and define the steady state slip

$$s_0 = s(t_0) \tag{4.31}$$

also, let the torque estimate

$$\hat{\tau}(t) = i_{ds}\hat{i}_{qr} - i_{qs}\hat{i}_{dr}. \tag{4.32}$$

With these definitions, β is determined as a function of γ:

$$\beta = \frac{-\gamma\hat{\tau}(t_0)}{1 - s_0}. \tag{4.33}$$

Substituting into (2.16),

$$\frac{d\hat{s}}{dt} = \gamma\hat{\tau}(t) + \frac{-\gamma\hat{\tau}(t_0)}{1 - s_0}(1 - \hat{s}). \tag{4.34}$$

Assuming that s_0 and t_0 are specified, Equation 4.34 is a boundary value problem. Thus, β, γ and $\hat{s}(t)$ can be found by a one dimensional shooting method [26]. Effectively, the parameters s_0 and t_0 have been substituted for the parameters γ and β. There are many advantages to this reparametrization. Any t in the steady state operating region of the motor is a suitable t_0. In practical terms, if the startup transient of a motor is fully captured, t_0 is simply the last time coordinate in the data set. Also, it is likely that s_0 is known with reasonable accuracy; the speed at rated load is often printed on the machine. Even if the steady state slip cannot be determined *a priori*, s_0 is a much "friendlier" parameter than either γ or β because it is dimensionless, bounded absolutely[2] by 1 and 0, and bounded approximately by the torque estimate $\hat{\tau}(t_0)$ and the machine rating. Another advantage of introducing s_0 as a parameter is

[2]It is assumed that whether or not the machine is being used a generator is known.

that the slip estimate $\hat{s}(t)$ is approximately independent of the electrical parameters. The electrical parameters are involved through $\hat{i}_r$, but the overall scaling of the slip estimate is determined by the "boundary condition" s_0.

The shooting method for slip estimation is implemented in Appendix E. A hybrid Newton/bisection shooting algorithm finds a γ on each iteration to match the boundary condition s_0 given the updated rotor current estimates. The Newton method is tried first, and the bisection method is used if the Newton method fails. The computational cost of the shooting method is minimal, since the results of the previous iteration are used as an initial guess. Shooting methods are discussed in [26],[19].

4.4.2 The general case

The general case, with an arbitrary mechanical load, can be addressed by estimating the slip based on the rotor current observer, the mechanical model and its tentative parameters. However, with the rotor current observer used here the approach is problematic. The difficulty is that the rotor current observer pole introduces high frequency errors in the rotor currents. To predict torque, the rotor currents are *multiplied* by the stator currents. If the stator currents have frequency content at the rotor observer pole frequency, the multiplication produces an image at DC and at twice the frequency of the rotor observer pole. The high frequency component tends to have little effect on the slip due to the mechanical filter formed by the inertia in the mechanical load model; the DC torque component, on the other hand, has a substantial effect. This is an area that requires further investigation.

4.5 Summary

The induction motor system identification problem is relatively easy to state mathematically as a minimization. Given a simulator for a system, a loss function can be written incorporating the simulator that expresses exactly the intent of the identification procedure – to find a set of parameters matching the observations in a least squared sense. The problem is that such a loss function is expensive to evaluate and

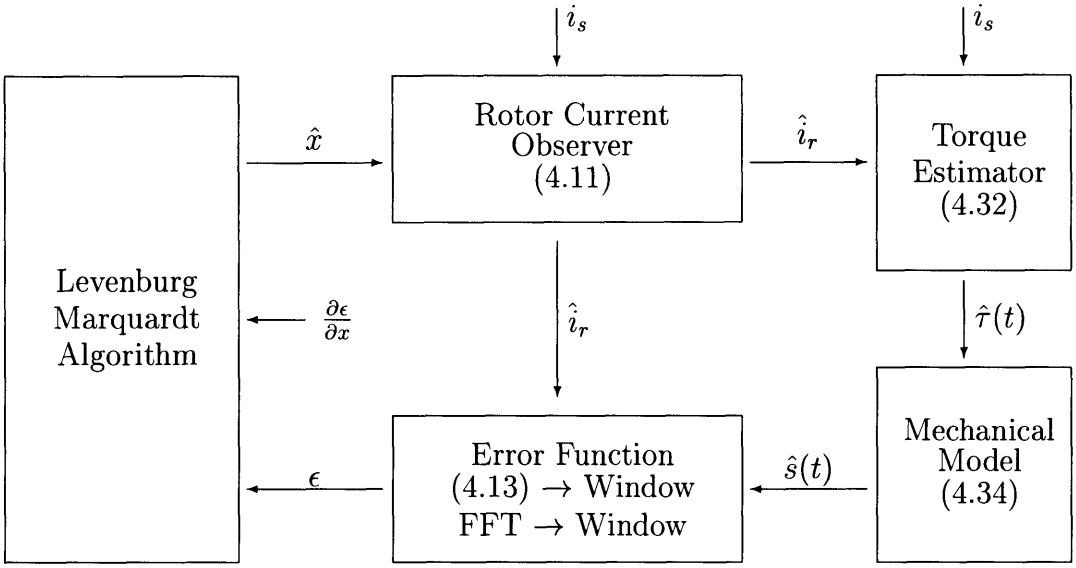

Figure 4-4: Schematic diagram of modified least squares method.

difficult to minimize.

In this chapter, a computationally efficient loss function is formulated. First, a rotor current observer is presented. The estimated rotor currents are used to estimate torque and also to compute components of the error. The error is minimized using the Levenburg-Marquardt method, which requires the partial derivatives of the error with respect to the parameters.

Two methods of dealing with the mechanical system are presented. First, if $s(t)$ is supplied, the identification of the electrical parameters is decoupled from the mechanical interaction. Slip measurements are very valuable if available, because then the electrical and mechanical identification problems can be treated independently. If $s(t)$ is not available, and the load is assumed to be inertia and damping only, the mechanical subsystem can be reformulated using a shooting technique so that the new parameters are either known *a priori* or easily estimated. An additional advantage of the reparametrization is that the approximate slip can be treated as independent of the electrical parameters. This allows the same partial derivatives to be used for minimization whether or not the slip is measured.

A block diagram of the entire estimator is shown in Figure 4-4. In Figure 4-4 $\hat{x}$ is the estimated vector of parameters, and it is assumed that the slip must be estimated. The numbers in the diagram indicate the equations associated with each block.

Chapter 5

Results

The induction motor identification methods presented in this thesis were validated using both simulated and measured data. Tests using simulated data compare the estimated parameters to the parameters used to create the simulated data. No noise was added to the simulated data. To evaluate the performance of the methods on real data, estimated parameters were used to create simulated transients which were compared to the collected data. Note that validation with simulated data tests the identification procedure in isolation, where validation with measured data tests both the motor model and the identification procedure.

5.1 Simulated data

Simulated data were obtained using the simulator discussed in Chapter 2. Figure 5-1 demonstrates the variety in the simulated motor test data. Figure 5-1 is a composite plot of transient currents i_{qs} and slip curves for several different motors. The motors are the 3 hp, 50 hp and 500 hp motors used as examples in [11]. Note that the 500 hp transient includes generative operation of the motor, as the slip goes negative around sample number 3500. The 2250 hp test motor is qualitatively similar to the 500 hp transient, except that the peak current is in the neighborhood of 4000 A. To preserve the character of the lower power waveforms, the 2250 hp transient is not shown. Note that the sampling rates for the different transients are not the same; the data is sampled as used for identification.

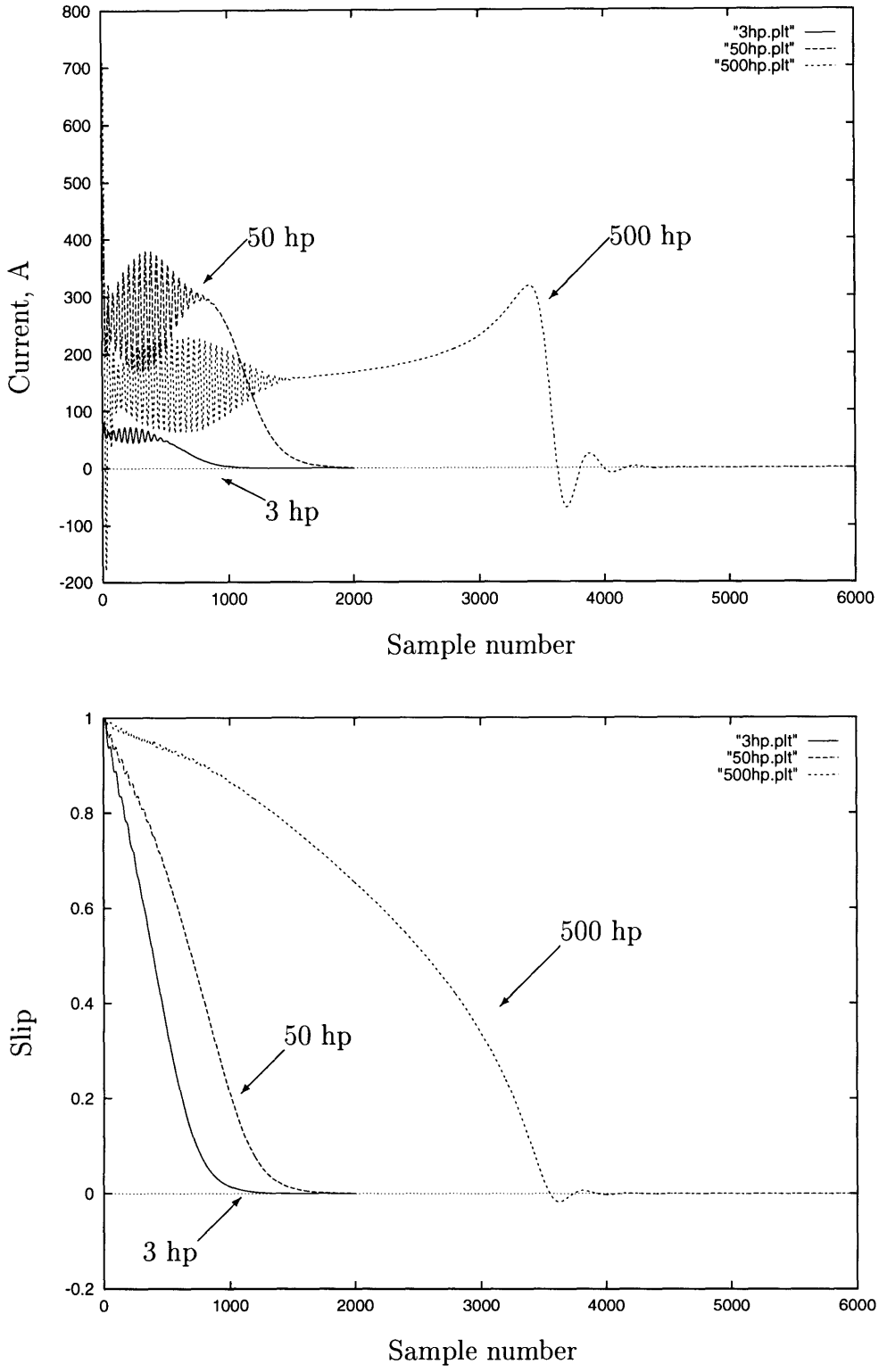

Figure 5-1: Composite plot of slip and current transients for test motors.

Motor		True Parameters (Ω at 60 Hz)	Estimated Parameters (Ω at 60 Hz)
3 hp 220 V	X_m	26.13	26.144
	X_l	.754	.739
	r_r	.816	.808
	r_s	.435	.434
50 hp 460 V	X_m	13.08	13.38
	X_l	.302	.2917
	r_r	.228	.2702
	r_s	.087	.0647
500 hp 2300 V	X_m	54.02	54.10
	X_l	1.206	1.205
	r_r	.187	.472
	r_s	.262	.214
2250 hp 2300 V	X_m	13.04	13.03
	X_l	.226	.231
	r_r	.022	.034
	r_s	.029	.029

Table 5.1: Extrapolative method estimates, simulated data.

5.1.1 Extrapolative method

Results for the extrapolative method presented in Chapter 3 are shown in Table 5.1. The column labeled "true parameters" contains the values used by the simulator to generate a transient. The column labeled "Estimated Parameters" contains the parameters output by the extrapolative identification program.

Recall that no initial guess is required for the extrapolative method, and that the method is not iterative.

5.1.2 Modified least-squares method

Table 5.2 shows data obtained using the modified least-squares method presented in Chapter 4. No particular care was taken in picking initial guesses, although informal experimentation revealed that the convergence of the method was rather robust. Rigorous analysis of the convergence properties of the method is left to future work.

Motor		True Parameters (Ω at 60 Hz)	Initial Guess (Ω at 60 Hz)	Estimated Parameters $s(t)$ given (Ω at 60 Hz)	Estimated Parameters $s(t)$ unknown (Ω at 60 Hz)
3 hp 220 V	X_m	26.13	24.0	26.19	26.49
	X_l	.754	1.0	.7496	.7521
	r_r	.816	1.0	.8130	.7705
	r_s	.435	0.3	.4358	.4169
50 hp 460 V	X_m	13.08	10.0	13.07	12.99
	X_l	.302	0.5	.3018	.2984
	r_r	.228	0.5	.2263	.1950
	r_s	.087	.1	.0880	.0845
500 hp 2300 V	X_m	54.02	50.0	48.82	52.22
	X_l	1.206	1.0	1.207	1.207
	r_r	.187	.3	.1843	.1829
	r_s	.262	.4	.2672	.2643

Table 5.2: Modified least-squares method, simulated data

In Table 5.2, "True Parameters" are the parameters used by the simulator to generate test data. The "initial guess" column lists the parameter guess passed to the iterative method. Note that the initial guesses in Table 5.2 are uniformly worse than than the results from the extrapolative method listed in Table 5.1. The modified least-squares method was run both with a "measured slip" and with a slip estimator, as discussed in Chapter 4. The slip might be available in some situations; for example, when characterizing a particular motor for control purposes. The "slip unknown" column corresponds to the nonintrusive diagnostics scenario, where slip measurement is not possible.

5.2 Measured Data

Real data was obtained from transient tests on a typical three-phase industrial induction motor. The test induction motor was connected to a three-phase 208 V line to line 30 A rated service using a solid state three-phase switch with a programmable firing angle [15]. Voltage and current data during the startup transient were collected on a four channel TDS420A digital storage oscilloscope using isolated A6909 voltage

Leyland-Faraday Electric Company		
Type: AEEA	Model: LFI-3050	Phase: 3
Hp: 5	Volts: 208-230/460	
Rating: Cont	Cycles: 60	
AmbTemp: 40 C	RPM: 3420-3480	
Frame: 184 T	Service Factor: 1.15	
Amps: 12/6	Nema Design: B	

Table 5.3: Boiler-plate data from test induction motor

and A6303 current probes. Since the firing angle was under computer control, the experiment was assumed to be repeatable. Hence, the current and voltage measurements were actually made in two successive tests. For synchronization of the two tests, one channel of voltage information was stored while collecting the three currents. The oscilloscope was set to trigger from this voltage channel for both current and voltage measurements. Data collected by the oscilloscope was stored on disk in Tektronix .WFM format and translated to files suitable for input to the identification procedures by the BASH script and C programs listed in Appendix G.

The boiler-plate data from the test induction motor is reproduced in Table 5.3. No mechanical load, except for the rotor inertia and windage, was attached to the motor.

Figure 5-2 shows the current transient and resulting voltage distortion for the test motor in the dq0 frame. Note that while the maximum-load power rating of the motor is within the 30 A per phase rating of the three phase service, the motor draws currents well in excess of 30 A during the startup transient. Also, the assumption of balanced conditions seems to be violated; both the voltage and current have a "zero" component, particularly in the high current portion of the transient.

The "spikes" on the voltage plots are associated with the switching of the alternistor used to control the three phase switch. The currents per phase during the first $.2s$ exceed the alternistor's steady state rating by about a factor of four.

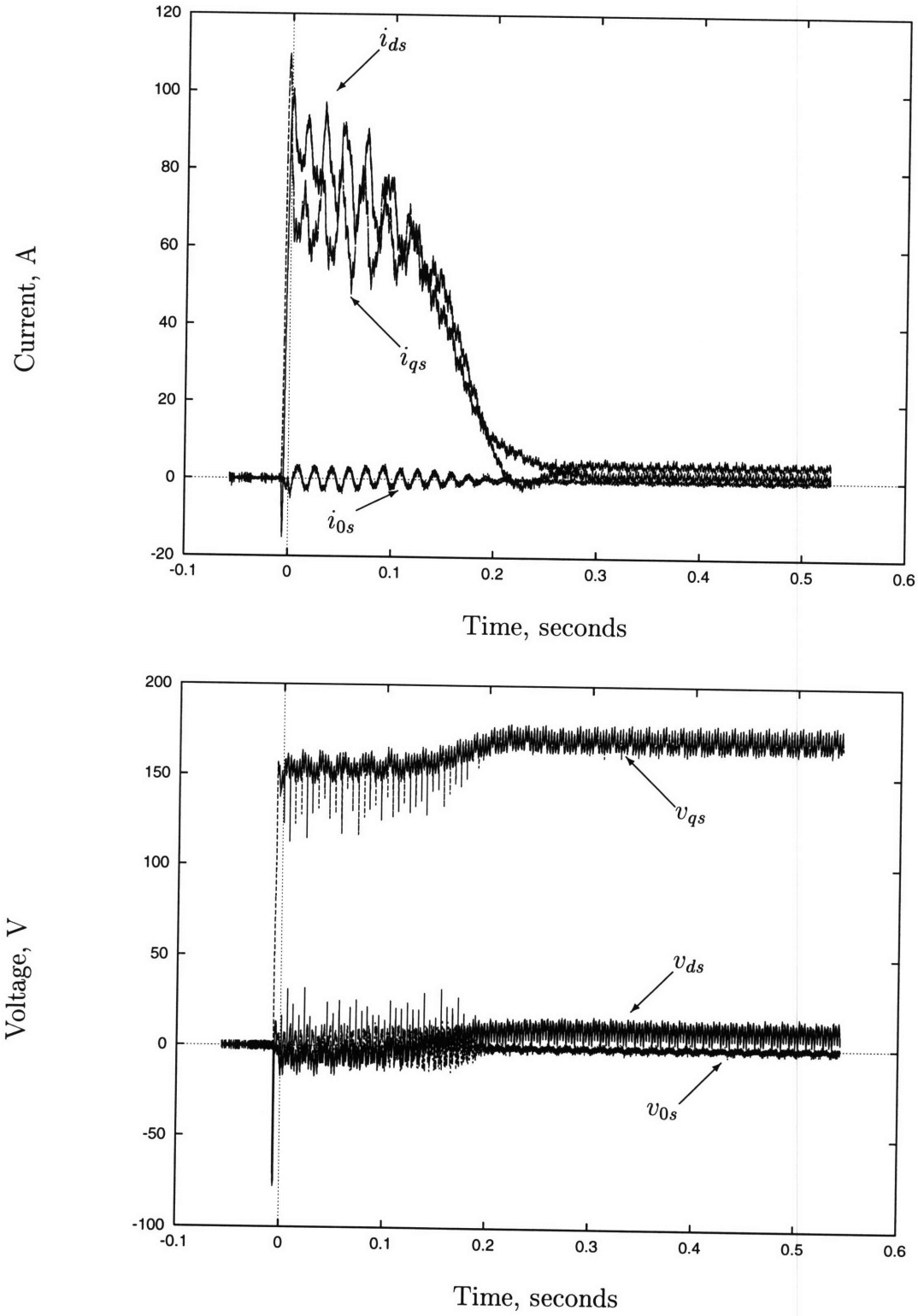

Figure 5-2: Current and voltage measurements in dq0 frame, unloaded motor.

5.2.1 Extrapolative method

The extrapolative method was used to analyze the motor transient data in Figure 5-2. The program `elh` from Appendix C was used, as shown in the following interaction.

```
darkstar:/home/sshaw/elh# elh < small.elh
Elh Version 0.1a | Data from 5 hp induction motor
Steve Shaw, 1996
1250
1400
5000
10000
0.03
14000
4e-05
Error Margins show error expected in extrapolation.
They are not a reflection of the data quality.
Xl    = 0.546740, +/- -0.027198
Rr+Rs = 0.710156, +/- 0.048885
Xss   = 39.521109, +/- 0.049014
Rs    = 7.609524, +/- 1.864681
darkstar:/home/sshaw/elh#
```

The output above seems reasonable except for the value of r_s. The error margin and unrealistically large value for r_s indicate that the method had problems identifying the "low-slip" model. This was expected, because the low-slip model used was for an inertial load; the real motor's mechanical load includes unmodeled friction from the fan and bearings. The extrapolative method is intended to provide reasonable initial guesses for an iterative method, however, and accurate identification of a partial set of parameters is still valuable information. To confirm that the estimates X_l, $r_r + r_s$ and $X_{ss} = X_m + X_l$ determined by the extrapolative method are reasonable requires comparison of a simulated data set (using those parameters) with the measured data set. The following parameters were used to generate a simulated data set.

$$X_l = .547\Omega$$

$$X_m = 38.97\Omega$$

$$
\begin{aligned}
R_r &= .355\Omega \\
R_s &= .355\Omega \\
\alpha &= .05\Omega
\end{aligned}
$$

Since the extrapolative method estimate of r_s was assumed invalid, r_s and r_r in the above parameter set are arbitrarily made equal. The mechanical parameter $\alpha = .05$ was picked so the transient would have approximately the right length. It should be emphasized that the above parameter set was *not* completely generated by the extrapolative method presented. Only three degrees of freedom in the parameter space were determined by the extrapolative method. Guesses for the remaining two degrees of freedom are supplied so that an induction motor simulation can be used to confirm that the estimates that *were* produced by the extrapolative method are valid. The simulation should closely match the measured data at the beginning of the transient, since the extrapolative method was able to identify the high-slip model. Comparison of the simulated and measured data in the middle of the transient is not meaningful. At the end of the transient, the steady state should be reasonably accurate since $X_m > r_s$ and the load is light. The simulated and measured data sets are compared in Figure 5-3.

The insets in Figure 5-3 show the agreement between measured data and simulated data in the high slip region where the extrapolative method was successful. Note that the large discrepancy in the comparison of i_{ds} is likely due to the voltage distortion of v_{ds} (see Figure 5-2). The simulations shown in Figure 5-3 do not take the line voltage distortion into account.

5.2.2 Modified least-squares method

The measured induction motor data was also analyzed with the modified least squares method presented in Chapter 4. The initial guess (in Ω at 60 Hz) was $X_l = 1.00$, $X_m = 45.0$, and $r_r = r_s = .5$. In addition, the slip s_0 at $t_0 = .4s$ was estimated to be .005, approximately an order of magnitude below the slip at rated load according

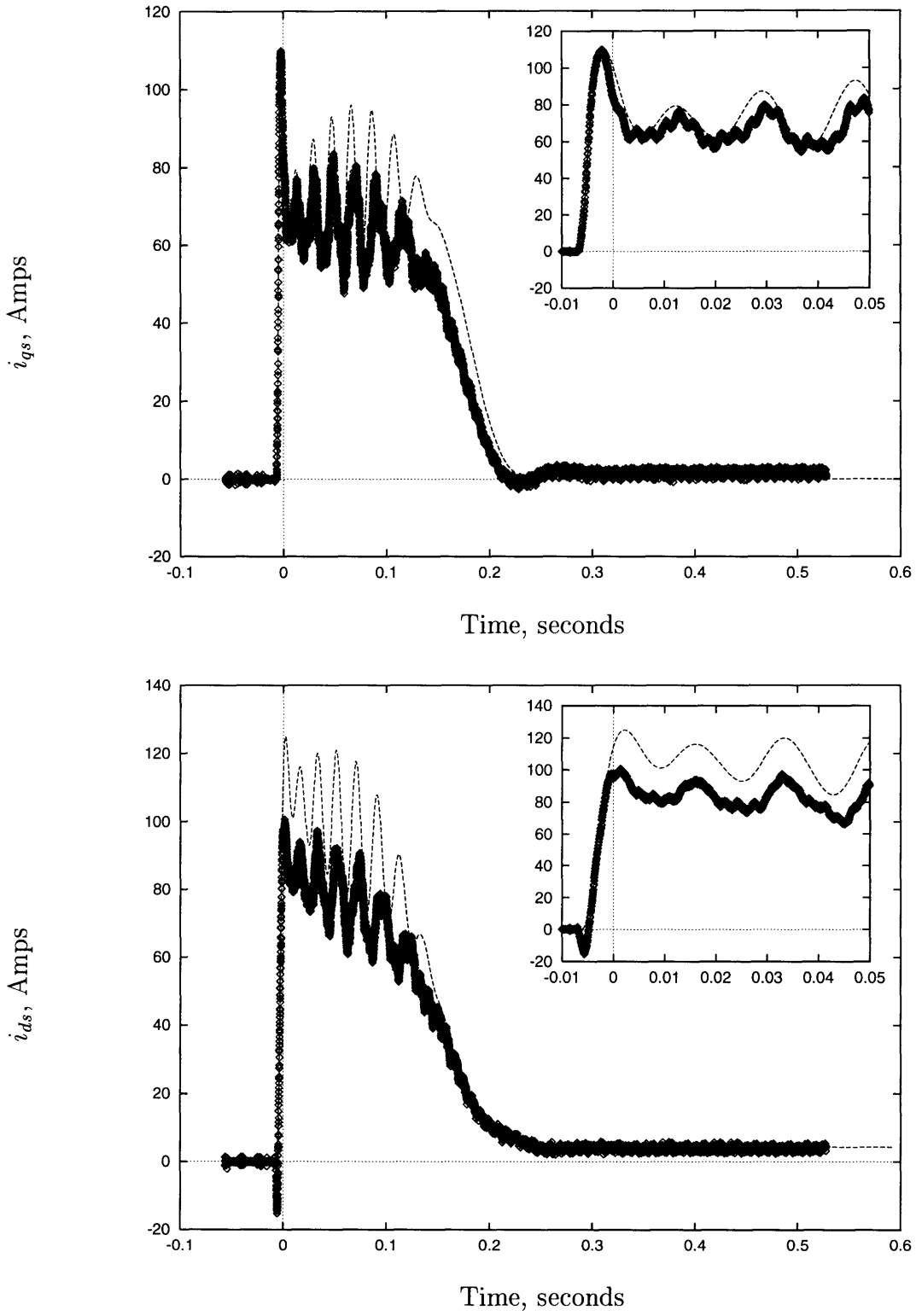

Figure 5-3: Comparison of simulated and measured transients.

to the motor rating. The following screen interaction was obtained by running the program `identify` from Appendix D. Note that the parameters below are listed in the order X_m, X_l, r_r, r_s. The fifth parameter shown is not used, and `beta` is the mechanical parameter used to estimate the slip. `Chisq` is the squared magnitude of the residual, and $i = 6$ indicates that the method converged in six iterations.

```
beta = 0.396901
Chisq[6] = 546.911638
Alambda[6] = 0.000100


Stopping. i = 6
Chisq = 546.911638

parameters:
3.847e+01
6.048e-01
3.356e-01
4.767e-01
3.000e-01
covariance matrix:
 2.8e+00    9.5e-04   -1.3e-04    -4.4e-04   0.0e+00
 9.5e-04    4.5e-06    4.5e-07    -2.6e-07   0.0e+00
-1.3e-04    4.5e-07    1.4e-05    -2.6e-05   0.0e+00
-4.4e-04    2.6e-07   -2.6e-05     6.2e-05   0.0e+00
 0.0e+00    0.0e+00    0.0e+00     0.0e+00   0.0e+00
```

The parameter estimates displayed above compare favorably to the results from the extrapolative method. In addition to the electrical parameters, the modified least squares method also generates a slip estimate shown in Figure 5-4, as described in Chapter 4. Note that the test motor overshoots at $t = .2s$, according to the predicted slip curve. Note that the mechanical parameters, although not displayed above, are easily obtained from the estimated slip curve and electrical parameters.

To validate the estimates generated by the modified least squares method, the estimated slip shown in Figure 5-4, the distorted voltage waveform from Figure 5-2 and the parameter estimates were input to a simulator in an attempt to reproduce

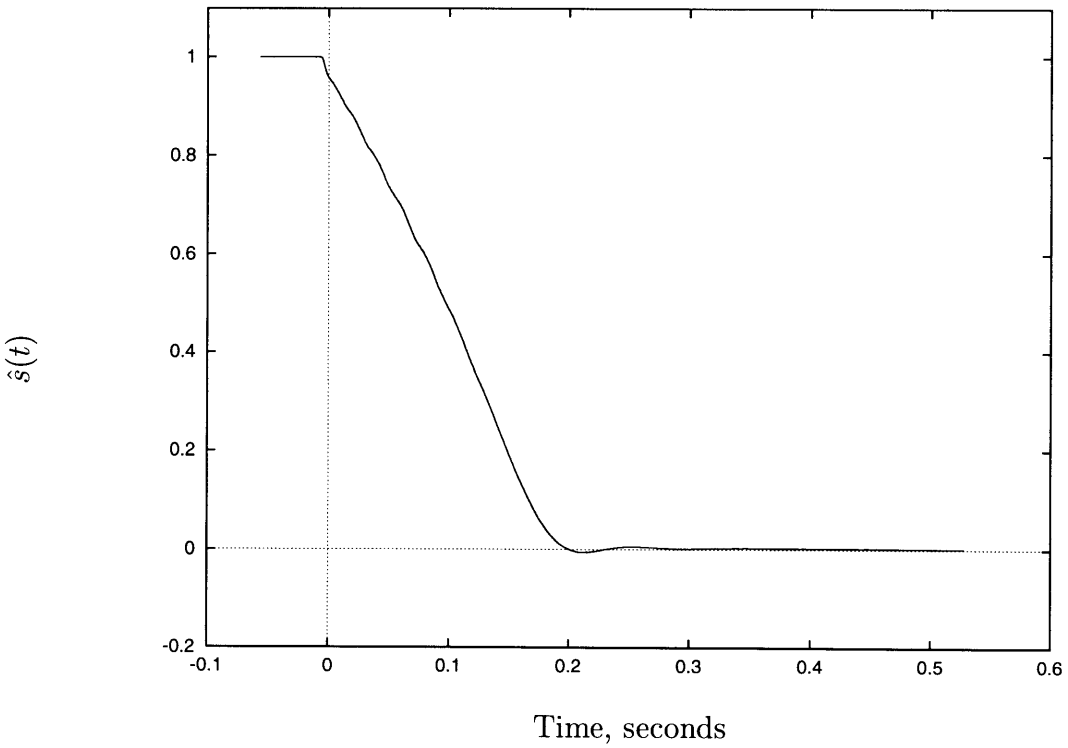

Figure 5-4: Estimated slip, modified least-squares method.

the measured currents. This test not only verifies the accuracy of the parameter estimates, it also validates the applicability of the induction motor model assumed in this thesis. It should be noted that while the distorted voltages v_{qs} and v_{ds} were used in the simulation, the imbalance (i.e. v_{0s} in Figure 5-2) was ignored. The measured and simulated currents i_{qs} and i_{ds} are shown in Figure 5-5.

The agreement in Figure 5-5 is quite good overall. The slight mismatch between estimated and measured currents is likely due to unmodeled effects and measurement noise. For example, one feature not modeled in this thesis is the variation of rotor resistance with slip. In the frame of the rotor the flux wave imposed by the stator has a frequency determined by the slip; at high slip the rotor is immersed a high frequency magnetic field, and at low slip a low frequency magnetic field. Assuming that the motor has a cast-rotor squirrel cage design, the radial penetration of the rotor currents into the rotor bars is governed by magnetic diffusion. The cross sectional area of the rotor bar effectively used to conduct the rotor currents is therefore determined

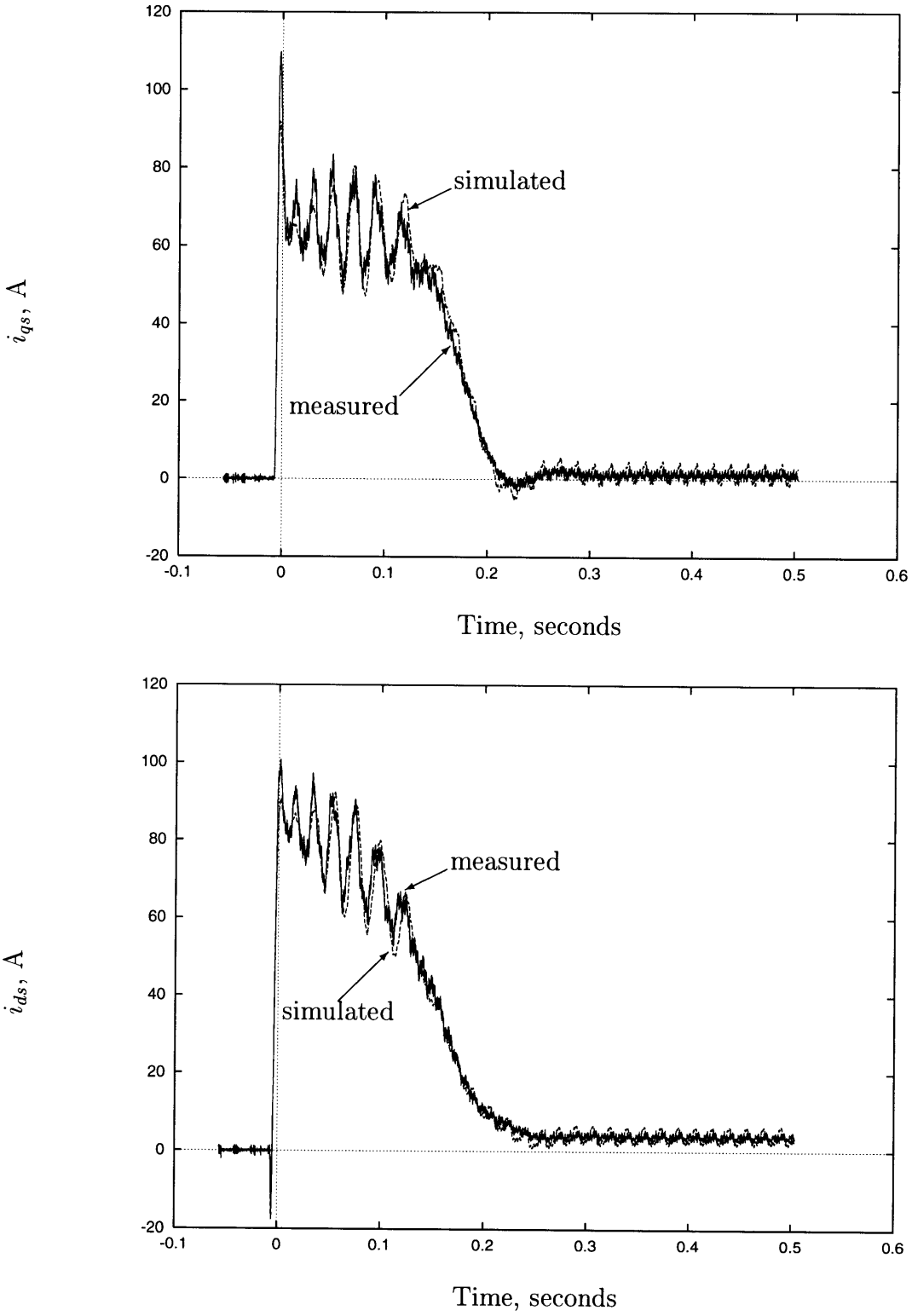

Figure 5-5: Comparison of measured and simulated dq currents.

by the frequency of the impinging stator field, which is in turn determined by the slip. The interaction between rotor current penetration depth and slip is exploited in some induction motor designs to enhance the torque slip characteristics. Another unmodeled phenomenon is the effect of heat. The currents drawn by the test motor during the startup transient are quite large compared to the steady state full-load current one would expect a 4 kW machine to draw. The large startup transient currents heat the conductors, resulting in a change of resistance as the machine warms up. Similarly, the effects of non-ideal magnetics are not considered.

Chapter 6

Conclusions

This thesis presents two techniques for identifying the parameters of an induction motor from voltage and current measurements made during the startup transient. In addition, induction motor simulation and basic techniques for system identification are reviewed. The system identification techniques developed here were tested on both simulated and measured data with good results.

6.1 Extrapolative method

The extrapolative method performed extremely well on simulated data, particularly in view of the fact that the method was developed to obtain reasonable initial guesses. For example, when identifying the 3 hp simulated motor data, the largest fractional error for any parameter was 2%. For other motors, and on measured data, the extrapolative method was typically off in one or two parameters. For example, on a 2250 hp motor, the extrapolative method was accurate to 2.5% on all parameters except for r_r. However, the results for the 2250 hp motor would still reduce a four dimensional non-linear optimization problem to a one dimensional problem. Based on comparison with the modified least-squares parameters, the extrapolative method did not perform as well on real data as on simulated data. In fairness, it should be noted that the tests for the extrapolative method did not take the voltage distortion during the transient into account. Also, on the assumption that the damping in the test motor could be

ignored, the mechanical model used in the implementation of the extrapolative method was for an inertial load only.

It is quite clear that the extrapolative method deserves further development and testing. Based on the results, it seems that the most profitable course for refinement would be to add more sophistication to the mechanical model assumed in the method. From the practical point of view, the method would benefit from an improved ability to detect when an estimate is spurious. This would allow an iterative procedure to ignore those parameters that the extrapolative method might have identified correctly, and optimize in a reduced parameter space. Some refinements are probably not worthwhile, since any estimate produced by the extrapolative method that is close enough to the minimum for a Newton-type method to have near-quadratic convergence is essentially as good as the true answer.

6.2 Modified least squares

The performance of the modified least squares method on simulated data was quite good. On simulated data with slip supplied, the method gave parameter estimates that were typically good to three significant figures. With no slip data, performance was not quite as good. Overall, the estimates without slip data might be characterized as good to two significant figures. The estimates without slip data were sometimes quite accurate, however. For example in the 500 hp motor test, with the exception of X_m, the parameters were characterized to three significant digits. One possible explanation is the stopping criterion used for the method. The stopping criterion is the test used to determine when to stop refining an estimate. In the implementation used to obtain the results in Chapter 5, the stopping criterion was set to terminate the method when a certain small fraction of the norm of the original error was achieved. It may be that the estimates in Table 5.2 could be improved by revising the stopping criterion.

Performance of the modified least squares method on real data was quite good. Although the true parameters of the motor were not available for comparison, the

agreement between the measured data and simulation using the estimates is very close. Of course, only one data set and one motor was characterized; more experimentation will be necessary to determine if the result presented in Chapter 5 is typical or not. As discussed in Chapter 5, it seems completely reasonable that the slight errors that are present in the fit between model and experiment are due to deficiencies in the model.

The best stopping criterion, enhancement of the mechanical model, a characterization of the noise performance, and analysis of the convergence of the modified least squares method remain topics for research.

Appendix A

Power quality prediction

The following document describes an extension of work initiated in [15]. This also appeared in [23]. It is included as a thorough, complete example of the techniques developed in Chapter 1.

A.1 Power Quality Prediction

This section describes a system for estimating the parameters of a simple model of an electric utility outlet using a transient measurement. Parameters of the utility model are estimated using data collected by the prototype. Nonlinear, frequency dependent effects observed in previous work in this area are accounted for with a physically based model. The performance of the entire system is demonstrated by comparison of measured and predicted line voltage distortion during current transients created by a laser printer.

A.1.1 Background

From a service outlet, the electrical utility can be modeled as a sinusoidal voltage source in series with an inductor and a resistor. In a commercial or industrial building, impedances seen at the "user interface" arise predominantly from an upstream transformer, protection circuitry, and cabling. Harmonic currents generated by loads

flow through these impedances, creating voltage drops that result in a distorted voltage waveform at the service outlet.

In [1], the authors present an ingenious technique for determining the local apparent impedance of the electrical utility service. The impedance is identified by briefly closing a capacitor across the electrical service at a precise point in the line voltage waveform. The shape and decay of the transient capacitor current in the resulting RLC circuit can be used to estimate the line impedance.

Here we reformulate the technique in the DESIRE (Determination of Electrical Supply Inductance and Resistance) system for characterizing a local electrical service. This new system offers several advantages. The hardware features a power-level, digitally programmable test capacitor, a precision switch with a programmable firing angle, and a data collection interface. The flexibility of the DESIRE hardware, in particular its digital control, allows it to collect the data required to accurately characterize the local electrical service. The software uses methods we describe here to estimate the parameters of the local distribution service, given the transient test data generated by the DESIRE hardware. The estimation method is particularly attractive because it does not require calibration of the parasitics introduced by the DESIRE hardware. This paper also develops a model, motivated by theory, to account for the measured increase in utility resistance with increasing test frequency observed in [1]. The model is used to predict the characteristics of the service impedance over a wide range of frequencies, given a limited number of test measurements.

In [13] and [14], a transient event detector for nonintrusive load monitoring was introduced, which can determine in real time the operating schedule of the individual loads at a target site, strictly from measurements made at the electric utility service entry. With knowledge of the impedances of the distribution network in a building, collected by a one-time application of the DESIRE system, the nonintrusive load monitor could in many cases predict power quality (i.e., the extent of local voltage waveform distortion) using only information from the service entry. We conclude with a demonstration of this technique by predicting the local voltage waveform distortion created by a laser printer.

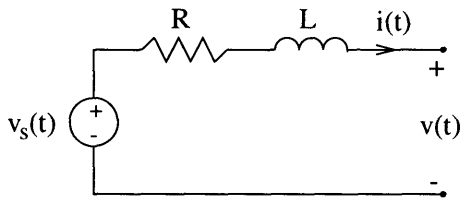

Figure A-1: Utility model.

A.1.2 Service Model

Consider a single phase, line-to-neutral connection to the electric utility. Electrical loads are presumed to be connected to the secondary of a single phase transformer driven at its primary by a stiff AC voltage source. Figure A-1 shows a model for such a connection to the electric utility [1]. With certain simplifying assumptions, the resistance R and inductance L represent the composite impedances of cabling, protection circuitry, and the dominant transformer in the service stream. If the transformer is represented by a T-circuit model [25], the circuit in Fig. A-1 can be developed as a Thevenin equivalent by assuming that $X_{mag} > (R_p + X_p)$, where X_{mag} and $R_p + X_p$ represent the reflected magnetizing and series primary impedances (series resistance and leakage inductance), respectively.

For low frequency power quality estimation, we are concerned with a frequency range from fundamental (60 Hz) to about 16th harmonic. Over this frequency range, the resistance R in Fig. A-1 is a nonlinear, increasing function of frequency. The inductance in the model arises in part from the primary and secondary leakage inductances in the transformer, and also from stray fields around the cabling and conduits. The inductance is relatively independent of frequency. We assume that other parasitic components, especially inter- and intra-winding capacitances, have a negligible effect at the frequencies of interest, and are therefore ignored. Extensions of the techniques in this paper to other situations, including a full three phase service, are possible.

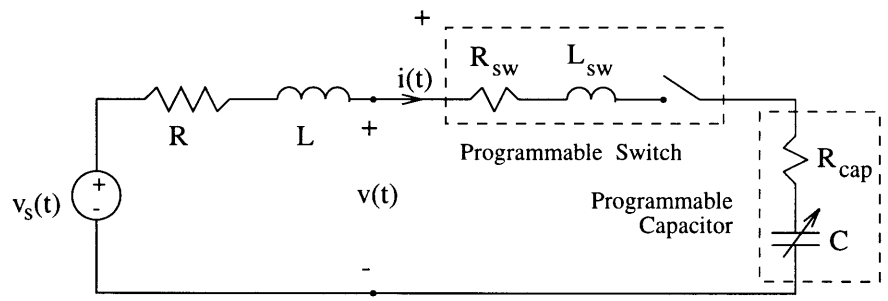

Figure A-2: DESIRE hardware.

A.1.3 DESIRE Prototype

To determine the effects of load currents on the voltage waveform, the parameters R and L of the utility model in Fig. A-1 must be identified. The DESIRE prototype connects a capacitive load to the utility service and analyzes the resulting transient waveforms to determine these parameters. A circuit model of the system, including the electrical load created by the DESIRE prototype, is shown in Fig. A-2.

The switch in Fig. A-2 is controlled by a timing circuit that is phase-locked to the AC input voltage waveform. The firing angle of the switch can be programmed with 10-bit resolution, i.e., a resolution of one part in 1024 parts of a line cycle. Varying the firing angle allows the magnitude of the transient current $i(t)$ to be kept within the range of the current sensor. The value of the capacitor in Fig. A-2 is also digitally programmable, with a resolution of seven bits. This is accomplished with a parallel array of seven fixed capacitors whose relative values are powers of two. The programmability of the switch firing angle and capacitor value in the DESIRE hardware permits the automated examination of transient current waveforms for a wide range of service power levels. It also facilitates rapid, computer-based data collection at a variety of transient frequencies. As will be shown in the following sections, accurate characterization of the frequency dependence of R in Fig. A-1 depends on the ability to collect data at different transient frequencies.

For accurate power quality prediction, the estimation method must determine the impedances of the utility model independently of parasitic impedances in the DESIRE

hardware. As modeled in Fig. A-2, the DESIRE programmable switch contains parasitic resistance and inductance. The load shown in Fig. A-2 is the the programmable capacitor, modeled with an equivalent series resistance. In the DESIRE prototype, no extreme effort was expended to minimize these parasitic elements or calibrate the test capacitances, since the parasitics are likely to depend on time, temperature and other environmental factors. The parameter estimation scheme described in the next section does not depend on any *a priori* knowledge of this kind.

A.1.4 Parameter Estimation and Extrapolation

If measurements are made of $v(t)$, $v_s(t)$ and $i(t)$, as indicated in Fig. A-2, the unknown parameters R and L of the utility model can be estimated. Because the parameters R and L are unknown,the voltage $v_s(t)$ can only be measured when $i(t)$ is zero, which precludes direct measurement during the transient. For practical purposes, we assume that $v_s(t)$ is shift invariant over a small integer multiple n of the fundamental period T, i.e. $v_s(t) \approx v_s(t + nT)$. By collecting reference waveforms immediately before performing a transient test, which is easily accomplished with a computerized data acquisition system, the shift nT above can be made quite small. Note that the requirement that $i(t) = 0$ does not imply that the transformer is unloaded. The transformer load need only be in steady state over the short interval required to collect $v_s(t)$ and perform the transient test. Measurement of $i(t)$ and $v(t)$ during the transient is straightforward.

A.1.5 Identification of the parameters R and L

Assuming that $v_s(t)$ is shift invariant as above, the parameters R and L constrain the signals $v(t)$, $i(t)$, and $v_s(t)$ according to the relationship in Eqn. A.1. In the following, p represents the differentiation operator $\frac{d}{dt}$:

$$v_s(t) - v(t) = (R + Lp)i(t). \tag{A.1}$$

The parameters R and L could be found directly from Eqn. A.1 if the continuous

time current waveform were available and could be differentiated accurately. The measured data, however, consists of the samples $i(nT_s)$, $v(nT_s)$, and $v_s(nT_s)$, where T_s is the sampling period. We eliminate the problems associated with measuring or approximating the derivatives in Eqn. A.1 by introducing the causal, "low-pass" operator λ, with $\tau > 0$, [10]:

$$\lambda = \frac{1}{1 + p\tau}. \tag{A.2}$$

Solving Eqn. A.2 for p, we obtain the following:

$$p = \frac{1 - \lambda}{\lambda\tau}. \tag{A.3}$$

Equation A.1 can be reformulated by substitution with Eqn. A.3 to produce a linear least squares tableau that can be used to estimate R and L:

$$\begin{pmatrix} [\tau\lambda(v_s - v)](t) \\ [-\tau\lambda i](t) \end{pmatrix}^T \begin{pmatrix} \hat{\beta}_1 \\ \hat{\beta}_2 \end{pmatrix} = i(t), \tag{A.4}$$

where $\hat{\beta}_1$ and $\hat{\beta}_2$ are estimates of $\frac{1}{L}$ and $\frac{R}{L}$, respectively. The notation $[\lambda i](t)$ indicates the row vector $([\lambda i](T), [\lambda i](2T)...[\lambda i](NT))$, where $[\lambda i](t)$ is the result of applying λ to $i(t)$ at time t. Although λ is a continuous time operator, we have found that it can be applied off-line to linear or zero-order hold interpolations of the finely sampled quantities with little error. It is desirable to apply λ to sampled data for reasons of implementation. The time constant τ associated with λ must be determined by the user. The time constant should be chosen to preserve information content, and also so that the effects of noise in the regressors and the errors associated with interpolation of the sampled data are minimized. All of the results presented here were obtained using $\tau = .002$s. In practice, a relatively wide range of values of τ produces satisfactory estimates.

Equation A.4 is arranged to minimize the bias in the parameter estimate introduced by disturbances in the measurements. In particular, the regressors are picked so that they are low-pass. Unless disturbances are pathologically low-pass, the error in the filtered regressors will be substantially uncorrelated to the unfiltered right-hand side.

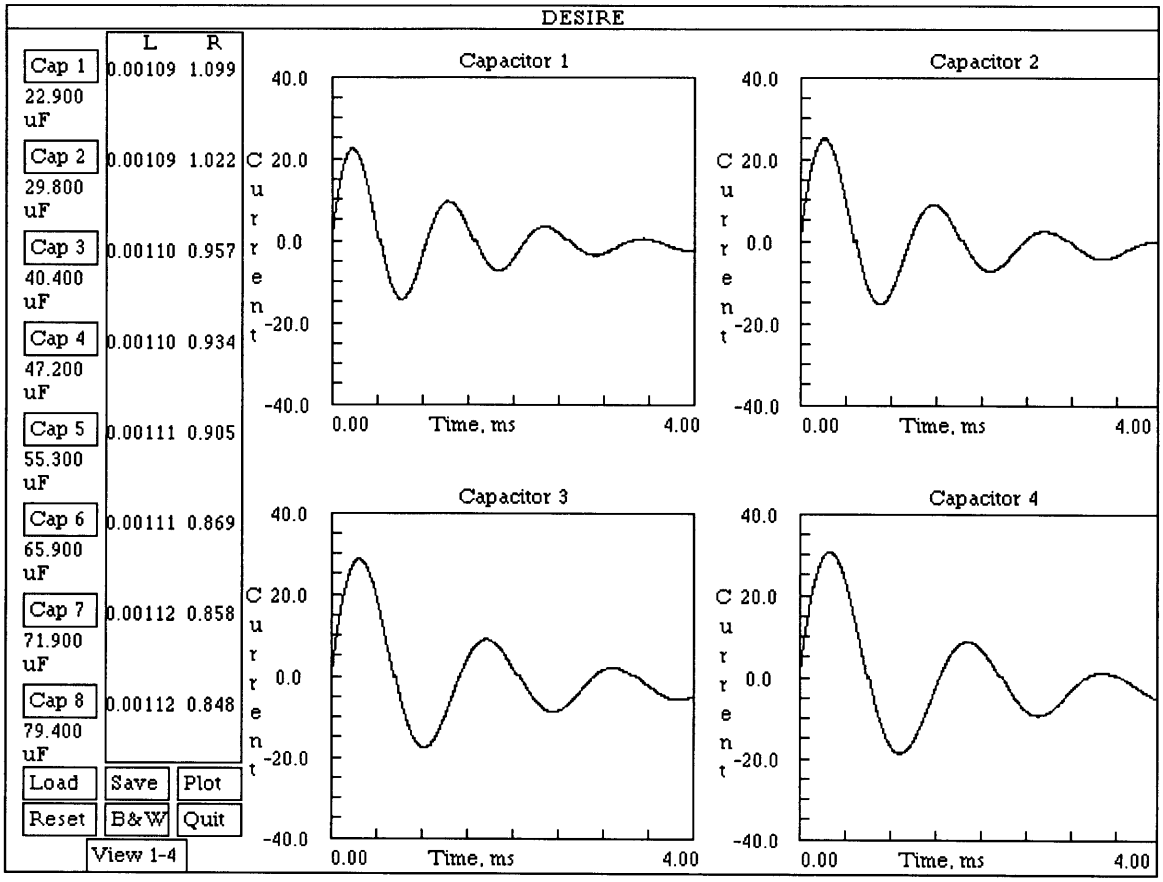

Figure A-3: Screen interaction with DESIRE prototype.

If the disturbances are symmetrically distributed and uncorrelated to the regressors, for large N the estimates will be unbiased. A more thorough discussion of the role of noise in this method can be found in [10].

A.1.6 Estimating transient frequency

It is important to associate a frequency f with the parameter R found by the methods outlined above, as R is a function of frequency. We account for the parasitic elements in Fig. A-2 by defining quantities $R_{tot} = R + R_{sw} + R_{cap}$ and $L_{tot} = L + L_{sw}$. The transient frequency f (in Hertz) is given by Eqn. A.5 in terms of these new parameters.

$$f = \frac{1}{2\pi}\sqrt{\frac{1}{L_{tot}C} - \frac{R_{tot}^2}{4L_{tot}^2}} \tag{A.5}$$

One approach to find the estimate $\hat{f}$, therefore, is to first determine the unknown parameters R_{tot}, L_{tot}, and C and then use Eqn. A.5. This technique is preferable to timing zero crossings, for example, because it is relatively insensitive to noise at the zero crossings and is independent of the steady state response.

To find the estimates $\hat{R_{tot}}$, $\hat{L_{tot}}$, and $\hat{C}$, we employ the λ-operator substitution technique to the equation relating $i(t)$ to $v_s(t)$ in Fig. A-2:

$$v_s(t) = (R_{tot} + L_{tot}p + \frac{1}{Cp})i(t). \tag{A.6}$$

Substitution to eliminate p yields the following equation in terms of the operator λ and its parameter τ.

$$\begin{pmatrix} [\tau(\lambda - \lambda^2)i](t) \\ [\tau^2\lambda^2 i](t) \\ [\tau(\lambda^2 - \lambda)v_s](t) \end{pmatrix}^T \begin{pmatrix} \hat{\alpha_1} \\ \hat{\alpha_2} \\ \hat{\alpha_3} \end{pmatrix} = [-i + 2\lambda i - \lambda^2 i](t) \tag{A.7}$$

where $\hat{\alpha_1}$, $\hat{\alpha_2}$ and $\hat{\alpha_3}$ are estimates of $\frac{R_{tot}}{L_{tot}}$, $\frac{L_{tot}}{C}$, and $\frac{1}{L_{tot}}$, respectively. Equation A.7 is solved in a least-squares sense and the parameter estimates are used to compute the transient frequency f using Eqn. A.5.

A.1.7 Frequency dependence of R

In [1] and in the experiments in our laboratory, the apparent resistance R was observed to be an increasing function of the frequency f of the transient. Phenomena that could explain this observation include, for example, eddy currents induced in conductors adjacent to current carrying wires and skin effect in the wires themselves.

In [3] the change of resistance due to the skinning effect in a conductor with cylindrical geometry is given for $x \ll 1$ ("low" frequencies) as

$$\frac{R}{R_0} \approx 1 + \frac{x^4}{192} \tag{A.8}$$

where $x \propto \sqrt{f}$ and the constant of proportionality, given explicitly in [3], is related to

the physical properties and geometry of the conductor. R_0 is the DC resistance.

From [3], eddy currents in conductive materials adjacent to current carrying wires produce changes in effective resistance as in Eqn. A.9, which is valid for $\theta \ll 1$.

$$R \approx R_0 + 2\pi f L_0 \frac{\theta^2}{6} \tag{A.9}$$

Here, $\theta \propto \sqrt{f}$. Again, the constant of proportionality is geometry and material dependent, and can be found analytically for certain geometries.

Assuming that the constants relating x and θ to $\sqrt{f}$ are favorably scaled, Equations A.8 and A.9 suggest the following fitting function, with parameters R_0 and δ.

$$R(f) = R_0 + \delta f^2 \tag{A.10}$$

With several estimates $\hat{R}(f)$ made at different frequencies, a least-squares solution for the parameters $\hat{R}_0$ and $\hat{\delta}$ can be found which satisfies Eqn. A.10. Transient tests at different frequencies can be automatically conducted by the DESIRE system simply by programming a range of values for C. It may be possible, based on the value of δ, to determine frequencies above which Eqn. A.10 becomes invalid. However, this was not investigated in detail here because the purpose is to extrapolate the data to *lower* frequencies, and because the collected data at higher frequencies are well interpolated by Eqn. A.10.

A.1.8 Experimental Results

The test setup used to validate the DESIRE hardware and software consisted of a single phase 1 kVA isolation transformer connected between phase and neutral of a three-phase 60 Hz, 30 A per phase, electrical service. A relatively small transformer was chosen so that it could be removed from service and characterized independently during development. After the transformer was characterized using the DESIRE system, a laser printer with a base plate rating of 7.6 A at 115 VAC RMS was connected to the transformer. Given the laser printer's remarkable current waveform, we predict

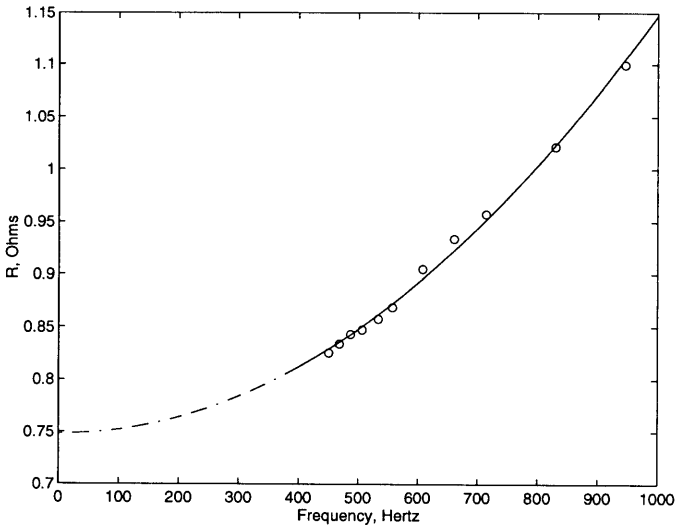

Figure A-4: Resistance R as a function of frequency f.

the voltage distortion due to the service parameters $R(f)$ and L.

Figure A-3 shows a screen interaction with the DESIRE system. DESIRE can be configured to conduct eight transient measurements on a utility connection automatically. The user selects nominal capacitor values which will be connected across the utility. The estimated capacitor values used during the tests are shown at the far left hand side of the screen in Fig. A-3. The transient currents during each of the experiments are plotted on the right hand side of the screen for four of the eight different capacitor values. The user may view either the first or second set of four transient plots. The estimated values of R and L for the the service model are displayed to the left of the transient current plots.

Fig. A-4 shows estimated resistance $\hat{R}$ as a function of transient frequency f (Hz). The solid line is the interpolation of the data according to the model of Eqn. A.10, and the dashed line shows the extrapolation of the model to lower frequencies. The estimated inductance $\hat{L}$ was 1.10 mH.

Fig. A-5 shows a transient current waveform drawn by the printer during operation.

In Fig. A-6, the measured and predicted line voltages are displayed. The data points show the measured voltage waveform, decimated for clarity. The solid line is a linear interpolation of the simulation results.

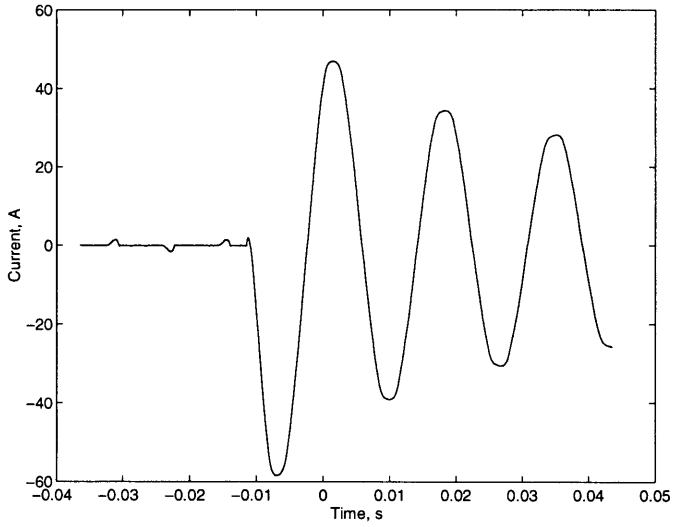

Figure A-5: Sample current transient.

The predicted data was obtained using the estimate $\hat{L}$, the extrapolation of Eqn. A.10 to 60 Hertz using the data in Fig. A-4, the measured current $i(t)$, and Eqn. A.1.

A.1.9 Conclusions

The DESIRE prototype and analysis software provide a flexible system for characterizing the effective impedance of a utility service connection. This information can be used for a variety of applications by utilities, and also commercial and industrial facilities managers. In [13], the nonintrusive load monitor was demonstrated to have the ability to disaggregate the operating schedule of individual loads given access only to the aggregate current waveforms at the service entry. With additional knowledge collected during a one-time (or at least infrequent) examination of the details of a building's wiring harness, the location of loads on the harness, and the service connection impedances as determined by the DESIRE system, the nonintrusive load monitor could provide continuous prediction of the local voltage waveform at points of interest. We have demonstrated the basis for this power quality monitoring technique in this paper.

The impedance calculations made by DESIRE could also be used, for example, to compute the available fault currents at a service connection. This information would

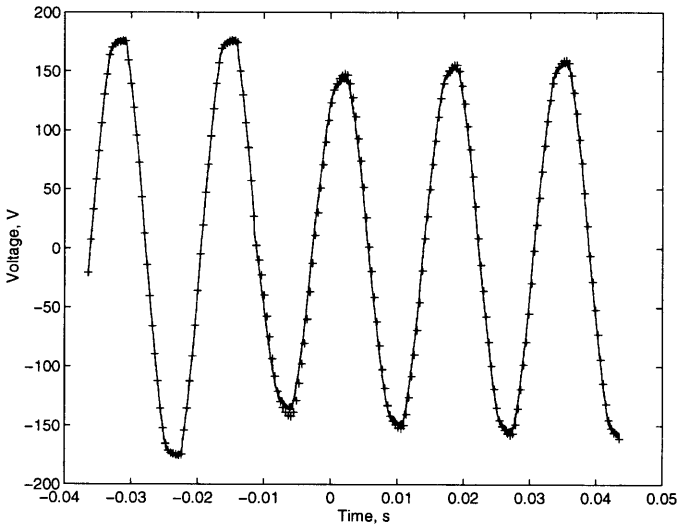

Figure A-6: Measured and predicted line voltage distortion.

help to verify the proper rating of protection circuitry in the wiring harness. During our experimentation, we have observed that careful, high sample rate examination of the transient current waveforms collected by DESIRE reveal additional, very high frequency ringing that we suspect can be attributed to neglected elements in the utility model, e.g., inter- and intra-winding capacitances. It is conceivable that additional information about the utility service impedance could be determined by DESIRE, which might be useful for determining the propagation of high frequency EMI or power line carrier modem signals.

A.2 MATLAB Source Code

This is the Matlab source code used for the analysis described above.

meister.m

This is the main driver program; it processes all the data files.

```
clear M;
```

```
M(1, :) = run2( 'tek00000.csv', 'tek00001.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(2, :) = run2( 'tek00002.csv', 'tek00003.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(3, :) = run2( 'tek00004.csv', 'tek00005.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(4, :) = run2( 'tek00006.csv', 'tek00007.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(5, :) = run2( 'tek00008.csv', 'tek00009.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(6, :) = run2( 'tek00010.csv', 'tek00011.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(7, :) = run2( 'tek00012.csv', 'tek00013.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(8, :) = run2( 'tek00016.csv', 'tek00017.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(9, :) = run2( 'tek00018.csv', 'tek00019.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(10,:) = run2( 'tek00020.csv', 'tek00021.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(11,:) = run2( 'tek00020.csv', 'tek00021.csv',
'ref1.csv', 'ref2.csv', 10.0);
M(12,:) = run2( 'tek00024.csv', 'tek00025.csv',
'ref1.csv', 'ref2.csv', 10.0);


f = M(:,1);
r = M(:,5);
l1 = M(:,3);
l2 = M(:,6);

rob2(r,f);

figure;

subplot(211),plot(f,r,'o');
v = axis;
v(4) = mean(l2)*max(r)/mean(r);
v(3) = mean(l2)*min(r)/mean(r);
subplot(212),plot(f,l2,'o'),axis(v);
```

run2.m

```
%
function qr = run2( f1, f2, f3, f4, Aset )

% Load up the data.
[t2,y] = tekload(f1);   % Current
K(:,1) = y(:);
[t2,y] = tekload(f4);       % subtract current DC offset setting.
K(:,1) = K(:,1) - y(:);
[t2,y] = tekload(f2);   % Vcap
K(:,2) = y(:);
[t2,y] = tekload(f3);   % input voltage waveform.
K(:,3) = y(:);

M = K(1:1000,:);
t = t2(1:1000);


% Zero out the stuff that happens before t = 0.0
for i = 1:size(M,1)
  if t(i) < 0.0
     M(i,:) = 0.0 * M(i,:);
  end;
end;

% Scale up the data according to probe settings.

M(:,1) = M(:,1) * Aset / .01;
M(:,2) = M(:,2) * 100 / .2;
M(:,3) = M(:,3) * 100 / .2;

% analyze the data.
tau = .0002;

qr = robme2(M,t,tau);
qr = qr(:)';
```

robme2.m

```
%
% Improved identification procedures for rob-meister's
% DESIRE box.
%
function qr = robme2(M,t,tau)

Volts(:) = M(:,3);
icap(:)  = M(:,1);
vcap(:)  = M(:,2);

%
% Now apply lambda operators.
%
li  = lsim(1,[tau,1],icap,t);
l2i = lsim(1,[tau,1],li,t);
lv  = lsim(1,[tau,1],Volts,t);
l2v = lsim(1,[tau,1],lv,t);

Q(:,1) = tau * ( li(:) -l2i(:));
Q(:,2) = tau*tau*l2i(:);
Q(:,3) = -tau*(lv(:) - l2v(:));
rhs(:) = -( icap(:) - 2*li + l2i );

%subplot(211),plot(li);
%subplot(212),plot(lv);

p = Q \ rhs;

C = p(3)/p(2);
L = 1/p(3);

R = p(1)/p(3);

omega = sqrt(1/(L*C)-R*R/(4*L*L)) / (2*pi);

qr(1) = omega;
qr(2) = R;
qr(3) = L;
qr(4) = C;
```

```
% Do RC

lvind = lsim(1,[tau,1],Volts(:)-vcap(:),t);
W(:,1) = tau*li(:);
W(:,2) = icap(:)-li(:);

px = W \ (tau*lvind(:));

qr(5) = px(1);
qr(6) = px(2);
```

rob2.m

This file produces the plot of resistance as a function of frequency.

```
function rob2( hrm, frqm)
clear global frq;
clear global hr;

frq = frqm;
hr  = hrm;

global frq;
global hr;

x0 = [1.0,1.0];
x = fsolve('fun', x0)
x

clf;
plot(frq, hr, 'bo');     % plot data points.
hold on;

clear global frq;
clear global hr;
frq = 400:1000;
hr = 0 * frq;
global frq;
global hr;
y = fun(x);
plot(frq,y);              % plot the fit.
```

```
clear global frq;
clear global hr;
frq = 0:400;
hr = 0 * frq;
global frq;
global hr;
y = fun(x);
plot(frq,y,'r-.');              % plot the extrapolation.

y(60)


%title('Resistance as a function of frequency');
ylabel('R, Ohms');
xlabel('Frequency, Hertz');
```

tekload.m

This file loads a Tektronix ".CSV" file.

```
function [x,y]=tekload(name)
%
%
%
fid=fopen(name,'r');
output=fscanf(fid,'%f,%f',[2,inf]);
x=output(1,:);
y=output(2,:);

fclose(fid);
```

checktime.m

This file is used to check alignment of data files collected at different times.

```
%
% Check synchronization signals graphically.
```

```
%
clf;
clear;


[t,edge] = tekload('tek00003.csv');
plot(t,edge,'y+');

hold;

[t,edge] = tekload('tek00006.csv');
plot(t,edge,'m+');

[t,edge] = tekload('tek00009.csv');
plot(t,edge,'c+');

[t,edge] = tekload('tek00012.csv');
plot(t,edge,'r+');

[t,edge] = tekload('tek00015.csv');
plot(t,edge,'g+');

[t,edge] = tekload('tek00018.csv');
plot(t,edge,'b+');

[t,edge] = tekload('tek00021.csv');
plot(t,edge,'w+');

[t,edge] = tekload('tek00024.csv');
plot(t,edge,'yx');

[t,edge] = tekload('tek00027.csv');
plot(t,edge,'mx');

[t,edge] = tekload('tek00030.csv');
plot(t,edge,'cx');

[t,edge] = tekload('tek00033.csv');
plot(t,edge,'rx');

[t,edge] = tekload('tek00036.csv');
plot(t,edge,'gx');

[t,edge] = tekload('tek00039.csv');
plot(t,edge,'bx');
```

```
[t,edge] = tekload('tek00042.csv');
plot(t,edge,'wx');

[t,edge] = tekload('tek00045.csv');
plot(t,edge,'yo');

[t,edge] = tekload('tek00048.csv');
plot(t,edge,'mo');

title('Synchronization Edge Alignment');
ylabel('Sync Edge (Volts)');
xlabel('Time (s)');
```

Appendix B

CSIM Simulation Environment

The following document describes a general purpose set of C routines designed to ease coding of simple simulations. Although not developed for this thesis work, it is included here because of the applicability to further work on this subject.

B.1 Introduction

`CSIM` is intended to provide a relatively robust, easy to use simulation environment for simulation of simple differential equations. Ideally, the user need only express the differential equation model of a system in C syntax to simulate the system.

This document describes how to translate a simple physical model of the electrical terminal behavior of a lightbulb into the procedures required by `CSIM`. C is an inherently flexible language; the discipline and structure needed to create readable and reliable programs comes from the programmer, not the language. This example should be regarded not only as documentation of `CSIM` itself, but also as a paradigm of the user coded routines.

`CSIM` contains portions of code which are an adaptation of the routines published in *Numerical Recipes in C* (Press, Flannery, et. al). Those routines are NOT the public domain and require the purchase of *Numerical Recipes in C* for licensed use.

B.2 General description of the integration procedure

`CSIM` is a numerical integrator based on a Runge-Kutta procedure. Local truncation errors are estimated at each step using a lower order embedded method, and the step size is adjusted so that the errors are bounded by a user selectable ϵ. The default is to restrict relative errors to $\epsilon = 1e - 6$, but the value can be changed by calling `setepsilon`. The variable step size algorithm is executed between evenly spaced "grid points". The idea is to simulate as closely as possible actual experimental conditions – for reasons of implementation, data are usually collected at a constant sampling rate determined *before* the experiment is performed and independent of the local frequency content of the results.

There are two procedures that the user must code to perform a simulation. The third is optional.

- Derivatives as a function of states and inputs
- Mapping of states, etc. to outputs
- A routine introducing discontinuous changes in state/topology

In translating a model to C code, then, the state matrix would be expressed in the first procedure, the output matrix in the second procedure, and features like switches and diodes and relays would be contained in the third procedure. We will consider a simple example that requires all three routines.

B.3 A physical description of a lightbulb

An incandescent light bulb is a thin wire surrounded by a protective atmosphere contained in a glass envelope. When the bulb is connected to the mains, the resulting current flow rapidly heats the filament until the electrical power deposited in the filament equals the radiative flux emanating from the bulb. We will model the filament as a black body, with the consequence that the radiant power $P_{rad} = a_1 T^4$. Here T is the filament temperature. The electrical power deposited in the filament is simply

$P_e = IV$. We assume that the thermo and thermoelectric constitutive relations of the filament itself are $P_{fil} = a_2 \frac{dT}{dt}$ and $R_{fil} = a_3 + a_4 T$. Specifically, we assume that a heat capacity is a sufficient thermal description of the filament, and that the filament has a linearly increasing resistance as a function of temperature. Further, suppose (mostly for purposes of illustration) that the filament might undergo an unstable sublimation process if the filament exceeds a certain critical temperature. This is not an unreasonable assumption, because a "hot spot" on the filament will sublimate faster than the rest of the filament, and subsequently will get even hotter.

Differentiating the first law of thermodynamics (conservation of energy) with respect to time, we have $P_{fil} = P_e - P_{rad}$, or, equivalently.

$$a_2 \frac{dT}{dt} = \frac{V^2}{a_3 + a_4 T} - a_1 T^4 \tag{B.1}$$

This is our equation of state, which must be translated to C in the first routine required by `CSIM`.

In the power monitoring scenario, $V = V(T) = 179 \sin 377t$, and we are interested in the current. Therefore the output relationship is

$$I(t) = \frac{V(t)}{a_3 + a_4 T}. \tag{B.2}$$

Furthermore, we will model the sublimation as severing the filament when the filament temperature exceeds a certain critical temperature a_5. That is, $I(t_0 + t) = 0$ for all positive t if $T(t_0) > a_5$.

B.4 C Code

The C code to implement the model described above is reproduced and explained below. A clean copy without additional comments and editorializing is in Section 7.

```
/*
** Simulation of a lightbulb.
*/
#include <stdio.h>
```

```
#include <math.h>
#include "integrate.h"
#include "csim.h"
```

The include files in quotes are part of the `CSIM` routines and must be in the current working directory. The other include files are required to use functions like `printf` and `cos`. The sensible way to organize the list of include files is with the standard includes (with "<>") first and the "local" includes next.

```
/* Prototypes. */
void burnout(double , double , double , double );
void derivs(double , double , double , double );
void output(double , double , double , double );
```

Next come the "prototypes." Prototypes are declarations of a function's expected return and argument types that allow the compiler to do a more thorough job of making sure that the functions are used as anticipated. All functions should be prototyped.

```
static char *output_file = "results.out";
```

Here is the main program. The operating system starts the thread of execution here.

```
int main(void)
{
  FILE *f = fopen(output_file, "w");
  static double s[10],ds[10],aux[10];
  static double par[10];

  if(f == NULL) {
    printf("Problem opening file...\n");
    return 1;
```

```
}
```

In this first part of main, a file to put the results in is opened and arrays to store the state (s), derivative (ds), some auxiliary variables (aux) and the parameters of the model (par) are created. Although we only have 1 state variable, there is no harm done in being on the safe side. Note that the array you allocate here must have **AT LEAST** $N+1$ elements if there are N states, because the convention used in the simulator code is to begin indexing arrays at 1 not 0. The auxiliary variables are available for the communication of any interesting quantities between the three routines. In the lightbulb example, the auxiliary variables are used to store the state of the filament (broken or not) and the current. Whenever a file is opened, check to be sure that the pointer is valid.

```
#ifndef QUIET
  printf("Simulating lightbulb...\n");
#endif

                                                                    30
  filestore_init(f);
```

You might want to run the simulator later without seeing a lot of messages (for example, if you want to run it in a script). That's the reason for the preprocessor directives on lines 27 and 29. If you have debugging "printf's", bracket them with an `#ifdef DEBUG`. The function call on line 31 just tells the informs the file storage system of the file we'd like to use. On line 111, `filestore` will write to this file.

```
  /* Some parameters. */
  par[1] = 1e-8;
  par[2] = 1.0;
  par[3] = 2;
  par[4] = .05;
  par[5] = 350;

  /* Initialize all routines. */                                    40
  derivs(INITIALIZE,par,ds,aux);
  output(INITIALIZE,par,ds,aux);
```

```
  burnout(INITIALIZE,par,ds,aux);
```

Here's where the three functions that must be written by the user are initialized. The second argument, which is normally a state vector, is in this special circumstance a vector with any initialization information that the routines might need. In this case, the routines only need to be informed of the parameters to use for the simulation.

```
  simulate(s,ds,aux,1,.0002,2000,output,derivs,burnout);
  fclose(f);

#ifndef QUIET
  printf("Results are in %s...\n", output_file);
  printf("Output variables are 1:Time    2:Temperature    3:Current\n");
#endif
  return 0;
}
```

Next, we call the `simulate` procedure that's part of the `CSIM` package. The arguments (from left to right) are the state vector, the derivatives, the auxiliary variables, the number of state variables, the grid time step, the number of points to simulate, and the output mapping function, the derivatives function, and the discontinuous state/topology change function. If you don't need a function like `burnout` for your simulation, you can pass an appropriately cast NULL instead. After simulating, the output file is closed and the program exits.

```
#define I       (aux[1])
#define fil     (aux[2])
#define T       (s[1])
#define dTdt    (ds[1])
```

These defines just establish a standard map between our equations and the state and auxiliary vectors. We declare here, for example, that the temperature T is to be stored in the state $s[1]$.

```
/*
** Derivatives
*/
```

```
void derivs(double t, double s[], double ds[], double aux[])
{
  static double a1,a2,a3,a4;

  double Pe, Prad, V;

  if(t == INITIALIZE) {
    a1 = s[1];                                                    70
    a2 = s[2];
    a3 = s[3];
    a4 = s[4];
    return;
  }
```

Here is the beginning of the derivatives routine, one of the functions that must be be written by the user. The arguments of the three user written routines are all the same; time, states, derivatives, and auxiliary variables. Notice the initialization code. This is where the parameters passed as states (on line 40) are "remembered" by the derivatives function. The static variables a1..a4 are not allocated off the stack; they retain their values between calls and have local scope.

```
  if(t == ZEROSTATE) {
    fil = 1;                  /* Filament is intact          */
    I   = 0;                   /* No current                   */
    T   = 20.0;                /* Start at room temperature   */   80
    return;
  }
```

In this segment of code, the derivatives routine is asked to set the state vector to its initial conditions. You could do this in main(), if you so desired, and then this segment of code would do nothing but return.

```
  V = 179*sin(377*t);         /* Excitation                  */

  if(fil) {
    I = V/(a3+a4*T);          /* Constitutive law */
    Pe = I*V;
  } else {
    Pe = 0.0;                 /* Broken filament == No Pe          */   90
```

```
    I = 0.0;
  }

  Prad = a1*T*T*T*T;          /* Stefan-Boltzman law                          */
  dTdt = (Pe-Prad)/a2;        /* Heat/Temp constitutive relation + 2nd law    */
}
```

The rest of the derivatives routine is nothing more that the equations that were developed for the lightbulb. If you believe in physics and the approximations that were made, we should get something that looks just like a lightbulb.

```
/*
** Output mapping                                                    100
*/
void output(double t, double s[], double ds[], double aux[])
{
  static double col[5];

  if( t == INITIALIZE) return;
```

Here's the output routine. Note that this particular output function doesn't require any initialization information, but it still detects the `t == INITIALIZE` condition. This is important. If you don't do it, your output file will have spurious data on the first line.

```
  col[1] = t;
  col[2] = T;
  col[3] = I;                                                       110

  filestore(col, 3);
}
```

The rest of the output procedure is very simple. It simply puts the time in the first column, the temperature in the second column, and the current in the third column. Note that we use the current which was already compute for use in the derivs() function. It is communicated to the output function via the auxiliary variables. `filestore` sends the row out to the file pointer we opened up in main(). The second argument to `filestore` is the number of columns.

```
/*
** Discontinuous state/model changes
*/
void burnout( double t, double s[], double ds[], double aux[])          120
{
  static double a5;

  if( t == INITIALIZE) {
    a5 = s[5];
    return;
  }
```

Here's the "discontinuous state/model changing routine". Very simply, this is just a function which is guaranteed to be called only on the grid. The user supplied function `derivs()`, in contrast, can be called at any time. You can use this routine to set "initial conditions" in the middle of a simulation. If you have states that are not smooth, like ideal diodes, you can approximately simulate the system by putting your switching logic here.

```
  if(T > a5) {
    fil = 0;                    /* Filament is broken */                130
    I = 0.0;                    /* no current         */
  }
}
```

In this case, we check to see if the bulb temperature has exceed some arbitrary level. At this level, it is postulated that the filament's electrical constituitive relation will become discontinuous i.e. the bulb will burn out. This state of affairs is communicated back to the derivs() routine via the auxiliary variables.

B.5 Make File

In order for the `CSIM` source code to be appropriately compiled with the file contain the user written description of the simulation, a make file should be used. The lightbulb simulator can be compiled with the command `make -f lightbulb.make`, which produces an executable file call `lightbulb` in the current working directory. The make file `lightbulb.make` follows.

```
lightbulb : lightbulb.o integrate.o csim.o nrutil.o
    gcc -o lightbulb lightbulb.o integrate.o csim.o nrutil.o -lm

lightbulb.o: lightbulb.c
    gcc -c -Wall lightbulb.c

nrutil.o:  nrutil.c nrutil.h
    gcc -c nrutil.c

csim.o: csim.c csim.h
    gcc -c csim.c

integrate.o: integrate.c integrate.h
 gcc -c integrate.c
```

To make your own make file, simply replace every instance of `lightbulb` with `foo` where foo is the name of the .c file containing your simulation.

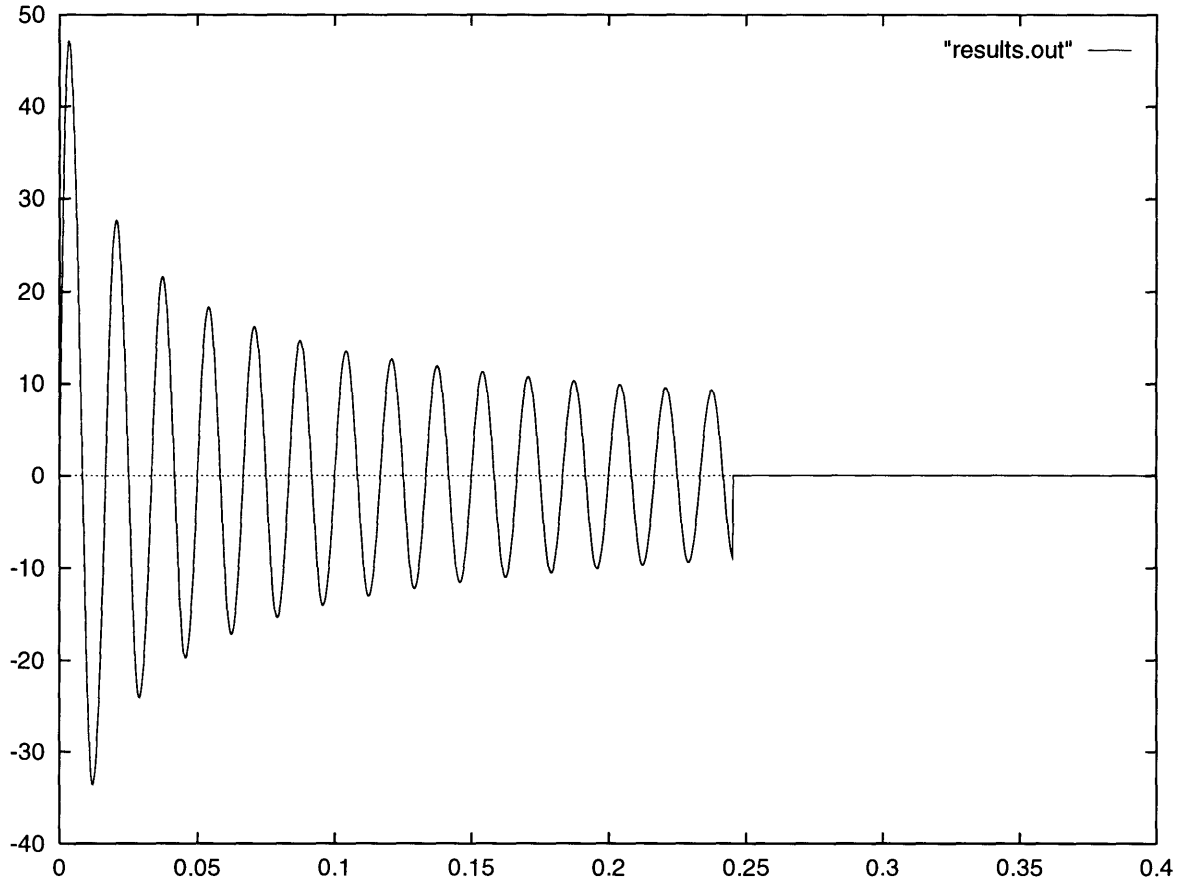

Figure B-1: Simulated light bulb current

B.6 Results

The program `lightbulb` made with the model, code, and make file above simulates correctly and produces some plausible waveforms. The current as a function of time is shown in Figure B-1. The filament temperature is shown in Figure B-2. The point at which the filament "burns out" is clear.

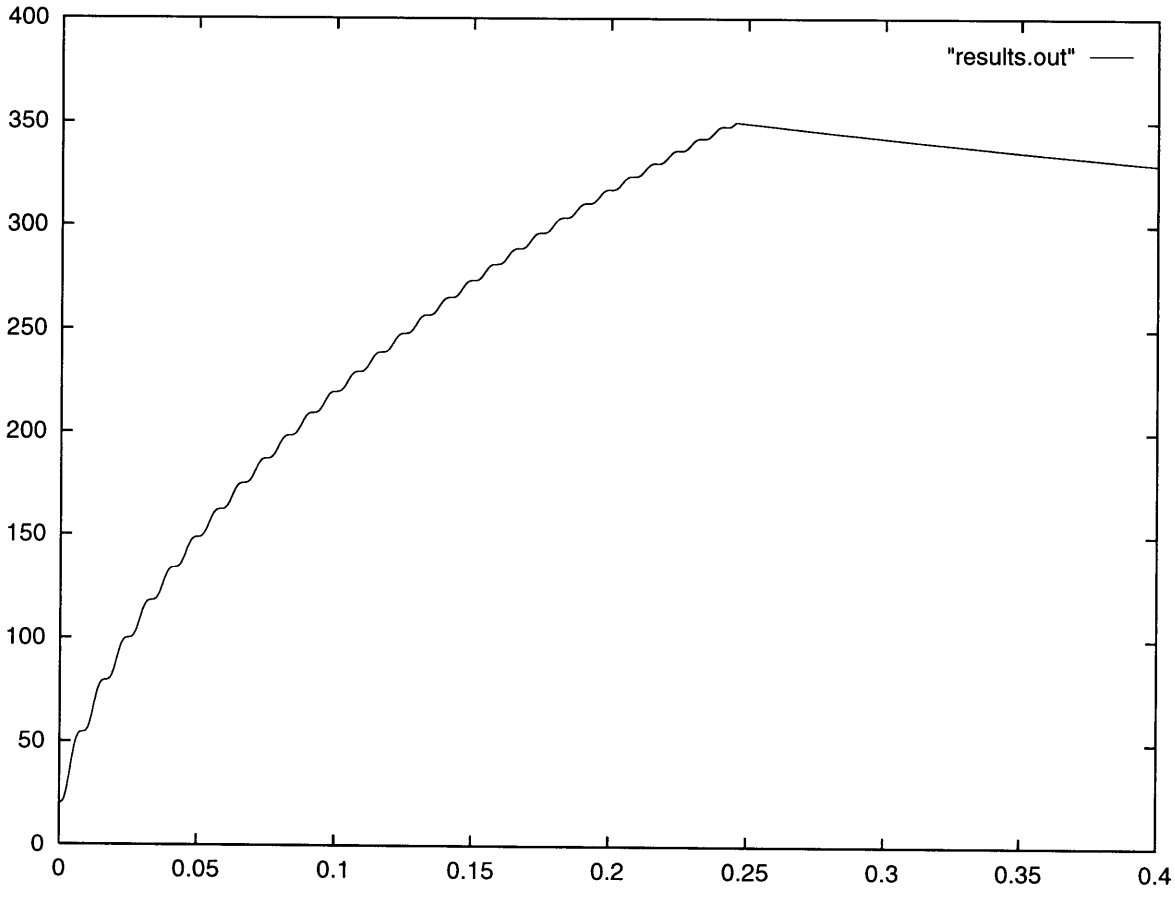

Figure B-2: Simulated light bulb temperature

B.7 lightbulb.c

```
/*
** Simulation of a lightbulb.
*/
#include <stdio.h>
#include <math.h>
#include "integrate.h"
#include "csim.h"

/* Prototypes. */
void burnout(double, double, double, double);
void derivs(double, double, double, double);
void output(double, double, double, double);

static char *output_file = "results.out";

int main(void)
{
  FILE *f = fopen(output_file, "w");
  static double s[10],ds[10],aux[10];
  static double par[10];

  if(f == NULL) {
    printf("Problem opening file...\n");
    return 1;
  }

#ifndef QUIET
  printf("Simulating lightbulb...\n");
#endif

  filestore_init(f);

  /* Some parameters. */
  par[1] = 1e-8;
  par[2] = 1.0;
  par[3] = 2;
  par[4] = .05;
  par[5] = 350;

  /* Initialize all routines. */
  derivs(INITIALIZE,par,ds,aux);
  output(INITIALIZE,par,ds,aux);
```

```
  burnout(INITIALIZE,par,ds,aux);

  simulate(s,ds,aux,1,.0002,2000,output,derivs,burnout);
  fclose(f);

#ifndef QUIET
  printf("Results are in %s...\n", output_file);
  printf("Output variables are 1:Time    2:Temperature    3:Current\n");
#endif
  return 0;
}
#undef N


#define I        (aux[1])
#define fil      (aux[2])
#define T        (s[1])
#define dTdt     (ds[1])

/*
** Derivatives
*/
void derivs(double t, double s[], double ds[], double aux[])
{
  static double a1,a2,a3,a4;

  double Pe, Prad, V;

  if(t == INITIALIZE) {
    a1 = s[1];
    a2 = s[2];
    a3 = s[3];
    a4 = s[4];
    return;
  }

  if(t == ZEROSTATE) {
    fil = 1;                /* Filament is intact          */
    I   = 0;                /* No current                  */
    T   = 20.0;             /* Start at room temperature   */
    return;
  }

  V = 179*sin(377*t);       /* Excitation                  */
```

```
  if(fil) {
    I = V/(a3+a4*T);          /* Constitutive law */
    Pe = I*V;                                                    90
  } else {
    Pe = 0.0;                 /* Broken filament == No Pe       */
    I = 0.0;
  }

  Prad = a1*T*T*T*T;          /* Stefan-Boltzman law                  */
  dTdt = (Pe-Prad)/a2;        /* Heat/Temp constitutive relation + 2nd law  */
}


                                                                 100
/*
** Output mapping
*/
void output(double t, double s[], double ds[], double aux[])
{
  static double col[5];

  if( t == INITIALIZE) return;

  col[1] = t;                                                    110
  col[2] = T;
  col[3] = I;

  filestore(col, 3);
}



/*
** Discontinuous state/model changes                            120
*/
void burnout( double t, double s[], double ds[], double aux[])
{
  static double a5;

  if( t == INITIALIZE) {
    a5 = s[5];
    return;
  }

                                                                 130
```

```
  if(T > a5) {
    fil = 0;                    /* Filament is broken */
    I = 0.0;                    /* no current         */
  }
}
```

B.8 Integration and Utility Codes

The following code may contain portions derived from Numerical Recipes

```
/*
   csim.c

   C Simulation code.

   Author: Steven R. Shaw
   Date  : 9/1/96

   Note all revisions after 26 Dec 1996 below:


*/

#include <stdio.h>
#include "integ.h"
#include "csim.h"

/* Prototype */
void filestore_(double [], int , FILE *);

/* Relative precision */
static double __eps  = 1e-6;

void simulate(double i[],    /* State Vector.             */
      double di[],   /* Derivative Vector.        */
      double aux[],  /* Auxilliary Variables.     */
      int n,         /* Number of states.         */
      double H,      /* Grid.                     */
      long L,         /* Number of points.         */
      void (*store) (double, double [], double [], double[]),
      void (*derivs)(double, double [], double [], double []),
      void (*ongrid)(double, double [], double [], double [])
      )
{
  long k;
  int nok,nbd;

  (*derivs)(ZEROSTATE,i,di,aux);

  for(k = 1; k <= L; k++) {
    /* Evaluate derivatives on grid prior to storage. */
    (*derivs)((k-1)*H,i,di,aux);

    /* Do something on the grid. */
    if( ongrid != NULL )
```

```
      (*ongrid)((k-1)*H,i,di,aux);

    /* Store. */
    (*store)((k-1)*H,i,di,aux);

    /* Integrate a step */
    odeint(i, aux, n, (k-1)*H, (k)*H, __eps, H,
   0.0, &nok, &nbd, derivs, rkqs);
  }
}


/*
   Interface to set the relative precision.
*/
void setepsilon( double precision )
{
  __eps = precision;
}


/*
   Initialize file storage tools
*/
void filestore_init(FILE *f)
{
  filestore_(NULL,0,(FILE *)f);
}

void filestore(double i[], int n)
{
  filestore_(i,n,(FILE *)NULL);
}

/*
** Just pass a vector to me.
*/
void filestore_(double s[], int n, FILE *M)
{
  static FILE *z;
  int j;

  if(M == NULL) {
    for(j = 1; j < n; j++)
      fprintf(z, "%.4e\t", s[j]);
      fprintf(z, "%.4e\n", s[n]);
  } else z = M;
}
```

```
/*
** csim.h
**
**
** Note any changes after 26 Dec 1996 Below:
**
** * Added definition of M_PI since C6 include files do
** not define it... Jan 1997.
**
*/

void simulate(double i[],
      double di[],
      double aux[],
      int n,
      double H,
      long L,
      void (*store)(double,double [], double [], double []),
      void (*derivs)(double, double [], double [], double []),
      void (*ongrid)(double, double [], double [], double []));

void filestore_init(FILE *f);
void filestore(double row[], int n);
void setepsilon(double eps);

#define INITIALIZE  (-1)
#define ZEROSTATE   (-2)


/* Make C6 compliant.  SRS */
#ifndef M_PI
#define M_PI (3.1415926535)
#endif
```

The following code may contain portions derived from Numerical Recipes

```
/*
   integ.c

   THIS FILE CONTAINS CODE FROM NUMERICAL RECIPES
   NUMERICAL RECIPES COPYRIGHT APPLIES

   SRS

   List all revision after 26 Dec 1996 below:


*/

#include <math.h>
#define NRANSI
#include "nrutil.h"
#include "integ.h"

#define FMIN(a,b)  (((a) < (b)) ? (a) : (b))
#define FMAX(a,b)  (((a) > (b)) ? (a) : (b))
#define SIGN(a,b) (((b) < 0) ? (-(a)) : (a))

#define vector(a,b)          dvector(a,b)
#define free_vector(a,b,c)   free_dvector(a,b,c)

#define MAXSTP 10000
#define TINY 1.0e-30

/* Prototype */

void rkck(double y[],
  double dydx[],
  double aux[],
  int n,
  double x,
  double h,
  double yout[],
  double yerr[],
  void (*derivs)(double, double [], double [], double []));


/* Code starts here */

void odeint(double ystart[],
    double aux[],
    int nvar,
    double x1,
    double x2,
    double eps,
    double h1,
```

```
    double hmin,
    int *nok,
    int *nbad,
    void (*derivs)(double, double [], double [], double []),
    void (*rkqs)(double [],
 double [],
 double [],
 int,
 double *,
 double,
 double,
 double [],
 double *,
 double *,
 void (*)(double, double [], double [], double [])))
{
  int nstp,i;
  double x,hnext,hdid,h;
  double *yscal,*y,*dydx;

  yscal=vector(1,nvar);
  y=vector(1,nvar);
  dydx=vector(1,nvar);
  x=x1;
  h=SIGN(h1,x2-x1);
  *nok = (*nbad) = 0;
  for (i=1;i<=nvar;i++) y[i]=ystart[i];

  for (nstp=1;nstp<=MAXSTP;nstp++) {
    (*derivs)(x,y,dydx,aux);
    for (i=1;i<=nvar;i++)
      yscal[i]=fabs(y[i])+fabs(dydx[i]*h)+TINY;

    if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x;
    (*rkqs)(y,dydx,aux,nvar,&x,h,eps,yscal,&hdid,&hnext,derivs);
    if (hdid == h) ++(*nok); else ++(*nbad);
    if ((x-x2)*(x2-x1) >= 0.0) {
      for (i=1;i<=nvar;i++) ystart[i]=y[i];

      free_vector(dydx,1,nvar);
      free_vector(y,1,nvar);
      free_vector(yscal,1,nvar);
      return;
    }
    if (fabs(hnext) <= hmin) nrerror("Step size too small in odeint");
    h=hnext;
  }
  nrerror("Too many steps in routine odeint");
}
#undef MAXSTP
#undef TINY
#undef NRANSI
```

```
#define NRANSI
#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}

void gaussj(double **a, int n, double **b, int m)
{
  int *indxc,*indxr,*ipiv;
  int i,icol,irow,j,k,l,ll;
  double big,dum,pivinv,temp;

  indxc=ivector(1,n);
  indxr=ivector(1,n);
  ipiv=ivector(1,n);
  for (j=1;j<=n;j++) ipiv[j]=0;
  for (i=1;i<=n;i++) {
    big=0.0;
    for (j=1;j<=n;j++)
      if (ipiv[j] != 1)
for (k=1;k<=n;k++) {
  if (ipiv[k] == 0) {
    if (fabs(a[j][k]) >= big) {
      big=fabs(a[j][k]);
      irow=j;
      icol=k;
    }
  } else if (ipiv[k] > 1) nrerror("gaussj: Singular Matrix-1");
}
    ++(ipiv[icol]);
    if (irow != icol) {
      for (l=1;l<=n;l++) SWAP(a[irow][l],a[icol][l])
for (l=1;l<=m;l++) SWAP(b[irow][l],b[icol][l])
    }
    indxr[i]=irow;
    indxc[i]=icol;
    if (a[icol][icol] == 0.0) nrerror("gaussj: Singular Matrix-2");
    pivinv=1.0/a[icol][icol];
    a[icol][icol]=1.0;
    for (l=1;l<=n;l++) a[icol][l] *= pivinv;
    for (l=1;l<=m;l++) b[icol][l] *= pivinv;
    for (ll=1;ll<=n;ll++)
      if (ll != icol) {
dum=a[ll][icol];
a[ll][icol]=0.0;
for (l=1;l<=n;l++) a[ll][l] -= a[icol][l]*dum;
for (l=1;l<=m;l++) b[ll][l] -= b[icol][l]*dum;
      }
  }
  for (l=n;l>=1;l--) {
    if (indxr[l] != indxc[l])
      for (k=1;k<=n;k++)
SWAP(a[k][indxr[l]],a[k][indxc[l]]);
  }
  free_ivector(ipiv,1,n);
  free_ivector(indxr,1,n);
  free_ivector(indxc,1,n);
```

```
}
#undef SWAP
#undef NRANSI


#define NRANSI
#define SAFETY 0.9
#define PGROW -0.2
#define PSHRNK -0.25
#define ERRCON 1.89e-4

void rkqs(double y[],
  double dydx[],
  double aux[],
  int n,
  double *x,
  double htry,
  double eps,
  double yscal[],
  double *hdid,
  double *hnext,
  void (*derivs)(double, double [], double [], double []))
{
  void rkck(double y[],
    double dydx[],
    double aux[],
    int n,
    double x,
    double h,
    double yout[],
    double yerr[],
    void (*derivs)(double, double [], double [],double[]));
  int i;
  double errmax,h,htemp,xnew,*yerr,*ytemp;

  yerr=vector(1,n);
  ytemp=vector(1,n);
  h=htry;
  for (;;) {
    rkck(y,dydx,aux,n,*x,h,ytemp,yerr,derivs);
    errmax=0.0;
    for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
    errmax /= eps;
    if (errmax <= 1.0) break;
    htemp=SAFETY*h*pow(errmax,PSHRNK);
    h=(h >= 0.0 ? FMAX(htemp,0.1*h) : FMIN(htemp,0.1*h));
    xnew=(*x)+h;
    if (xnew == *x) nrerror("stepsize underflow in rkqs");
  }
  if (errmax > ERRCON) *hnext=SAFETY*h*pow(errmax,PGROW);
  else *hnext=5.0*h;
  *x += (*hdid=h);
  for (i=1;i<=n;i++) y[i]=ytemp[i];
  free_vector(ytemp,1,n);
```

```
  free_vector(yerr,1,n);
}
#undef SAFETY
#undef PGROW
#undef PSHRNK
#undef ERRCON
#undef NRANSI

#define NRANSI

void rkck(double y[],
  double dydx[],
  double aux[],
  int n,
  double x,
  double h,
  double yout[],
  double yerr[],
  void (*derivs)(double, double [], double [], double []))
{
  int i;
  static double a2=0.2,a3=0.3,a4=0.6,a5=1.0,a6=0.875,b21=0.2,
  b31=3.0/40.0,b32=9.0/40.0,b41=0.3,b42 = -0.9,b43=1.2,
  b51 = -11.0/54.0, b52=2.5,b53 = -70.0/27.0,b54=35.0/27.0,
  b61=1631.0/55296.0,b62=175.0/512.0,b63=575.0/13824.0,
  b64=44275.0/110592.0,b65=253.0/4096.0,c1=37.0/378.0,
  c3=250.0/621.0,c4=125.0/594.0,c6=512.0/1771.0,
  dc5 = -277.00/14336.0;
  double dc1=c1-2825.0/27648.0,dc3=c3-18575.0/48384.0,
  dc4=c4-13525.0/55296.0,dc6=c6-0.25;
  double *ak2,*ak3,*ak4,*ak5,*ak6,*ytemp;

  ak2=vector(1,n);
  ak3=vector(1,n);
  ak4=vector(1,n);
  ak5=vector(1,n);
  ak6=vector(1,n);
  ytemp=vector(1,n);
  for (i=1;i<=n;i++)
    ytemp[i]=y[i]+b21*h*dydx[i];
  (*derivs)(x+a2*h,ytemp,ak2,aux);
  for (i=1;i<=n;i++)
    ytemp[i]=y[i]+h*(b31*dydx[i]+b32*ak2[i]);
  (*derivs)(x+a3*h,ytemp,ak3,aux);
  for (i=1;i<=n;i++)
    ytemp[i]=y[i]+h*(b41*dydx[i]+b42*ak2[i]+b43*ak3[i]);
  (*derivs)(x+a4*h,ytemp,ak4,aux);
  for (i=1;i<=n;i++)
    ytemp[i]=y[i]+h*(b51*dydx[i]+b52*ak2[i]+b53*ak3[i]+b54*ak4[i]);
  (*derivs)(x+a5*h,ytemp,ak5,aux);
  for (i=1;i<=n;i++)
    ytemp[i]=y[i]+h*(b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]+b65*ak5[i]);
  (*derivs)(x+a6*h,ytemp,ak6,aux);
  for (i=1;i<=n;i++)
```

```
    yout[i]=y[i]+h*(c1*dydx[i]+c3*ak3[i]+c4*ak4[i]+c6*ak6[i]);
  for (i=1;i<=n;i++)
    yerr[i]=h*(dc1*dydx[i]+dc3*ak3[i]+dc4*ak4[i]+dc5*ak5[i]+dc6*ak6[i]);
  free_vector(ytemp,1,n);
  free_vector(ak6,1,n);
  free_vector(ak5,1,n);
  free_vector(ak4,1,n);
  free_vector(ak3,1,n);
  free_vector(ak2,1,n);
}
#undef NRANSI
```

```
/*
   integ.h

   Note all revision after 26 Dec 1996 below:


*/

void odeint(double ystart[],
    double aux[],
    int nvar,
    double x1,
    double x2,
    double eps,
    double h1,
    double hmin,
    int *nok,
    int *nbad,
    void (*derivs)(double, double [], double [], double []),
    void (*rkqs)(double [],
 double [],
 double [],
 int,
 double *,
 double,
 double,
 double [],
 double *,
 double *,
 void (*)(double, double [], double [], double [])));


void rkqs(double y[],
  double dydx[],
  double aux[],
  int n,
  double *x,
  double htry,
  double eps,
  double yscal[],
  double *hdid,
  double *hnext,
  void (*derivs)(double, double [], double [], double []));


void gaussj(double **a, int n, double **b, int m);
```

The following code may contain portions derived from Numerical Recipes

```
/*
   nrutil.c

   THIS FILE CONTAINS MODIFIED CODE FROM NUMERICAL RECIPES

   Numerical Recipes copyright restrictions may apply.


   Note any revision after 26 Dec 1996 below:

   1. all instances of "float" explicitly replaced with
   "double" (SRS,12/27/96)
*/


/*
   CAUTION: This is the ANSI C (only) version of the Numerical Recipes
   utility file nrutil.c.  Do not confuse this file with the same-named
   file nrutil.c that is supplied in the 'misc' subdirectory.
   *That* file is the one from the book, and contains both ANSI and
   traditional K&R versions, along with #ifdef macros to select the
   correct version.  *This* file contains only ANSI C.
*/

#include <stdio.h>
/* #include <stddef.h>
 */
#include <stdlib.h>
#define NR_END 1
#define FREE_ARG char*

void nrerror(char error_text[])
/* Numerical Recipes standard error handler */
{
fprintf(stderr,"Numerical Recipes run-time error...\n");
fprintf(stderr,"%s\n",error_text);
fprintf(stderr,"...now exiting to system...\n");
exit(1);
}

double *vector(long nl, long nh)
/* allocate a double vector with subscript range v[nl..nh] */
{
double *v;

v=(double *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(double)));
if (!v) nrerror("allocation failure in vector()");
```

```
return v-nl+NR_END;
}

int *ivector(long nl, long nh)
/* allocate an int vector with subscript range v[nl..nh] */
{
int *v;

v=(int *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(int)));
if (!v) nrerror("allocation failure in ivector()");
return v-nl+NR_END;
}

unsigned char *cvector(long nl, long nh)
/* allocate an unsigned char vector with subscript range v[nl..nh] */
{
unsigned char *v;

v=(unsigned char *)malloc((size_t)
  ((nh-nl+1+NR_END)*sizeof(unsigned char)));
if (!v) nrerror("allocation failure in cvector()");
return v-nl+NR_END;
}

unsigned long *lvector(long nl, long nh)
/* allocate an unsigned long vector with subscript range v[nl..nh] */
{
unsigned long *v;

v=(unsigned long *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(long)));
if (!v) nrerror("allocation failure in lvector()");
return v-nl+NR_END;
}

double *dvector(long nl, long nh)
/* allocate a double vector with subscript range v[nl..nh] */
{
double *v;

v=(double *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(double)));
if (!v) nrerror("allocation failure in dvector()");
return v-nl+NR_END;
}

double **matrix(long nrl, long nrh, long ncl, long nch)
/* allocate a duble matrix with subscript range m[nrl..nrh][ncl..nch] */
{
long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
double **m;

/* allocate pointers to rows */
m=(double **) malloc((size_t)((nrow+NR_END)*sizeof(double*)));
if (!m) nrerror("allocation failure 1 in matrix()");
m += NR_END;
```

```
m -= nrl;

/* allocate rows and set pointers to them */
m[nrl]=(double *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(double)));
if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
m[nrl] += NR_END;
m[nrl] -= ncl;

for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

/* return pointer to array of pointers to rows */
return m;
}

double **dmatrix(long nrl, long nrh, long ncl, long nch)
/* allocate a double matrix with subscript range m[nrl..nrh][ncl..nch] */
{
long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
double **m;

/* allocate pointers to rows */
m=(double **) malloc((size_t)((nrow+NR_END)*sizeof(double*)));
if (!m) nrerror("allocation failure 1 in matrix()");
m += NR_END;
m -= nrl;

/* allocate rows and set pointers to them */
m[nrl]=(double *) malloc((size_t)((nrow*ncol+NR_END)
  *sizeof(double)));
if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
m[nrl] += NR_END;
m[nrl] -= ncl;

for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

/* return pointer to array of pointers to rows */
return m;
}

int **imatrix(long nrl, long nrh, long ncl, long nch)
/* allocate a int matrix with subscript range m[nrl..nrh][ncl..nch] */
{
long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
int **m;

/* allocate pointers to rows */
m=(int **) malloc((size_t)((nrow+NR_END)*sizeof(int*)));
if (!m) nrerror("allocation failure 1 in matrix()");
m += NR_END;
m -= nrl;


/* allocate rows and set pointers to them */
m[nrl]=(int *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(int)));
```

```
if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
m[nrl] += NR_END;
m[nrl] -= ncl;

for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

/* return pointer to array of pointers to rows */
return m;
}

double **submatrix(double **a, long oldrl, long oldrh, long oldcl, long oldch,
long newrl, long newcl)
/* point a submatrix [newrl..][newcl..] to a[oldrl..oldrh][oldcl..oldch] */
{
long i,j,nrow=oldrh-oldrl+1,ncol=oldcl-newcl;
double **m;

/* allocate array of pointers to rows */
m=(double **) malloc((size_t) ((nrow+NR_END)*sizeof(double*)));
if (!m) nrerror("allocation failure in submatrix()");
m += NR_END;
m -= newrl;

/* set pointers to rows */
for(i=oldrl,j=newrl;i<=oldrh;i++,j++) m[j]=a[i]+ncol;

/* return pointer to array of pointers to rows */
return m;
}

double **convert_matrix(double *a, long nrl, long nrh, long ncl, long nch)
/* allocate a double matrix m[nrl..nrh][ncl..nch] that points to the matrix
declared in the standard C manner as a[nrow][ncol], where nrow=nrh-nrl+1
and ncol=nch-ncl+1. The routine should be called with the address
&a[0][0] as the first argument. */
{
long i,j,nrow=nrh-nrl+1,ncol=nch-ncl+1;
double **m;

/* allocate pointers to rows */
m=(double **) malloc((size_t) ((nrow+NR_END)*sizeof(double*)));
if (!m) nrerror("allocation failure in convert_matrix()");
m += NR_END;
m -= nrl;

/* set pointers to rows */
m[nrl]=a-ncl;
for(i=1,j=nrl+1;i<nrow;i++,j++) m[j]=m[j-1]+ncol;
/* return pointer to array of pointers to rows */
return m;
}

double ***f3tensor(long nrl, long nrh, long ncl, long nch, long ndl, long ndh)
/* allocate a double 3tensor with range t[nrl..nrh][ncl..nch][ndl..ndh] */
```

```
{
long i,j,nrow=nrh-nrl+1,ncol=nch-ncl+1,ndep=ndh-ndl+1;
double ***t;

/* allocate pointers to pointers to rows */
t=(double ***) malloc((size_t)((nrow+NR_END)*sizeof(double**)));
if (!t) nrerror("allocation failure 1 in f3tensor()");
t += NR_END;
t -= nrl;

/* allocate pointers to rows and set pointers to them */
t[nrl]=(double **) malloc((size_t)((nrow*ncol+NR_END)*
   sizeof(double*)));
if (!t[nrl]) nrerror("allocation failure 2 in f3tensor()");
t[nrl] += NR_END;
t[nrl] -= ncl;

/* allocate rows and set pointers to them */
t[nrl][ncl]=(double *) malloc((size_t)((nrow*ncol*ndep+NR_END)
       *sizeof(double)));
if (!t[nrl][ncl]) nrerror("allocation failure 3 in f3tensor()");
t[nrl][ncl] += NR_END;
t[nrl][ncl] -= ndl;

for(j=ncl+1;j<=nch;j++) t[nrl][j]=t[nrl][j-1]+ndep;
for(i=nrl+1;i<=nrh;i++) {
t[i]=t[i-1]+ncol;
t[i][ncl]=t[i-1][ncl]+ncol*ndep;
for(j=ncl+1;j<=nch;j++) t[i][j]=t[i][j-1]+ndep;
}

/* return pointer to array of pointers to rows */
return t;
}

void free_vector(double *v, long nl, long nh)
/* free a double vector allocated with vector() */
{
free((FREE_ARG) (v+nl-NR_END));
}

void free_ivector(int *v, long nl, long nh)
/* free an int vector allocated with ivector() */
{
free((FREE_ARG) (v+nl-NR_END));
}

void free_cvector(unsigned char *v, long nl, long nh)
/* free an unsigned char vector allocated with cvector() */
{
free((FREE_ARG) (v+nl-NR_END));
}

void free_lvector(unsigned long *v, long nl, long nh)
```

```
/* free an unsigned long vector allocated with lvector() */
{
free((FREE_ARG) (v+nl-NR_END));
}

void free_dvector(double *v, long nl, long nh)
/* free a double vector allocated with dvector() */
{
free((FREE_ARG) (v+nl-NR_END));
}

void free_matrix(double **m, long nrl, long nrh, long ncl, long nch)
/* free a double matrix allocated by matrix() */
{
free((FREE_ARG) (m[nrl]+ncl-NR_END));
free((FREE_ARG) (m+nrl-NR_END));
}

void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch)
/* free a double matrix allocated by dmatrix() */
{
free((FREE_ARG) (m[nrl]+ncl-NR_END));
free((FREE_ARG) (m+nrl-NR_END));
}

void free_imatrix(int **m, long nrl, long nrh, long ncl, long nch)
/* free an int matrix allocated by imatrix() */
{
free((FREE_ARG) (m[nrl]+ncl-NR_END));
free((FREE_ARG) (m+nrl-NR_END));
}

void free_submatrix(double **b, long nrl, long nrh, long ncl, long nch)
/* free a submatrix allocated by submatrix() */
{
free((FREE_ARG) (b+nrl-NR_END));
}

void free_convert_matrix(double **b, long nrl, long nrh, long ncl, long nch)
/* free a matrix allocated by convert_matrix() */
{
free((FREE_ARG) (b+nrl-NR_END));
}

void free_f3tensor(double ***t, long nrl, long nrh, long ncl, long nch,
long ndl, long ndh)
/* free a double f3tensor allocated by f3tensor() */
{
free((FREE_ARG) (t[nrl][ncl]+ndl-NR_END));
free((FREE_ARG) (t[nrl]+ncl-NR_END));
free((FREE_ARG) (t+nrl-NR_END));
}
```

The following code may contain portions derived from Numerical Recipes

```
/*
   nrutil.h

   THIS FILE CONTAINS MODIFIED CODE FROM NUMERICAL RECIPES
   Numerical Recipes Copyright may apply.


   Note any modifications after 26 Dec 1996 below:

   1. Explictly replaced all floats with doubles (SRS,12/27/96).

*/

double *vector(long, long);
double **matrix(long,long,long,long);
double *dvector(long,long);
double **dmatrix(long,long,long,long);
int *ivector(long,long);
int **imatrix(long,long,long,long);
void free_vector(double *,long,long);
void free_dvector(double *,long,long);
void free_ivector(int *,long,long);
void free_matrix(double **,long,long,long,long);
void free_dmatrix(double **,long,long,long,long);
void free_imatrix(int **,long,long,long,long);
void nrerror(char *);
```

Appendix C

Simulators

C.1 Simulator, Chapter 2

This is the source code for the simulation results presented in Chapter 2. This code is dependent on the general purpose routines listed in Appendix F.

C++ Source

```
//
//
//  simulate.cc
//
//  Simulate the motor using the currents as state variables.
//
//  This is the ST2P module, essentially.
//
//  The vector of parameters is defined as follows.
//
//  p(1) = Xm;
//  p(2) = Xrr = Xm+Xl;
//  p(3) = Rr;
//  p(4) = Rs;
//  p(5) = Mystery Parameter related to J, Xm, #poles, etc.
//
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "linalg/linalg.h"
#include "tools.h"
#include "sysidtools.h"
#include "integrate.h"
```

```
#include "simulate.h"


void simulate(Vector &p, Vector &rhs, double h, int N,
void (*store)( int, Vector &, Vector &) );
void simulate(Matrix &M, Vector &p, double Volts, double h);
void LabFrame(Matrix &M, Matrix &L, double h, int SN);
void getinv( Vector &p, Matrix &A, Matrix &S, Vector &rhs);

// Private.

static void derivs(double , Vector &, Vector &);
static void derivs(double , Vector &);
static void derivs(double , Vector *, Vector *);
static void store(Matrix &);
static void store( int k, Vector &, Vector &);
static void store(int k, Vector &, Vector &, Matrix *);


#if STANDALONE
main()
{
  char comment[80];
  Vector rhs(2);
  Vector p(5);
  int N;
  double t;

  cout << "Simul Version 1.0 \n";

  cout << gets(comment) << "\n";

  cin >> N;                   // In from file.
  cin >> t;
  cin >> rhs(1);
  cin >> rhs(2);

  cin >> p(1);
  cin >> p(2);
  cin >> p(3);
  cin >> p(4);
  cin >> p(5);

  cout << N << "\n";                // Out to the next dude.
  cout << t << "\n";
  cout << rhs(1) << "\n";
  cout << rhs(2) << "\n";


  // Scale the voltages. Simul expects line to line voltages.
  rhs *= sqrt(2)/sqrt(3);

  // Transform inertia.
  p(5) = (1/p(5)) * (2.0)*(2.0) * (3.0/2.0) * (p(1)/(2*60*M_PI*2*60*M_PI));
```

```
  Matrix M(N,10);
  M = 0;

  simulate(M,p,rhs,t);

  int i;

/*
  for(i = 1; i <= N; i++)
    printf("%.3e  %.3e  %.3e\n", M(i,1), M(i,2), M(i,9));
*/

  for(i = 1; i <= N; i++)
    printf("%.3e %.3e %.3e %.3e\n",t*i, M(i,1), M(i,2), M(i,9));

}
#endif



//
// The simulator that we used before.
//
void simulate(Matrix &M, Vector &p, double Volts, double H)
{
  Vector rhs(2);
  rhs(1) = Volts;
  rhs(2) = 0.0;
  simulate(M,p,rhs,H);
}


void simulate(Matrix &M, Vector &p, Vector &rhs, double H)
{
  store(M);
  simulate(p,rhs,H,M.q_nrows(),store);
}

//
// Simulator, general purpose.
//
void simulate(Vector &p, Vector &rhs, double H, int N,
void (*store)( int, Vector &, Vector &) )
{
  int j,k;
  int nok,nbd;

  Vector i(5),di(5);

  // Begin Initialization
  i = 0.0;
  i(5) = 1.0;
```

```
  derivs(-1.0, p, rhs);
  // End Initialization

  // Start integrating.
  for(k = 1; k <= N; k++) {
    derivs(0.0,i,di);           // get di/dt before calling rk4
    (*store)(k,i,di);
    odeint(i, k*H, (k+1)*H, 1e-5, H, 0.0, nok, nbd, derivs, rkqs);
  }
}


//  derivs
//
// This routine computes the derivatives to be integrated
// to simulate the motor. The currents are taken as state
// variables. To use, the routine must be initialized by
// calling as:
//
// derivs(V, p);
//
// A pointer to the routine may then be passed to the
// numerical integration code of choice.
//
void derivs(double t, Vector &i, Vector &di)
{
  derivs(t, &i, &di);
}

void derivs(Vector &rhs, Vector &p)
{
  derivs(-1.0,&p,&rhs);
}

void derivs(double t, Vector *i, Vector *di)
{
  static Matrix A(5,5),B(5,5),S(5,5),TMP(5,5);
  static Vector rhs(5);
  static double omega = M_PI*60*2;

  // Initialize.
  if( t < 0.0 ) {
    A = 0.0;
    S = 0.0;
    B = 0.0;
    getmat((*i),A,B,S);              // get these matrices.
    rhs = 0.0;
    rhs(1) = (*di)(1);              // This is really the voltage.
    rhs(2) = (*di)(2);
    B *= (1/omega);
    B(5,5) = 1/(*i)(5);
  } else {
    TMP = S;
    TMP *= (*i)(5);
```

```
    TMP += A;

    (*di) = rhs;
    (*di) -= TMP*(*i);
    (*di)(5) =  -(*i)(1)*(*i)(4) + (*i)(2)*(*i)(3);

    TMP = B;
    lusolve(TMP,(*di),1);
  }
}


//
// begin storage code
//
void store(Matrix &M)
{
  Vector i(4);
  store(0,i,i,(Matrix *)&M);
}

void store(int k, Vector &i, Vector &di)
{
  store(k,i,di,(Matrix *)NULL);
}

// This code store the state and derivatives in the form that we
// are used to...
void store(int k, Vector &i, Vector &di, Matrix *M)
{
  static Matrix *z;
  int j;

  if(M == NULL) {
    for(j = 1; j <= 4; j++) {
      (*z)(k,j)   = i(j);
      (*z)(k,j+4) = di(j);
    }
    (*z)(k,9)   = i(5);
    (*z)(k,10)  = di(5);
  } else z = M;
}
//
// end storage code
//



//
// Utility code.
//
#define Xm  (p(1))
#define Xrr (p(2))
#define Xl  ((Xrr)-(Xm))
```

```
#define Rr  (p(3))
#define Rs  (p(4))
#define J   (p(5))
void getmat(Vector &p, Matrix&A, Matrix&B, Matrix&S)
{
  // Coefficients of the state variables...
  A(1,1) = Rs;     A(1,2) = Xm+Xl;A(1,3) = 0;  A(1,4) = Xm;
  A(2,1) = -Xm-Xl; A(2,2) = Rs;   A(2,3) = -Xm;A(2,4) = 0;
  A(3,1) = 0;      A(3,2) = 0;    A(3,3) = Rr; A(3,4) = 0;
  A(4,1) = 0;      A(4,2) = 0;    A(4,3) = 0;  A(4,4) = Rr;

  // Coefficients of the derivatives.
  B(1,1) = (Xm+Xl);B(1,2) = 0;      B(1,3) = Xm;     B(1,4) = 0;
  B(2,1) = 0;      B(2,2) = (Xm+Xl);B(2,3) = 0;      B(2,4) = Xm;
  B(3,1) = Xm;     B(3,2) = 0;      B(3,3) = (Xm+Xl);B(3,4) = 0;
  B(4,1) = 0;      B(4,2) = Xm;     B(4,3) = 0;      B(4,4) = (Xm+Xl);


  // Coefficients of the slip/state variable product...
  S(1,1) = 0;   S(1,2) = 0;   S(1,3) = 0;      S(1,4) = 0;
  S(2,1) = 0;   S(2,2) = 0;   S(2,3) = 0;      S(2,4) = 0;
  S(3,1) = 0;   S(3,2) = Xm;  S(3,3) = 0;      S(3,4) = Xm+Xl;
  S(4,1) = -Xm; S(4,2) = 0;   S(4,3) = -Xm-Xl; S(4,4) = 0;
}
#undef Xm
#undef Xrr
#undef Xl
#undef Rr
#undef Rs
#undef J



//
//
// Convert Motor Variables to the LabFrame.
//
//
void LabFrame( Matrix &M, Matrix &Lab, double h)
{
  int i,j;
  double Theta = 0, Beta = 0, ThetaR = 0;
  double omega = 2*60*M_PI;
  double p = 2 * M_PI / 3;
  double wr;

  for(i = 1; i <= M.q_nrows(); i++) {
    Lab(i,1) = cos(Theta+0) * M(i,1) + sin(Theta+0) * M(i,2);
    Lab(i,2) = cos(Theta-p) * M(i,1) + sin(Theta-p) * M(i,2);
    Lab(i,3) = cos(Theta+p) * M(i,1) + sin(Theta+p) * M(i,2);

    Beta = Theta - ThetaR;

    Lab(i,4) = cos(Beta+0) * M(i,3) + sin(Beta+0) * M(i,4);
```

```
    Lab(i,5) = cos(Beta-p) * M(i,3) + sin(Beta-p) * M(i,4);
    Lab(i,6) = cos(Beta+p) * M(i,3) + sin(Beta+p) * M(i,4);

    // Integrate theta.
    //
    wr = omega - M(i,9)*omega;
    Theta   += h*omega;
    ThetaR  += h*wr;
  }
}
```

Include file

```
void simulate(Matrix &M, Vector &p, Vector &rhs, double H);
void simulate(Matrix &M, Vector &p, double Volts = 127.17, double h = .000005);
void getmat(Vector &p, Matrix & A, Matrix & B, Matrix& S);
```

make file

```
simul : simul.o matrix1.o matrix2.o vector.o myenv.o tools.o
sysidtools.o integrate.o
g++ -o simul sysidtools.o simul.o integrate.o matrix1.o
matrix2.o vector.o myenv.o tools.o -lm -lg++

simul.o: simulate.cc linalg/linalg.h
g++ -o simul.o -c -ggdb -DSTANDALONE simulate.cc

matrix1.o: linalg/matrix1.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/matrix1.cc

matrix2.o: linalg/matrix2.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/matrix2.cc

vector.o:  linalg/vector.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/vector.cc

myenv.o:   linalg/myenv.cc  linalg/myenv.h
g++ -c linalg/myenv.cc

tools.o:   tools.cc linalg/myenv.h linalg/linalg.h tools.h
g++ -c tools.cc
```

```
sysidtools.o: sysidtools.cc sysidtools.h
g++ -c sysidtools.cc

integrate.o: integrate.cc integrate.h linalg/linalg.h
g++ -c integrate.cc
```

C.2 Simulator, Chapter 5

This is the source code used for the simulation results presented in Chapter 5. This simulator takes inputs v_{qs}, v_{ds} and s from a file. Discrete inputs are zero-order held over the step size. The file `induct.c` below is dependent on the file `csim.c` (and its support files) described in Appendix B.

C Source

```
/*
   induct.c

   Copyright (c) 1996
   Laboratory for Electronic and Electromagnetic Systems
   Massachusetts Institute of Technology

   Description:

   Induction motor simulator using Krause´s model.

   Date:      9/17/96

   Bugs:
   * currently does not support arbitrary voltage input waveforms
   * mechanical subsystem is assumed to be purely inertial
     (easily changed)


   Note any revision after 26 Dec 1996 below:

   * Units specified in output.

 **
 ** MODIFIED TO SIMULATE INDUCTION MOTOR USING EXPLICITLY DEFINED SLIP
 ** AND ARBITRARY VOLTAGE INPUTS.
 **
 ** This was done so that the parameters and slip estimate of IDENTIFY
 ** could be validated using experimental data.
 **
 ** Input file format:
 **
```

```
 ** [vq] [vd] [slip]
 **

 */

#include <stdio.h>
#include <math.h>
#include "integ.h"
#include "csim.h"

#define CLEANUP (-1234)

void induction(double, double[], double[], double[]);
void mystore(double, double[], double[], double[]);
void ongrid(double , double [], double [], double []);

char *_datafile = "induct.vdq";
char *_inputfile = "induct.in";
char *_outputfile = "induct.out";

int main(void)
{
    FILE *f;
    double i[10];
    double di[10];
    double p[10];
    double vs[10];
    double aux[10];
    double T;
    long N;

    /* Read in parameters. */
    f = (FILE *) fopen(_inputfile, "r");
    if (f == NULL) {
printf("Problem opening %s...\n", _inputfile);
return 1;
    }
    printf("Reading parameters from %s...\n", _inputfile);

                                    /* Value     Name     Units        */
    fscanf(f, "%lf\n", &(p[1]));    /* 26.13     XM       Ohms @ 60 Hz */
    fscanf(f, "%lf\n", &(p[2]));    /* 26.88     XSS      Ohms @ 60 Hz */
    fscanf(f, "%lf\n", &(p[3]));    /* .816      RR       Ohms         */
    fscanf(f, "%lf\n", &(p[4]));    /* .435      RS       Ohms         */
    fscanf(f, "%lf\n", &(p[5]));    /*  unused  */
    fscanf(f, "%lf\n", &(p[6]));    /*  unused  */
    fscanf(f, "%lf\n", &(vs[1]));   /*  unused  */
    fscanf(f, "%lf\n", &(vs[2]));   /*  unused  */
    fscanf(f, "%lf\n", &T);     /* Sample rate                    */
    fscanf(f, "%ld\n", &N);     /* Number of samples              */
    fclose(f);

    /* Initialize output function
       and derivatives function
```

```
     */
    mystore(INITIALIZE,p,vs,aux);
    ongrid(INITIALIZE,  p,vs,aux);
    induction(INITIALIZE, p, vs, aux);

    /* open output file and simulate
       the motor
     */
    f = (FILE *) fopen(_outputfile, "w");
    if (f == NULL) {
printf("Problem opening %s...\n", _outputfile);
return 1;
    }
    filestore_init(f);

    simulate(i, di, aux, 4, T, N, mystore, induction, ongrid);
    fclose(f);

    /* close up input file              */
    ongrid(CLEANUP,p,vs,aux);

    /* Tell user where the output is... */
    printf("Output is in %s:\n", _outputfile);
    printf(" 1:i_q(amps) \t 2:i_d(amps) \n");

    return 0;
}


/* states        */
#define Pdr     (s[4])
#define Pqr     (s[3])
#define Pds     (s[2])
#define Pqs     (s[1])

/* derivatives    */
#define dPdr    (ds[4])
#define dPqr    (ds[3])
#define dPds    (ds[2])
#define dPqs    (ds[1])

#define vqs     (aux[1])
#define vds     (aux[2])
#define slip    (aux[3])

/* Parameters     */
#define xrr     (xss)
#define D       (xss*xss - xm*xm)
#define omega   (2.0*M_PI*60.0)
#define phi     (2.0*M_PI/3.0)

void induction(double t, double s[], double ds[], double aux[])
{
    static double xm, xss, rr, rs; /* Parameters.            */
```

```
    static double vqr, vdr;         /* rotor excitation        */

    /* Save parameters locally. */
    if (t == INITIALIZE) {
xm = s[1];
xss = s[2];
rr = s[3];
rs = s[4];
vqr = 0.0;
vdr = 0.0;
return;
    }
    /* Set Initial condition (CSIM.C calls with this option) */
    if (t == ZEROSTATE) {
Pqs = Pds = Pqr = Pdr = 0.0;
    }
    /* ACTUAL SIMULATION CODE FOLLOWS */

    /*
       This is how an induction machine works.
     */
    dPqs = omega * (vqs - ((rs * xrr / D) * Pqs +
Pds - (rs * xm / D) * Pqr));
    dPds = omega * (vds - ((rs * xrr / D) * Pds -
Pqs - (rs * xm / D) * Pdr));
    dPqr = omega * (vqr - (-(rr * xm / D) * Pqs +
(rr * xss / D) * Pqr + slip * Pdr));
    dPdr = omega * (vdr - (-(rr * xm / D) * Pds -
slip * Pqr + (rr * xss / D) * Pdr));
}


void ongrid(double t, double s[], double ds[], double aux[])
{
  static FILE *f;
  double t1,t2,t3;

  if(t == INITIALIZE) {
    f = (FILE *)fopen(_datafile, "r");
    if(f == NULL) {
      printf("error opening input file \n");
      exit(1);
    }
    vqs = 0;
    vds = 0;
    slip = 0;
    return;
  }

  if(t == CLEANUP) {
    fclose(f);
    return;
  }
```

```
  fscanf(f,"%lf %lf %lf\n", &t1, &t2, &t3);

  if( feof(f) )
    t1 = t2 = t3 = 0;

  vqs  = t1;
  vds  = t2;
  slip = t3;
}


/*
   output formatter

 */
void mystore(double t, double s[], double ds[], double aux[])
{
    static double xss, xm;
    static double col[12];
    double iqs, ids;

    /* Initialize. Need XM and XSS to get currents
       from fluxes.
     */
    if (t == INITIALIZE) {
xm = s[1];
xss = s[2];
    } else {
/* Get time. */
/* PUT OUTPUT IN DQ FRAME */
/* get iqs, ids */
col[1] = (xss * Pqs - xm * Pqr) / D;
col[2] = (xss * Pds - xm * Pdr) / D;

filestore(col, 2);
    }
}
```

make file

```
# induct.make
#
# SRS, Jan 1997.
#
# To make this makefile work on your system:
#
# CC=[name of your compiler]
```

```
# CompileOnly=[options to compile but not link]
# CompileLink=[options to compile and link]
#
# For most Unix systems running GCC:
# CC=gcc
# CompileOnly=-c -o
# CompileLink=-lm -o
#
# For DOS running MS C6
# CC=cl
# CompileOnly=/AH /FPi87 /c /W4 /Fo
# CompileLink=/Fe
#

CC=gcc
CompileOnly=-c -o
CompileLink=-lm -o

induct : induct.o integ.o csim.o nrutil.o
$(CC) $(CompileLink)induct induct.o integ.o csim.o nrutil.o

induct.o: induct.c
$(CC) $(CompileOnly)induct.o induct.c

nrutil.o: nrutil.c nrutil.h
$(CC) $(CompileOnly)nrutil.o nrutil.c

csim.o: csim.c csim.h
$(CC) $(CompileOnly)csim.o csim.c

integ.o: integ.c integ.h
$(CC) $(CompileOnly)integ.o integ.c
```

Appendix D

Extrapolative Identification Code

This appendix contains the code implementing the extrapolative identification procedures described in Chapter 3. This code is dependent on the general purpose routines listed in Appendix E.

C++ Source

```
//
// ELH
//
// This filter operates on data [iqs] [ids] [lambda_iqs] [lambda_ids]
// and determines the parameters of the induction machine using an
// extrpolative technique.
//

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

#include "linalg/myenv.h"
#include "linalg/linalg.h"

#include "tools.h"
#include "sysidtools.h"
#include "lambda.h"

#define VERSION_NUMBER    (double)0.1
#define VERSION_CODE      'a'

#define Rows( q ) (  (q).q_nrows())
#define Cols( q )  ( (q).q_ncols())
#define Length(v)  ((v).q_no_elems())
```

```
//
// Local prototypes.
//
void go( void );
void Elh(Matrix &v1, Matrix &v2, double T, double tau);
void subsolve(Matrix &M, int i1, int i2, double &y1, double &y2, double &x,
double tau, double T);
void subsolve2(Matrix &M, int i1, int i2, double &y1, double &y2, double &x,
double tau, double T);
void ReadMatrix( Matrix &M );
void dumpf(Vector &x, char *fileme);

//
// main program
//
void main(int argc, char *argv[])
{
  double T,tau;
  int n1,n2,n3,n4;
  int i,j,k,N;
  char str[1024];

  // Source of data.
  cout << "Elh Version " << VERSION_NUMBER << VERSION_CODE
       << " | " << gets(str) << "\n";
  // Comment.
  cout << gets( str ) << "\n";

  cin >> n1;                    // Vsection1
  cin >> n2;
  cin >> n3;                    // Vsection2
  cin >> n4;
  cin >> tau;                   // The tau to use.

  cin >> N;                     // Number of data points total.
  cin >> T;                     // Time interval (sampling rate)

  cout << n1  << "\n";          // Echo to report.
  cout << n2  << "\n";
  cout << n3  << "\n";
  cout << n4  << "\n";
  cout << tau << "\n";
  cout << N   << "\n";
  cout << T   << "\n";


  Matrix P(N,8);
  Vector x(N);

  /* Read data file

     data file format:
     [iqs] [ids] [vqs] [vds]
```

```
   */
  Matrix M(N,4);
  ReadMatrix(M);


  /*
     apply lambda operators...
  */
  getcol(M,1,x);                  // X has iqs
//  lambda(x,T*10,T,1e-6,0.0);
  setcol(x,P,1);
  lambda(x,tau,T,1e-6,0.0);
  setcol(x,P,3);

  getcol(M,2,x);                  // X has ids
//  lambda(x,T*10,T,1e-6,0.0);
  setcol(x,P,2);
  lambda(x,tau,T,1e-6,0.0);
  setcol(x,P,4);

  getcol(M,3,x);                  // X has vqs
//  lambda(x,T*10,T,1e-6,0.0);
  setcol(x,P,7);
  lambda(x,tau,T,1e-6,0.0);
  setcol(x,P,5);

  getcol(M,4,x);                  // X has vds
//  lambda(x,T*10,T,1e-6,0.0);
  setcol(x,P,8);
  lambda(x,tau,T,1e-6,0.0);
  setcol(x,P,6);

  /*
     split into V-sections...
  */
  Matrix V1(n2-n1+1,8),V2(n4-n3+1,8);

  for(i = n1, j = 1; i <= n2; i++, j++)
    for(k = 1; k <= Cols(P); k++)
      V1(j,k) = P(i,k);

  for(i = n3, j = 1; i <= n4; i++, j++)
    for(k = 1; k <= Cols(P); k++)
      V2(j,k) = P(i,k);

  /*
     Format of P and V1 and V2 matrices:
     [iqs] [ids] [lambda_iqs] [lambda_ids] [lambda_vqs] [lambda_vds] [vqs] [vds]
  */
  Elh(V1,V2,T,tau);
}
```

```
/*
   read matrix from a file
*/
void ReadMatrix( Matrix &M )
{
  int R = Rows( M ), C = Cols( M ), i, j;
  char str[80];
  double tmp;

  for(i = 1; i <= R; i++) {
    for(j = 1; j <= C; j++) {
      cin >> str;
      sscanf(str,"%lf", &tmp);
      M(i,j) = tmp;
    }
  }
}


/*

   Solve the extrapolative sysid problem.

   Elh below applies the two submodels
   subsolve and subsolve2
*/


// number of partitions in V-Section 1
#define NV1   16
// Window Width, V section 1
#define WW1   200
#define NV2  100
#define WW2  100

void Elh(Matrix &v1, Matrix &v2, double T, double tau)
{
  int ii,i;
  double _y1,_y2,_x;


  Vector y1(NV1),x(NV1),y2(NV1);

  for(i = 1; i <= NV1; i++) {
    ii = (i-1)*(v1.q_nrows())/NV1;

    subsolve(v1,ii,min(ii+WW1, v1.q_nrows()),_y1,_y2,_x,tau,T);

    y1(i) = _y1;
    y2(i) = _y2;
    x(i) = _x;
  }

  double Xl,Xle,RrRs,RrRse;
```

```
  ratint(x,y1,0.0,Xl,Xle);
  ratint(x,y2,0.0,RrRs,RrRse);

/*
  plot(x);
  plot(y1);
  plot(y2);
*/

  Vector ny1(NV2);
  Vector ny2(NV2);
  Vector nx(NV2);

//  plot(v2);

  printf("Error Margins show error expected in extrapolation. \n");
  printf("They are not a reflection of the data quality.      \n");

  printf("Xl    = %lf, +/- %lf \n", Xl/2.0, Xle/2.0);
  printf("Rr+Rs = %lf, +/- %lf \n", RrRs, RrRse);

  for(i = 1; i <= NV2; i++) {
    ii = (i-1)*(v2.q_nrows())/NV2;

    subsolve2(v2,ii,min(ii+WW2,v2.q_nrows()),_y1,_y2,_x,tau,T);

    ny1(i) = _y1;
    ny2(i) = _y2;
    nx(i) = _x;
  }

  dumpf( nx, "x" );
  dumpf(ny1, "y1");
  dumpf(ny2, "y2");

  double Xm,Xme,Rr,Rre;

  ratint(nx,ny1,0.0,Xm,Xme);
  ratint(nx,ny2,0.0,Rr,Rre);

  printf("Xss   = %lf, +/- %lf \n", Xm, Xme);
  printf("Rs    = %lf, +/- %lf \n", Rr, Rre);
}

void dumpf(Vector &x, char *fileme)
{
  FILE *f;

  f = (FILE *)fopen(fileme,"w");
  dump((char *)NULL,x,"%.3e",f);
  fclose(f);
}
```

```
// Macros to unpack the matrix M
//
#define _iqs(n)  (M((n),1))
#define _ids(n)  (M((n),2))
#define _liqs(n) (M((n),3))
#define _lids(n) (M((n),4))
#define _lvqs(n) (M((n),5))
#define _lvds(n) (M((n),6))
#define _vqs(n)  (M((n),7))
#define _vds(n)  (M((n),8))

// Substitutions to obtain the parameters.
//
#define diqs(n)   (_iqs(n) - _liqs(n))
#define dids(n)   (_ids(n) - _lids(n))
#define iqs(n)    (tau * _liqs(n))
#define ids(n)    (tau * _lids(n))
#define vqs(n)    (tau * _lvqs(n))
#define vds(n)    (tau * _lvds(n))

//
// M is:  [iqs] [ids] [{lambda}iqs] [{lambda}ids] [{lambda}vqs]
// [{lambda}vds] [vqs] [vds]
//

/*
   Here's model #1
*/

void subsolve(Matrix &M, int i1, int i2, double &y1, double &y2, double &x,
double tau, double T)
{
  int i;
  double omega = M_PI*2.0*60.0;

  x = iqs((i1+i2)/2);

  Matrix A(2*(i2-i1), 2);
  Vector rhs(2*(i2-i1));
  Vector xp(2);

  for(i = 1; i <= i2-i1; i++) {
    A(i,1)        = ids(i+i1) + diqs(i+i1)/omega;
    A(i,2)        = iqs(i+i1);
    rhs(i)        = vqs(i+i1);
    A(i+i2-i1, 1) = -iqs(i+i1) + dids(i+i1)/omega;
    A(i+i2-i1, 2) = ids(i+i1);
    rhs(i+i2-i1)  = vds(i+i1);
  }

  normaleqn(A,xp,rhs);
```

```
  y1 = xp(1);
  y2 = xp(2);
}
#undef iqs(n)
#undef ids(n)
#undef vqs(n)
#undef vds(n)


/*
   here´s reduced model #2
*/
#define iqs(n) (_iqs(n))
#define ids(n) (_ids(n))
#define vqs(n) (_vqs(n))
#define vds(n) (_vds(n))

void subsolve2(Matrix &M, int i1, int i2, double &y1, double &y2, double &x,
double tau, double T)
{
  int i;
  double omega = M_PI*2.0*60.0;

  Matrix A(2*(i2-i1), 2);
  Vector rhs(2*(i2-i1));
  Vector xp(2);

  for(i = 1; i <= i2-i1; i++) {
    A(i,1)         =  ids(i+i1) /* + diqs(i+i1)/omega */ ;
    A(i,2)         =  iqs(i+i1);
    rhs(i)         =  vqs(i+i1);
    A(i+i2-i1, 1)  = -iqs(i+i1) /* + dids(i+i1)/omega */ ;
    A(i+i2-i1, 2)  =  ids(i+i1);
    rhs(i+i2-i1)   =  vds(i+i1);
  }

  normaleqn(A,xp,rhs);

  y1 = xp(1);
  y2 = xp(2);

  x = diqs( (i1+i2)/2);
}

#undef vqs(n)
#undef vds(n)
#undef iqs(n)
#undef ids(n)
```

make file

```
elh : elh.o lambda.o integrate.o weighting.o sysidtools.o matrix1.o matrix2.o
vector.o myenv.o tools.o simulate.o
g++ -o elh elh.o lambda.o integrate.o matrix1.o matrix2.o vector.o
  myenv.o tools.o sysidtools.o simulate.o -lm -lg++

elh.o: elh.cc linalg/linalg.h
g++ -o elh.o -c -ggdb -DSTANDALONE elh.cc

lambda.o: lambda.cc lambda.h
g++ -c lambda.cc

matrix1.o: linalg/matrix1.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/matrix1.cc

sysidtools.o: sysidtools.cc sysidtools.h
g++ -c sysidtools.cc

matrix2.o: linalg/matrix2.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/matrix2.cc

vector.o:  linalg/vector.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/vector.cc

myenv.o:   linalg/myenv.cc  linalg/myenv.h
g++ -c linalg/myenv.cc

tools.o:   tools.cc linalg/myenv.h linalg/linalg.h tools.h
g++ -c tools.cc

simulate.o: simulate.cc linalg/linalg.h
g++ -c simulate.cc

integrate.o: integrate.cc integrate.h
g++ -c integrate.cc
```

Appendix E

Modified Least Squares Code

This appendix contains C++ code implementing the modified least-squares procedure described in Chapter 4. The code is dependent on the general purpose routines listed in Appendix F.

There are two modules in this Appendix. The first is `identify.cc`, which contains the main program and drives the Levenburg-Marquardt solver in `mrqmin.cc`. The second is `estimate.cc`, which contains the rotor current and slip estimation code, the partial derivative calculations, and the loss function calculation.

E.1 `identify.cc`

C++ source

```
//
// Identify.cc
//
// Non-linear least square induction motor identification routine.
//
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "linalg/myenv.h"
#include "linalg/linalg.h"
#include "tools.h"
#include "sysidtools.h"
#include "simulate.h"
#include "mrqmin.h"
```

```
#include "estimate.h"
#include "lambda.h"
#include "filter.h"
#include "fourier.h"

#include "map.h"

void Identify(Matrix &Z, double T);
void InitialGuess( Matrix &Z, Vector &a, int ia[]);
void func(double x, Vector &a, double &y, Vector &dyda, int first);
void func(Matrix &P, double t, double v, double tau, double vqs, double vds);
void func(double x, Vector &a, double &y, Vector &dyda, int first, Matrix *_P);

#if STANDALONE
/*
main()
{
  int N = 16384,i;
  Matrix M(N,10);
  Vector p(5), rhs(4);

  double T = .00005;
  double Vln = 220;

  // Start out by simulating a motor (surprise,surprise)
  M = 0;

  p(1) = 26.13;
  p(2) = 26.88;
  p(3) = .816;
  p(4) = .435;
  p(5) = (1/.089) * (2.0)*(2.0) * (3.0/2.0) * (p(1)/(2*60*M_PI*2*60*M_PI));


  FILE *f  = (FILE *)fopen("identify.prm", "r");
  if(f == NULL)
    printf("Error opening parameter file...\n");
  else {
    double t;

    fscanf(f, "%lf\n", &t), p(1) = t;
    fscanf(f, "%lf\n", &t), p(2) = t + p(1);
    fscanf(f, "%lf\n", &t), p(3) = t;
    fscanf(f, "%lf\n", &t), p(4) = t;
    fscanf(f, "%lf\n", &t), p(5) = t;
    fscanf(f, "%lf\n", &T);
    fscanf(f, "%lf\n", &Vln);

    fclose(f);
  }

  printf("Simulating motor now...\n");
  printf("Xm      = %lf\n",    p(1));
  printf("Xss     = %lf\n",    p(2));
```

```
  printf("Rr      = %lf\n",    p(3));
  printf("Rs      = %lf\n",    p(4));
  printf("Pm      = %lf\n",    p(5));
  printf("T       = %lf\n",    T);
  printf("V l-n   = %lf\n",    Vln);

  rhs = 0;
  rhs(1) = Vln*sqrt(2.0)/sqrt(3.0);

  simulate(M, p, rhs(1), T);                // Adaptive step-size simulator.

  Vector qrt(N);

  Matrix Z(N, 7);
  char c;

  do {
    for(i = 1; i <= N; i++) {
      Z(i,1) = M(i,1);
      Z(i,2) = M(i,2);
      Z(i,3) = rhs(1);
      Z(i,4) = rhs(2);
      Z(i,5) = M(i,9);

      qrt(i) = M(i,1);
    }

    Identify(Z, T);             // Identify what we just simulated.

    printf("Run again? (y/n)\n");
    c = getchar();
  } while(c == 'y' || c == 'Y');
}
*/

main()
{
  int N = 16384,i;
  Matrix Z(N, 7);
  double iqs,ids,vqs,vds;

  // Fill with zero --> automatically zero pads.
  Z = 0;

  // Open file.
  FILE *f  = (FILE *)fopen("identify.in", "r");

  if(f == NULL) {
    printf("Error opening input file\n");
    return 1;
  }

  for(i = 1; i <= N && !feof(f); i++) {
    fscanf(f,"%lf %lf %lf %lf\n",&iqs,&ids,&vqs,&vds);
```

```
    if(!feof(f)) {
      Z(i,1) = iqs;
      Z(i,2) = ids;
      Z(i,3) = vqs;
      Z(i,4) = vds;
    }
  }
  fclose(f);

  double T = 4e-5;            // Sampling rate.

  Identify(Z, T);             // Identify what we just simulated.
}
#endif


// Get initial guess and target parameters from a file.
//
void InitialGuess(Matrix &Z, Vector &a, int ia[])
{
  // DEFAULTS
  xm = 24.0;
  xl = .8;
  rr = .9;
  rs = .3;
  J = (1.0/.089) * (2.0)*(2.0) * (3.0/2.0) * (26.13/(2*60*M_PI*2*60*M_PI));
//  J = (1.0/1.0) * (2.0)*(2.0) * (3.0/2.0) * (13.08/(2*60*M_PI*2*60*M_PI));

  ia[1] = 1;
  ia[2] = 1;
  ia[3] = 1;
  ia[4] = 1;
  ia[5] = 0;
  ia[6] = 0;

  FILE *f  = (FILE *)fopen("identify.igf", "r");
  if(f == NULL)
    printf("Error opening initial guess file...\n");
  else {
    double t;

    fscanf(f, "%lf, %d \n", &t, &(ia[1])), xm = t;
    fscanf(f, "%lf, %d \n", &t, &(ia[2])), xl = t;
    fscanf(f, "%lf, %d \n", &t, &(ia[3])), rr = t;
    fscanf(f, "%lf, %d \n", &t, &(ia[4])), rs = t;
    fscanf(f, "%lf, %d \n", &t, &(ia[5])), J = t;

    fclose(f);
  }

  printf("Initial Guess Report:\n");
  printf("Xm = %.3e,%d\n", xm, ia[1]);
  printf("Xl = %.3e,%d\n", xl, ia[2]);
```

```
  printf("Rr = %.3e,%d\n", rr, ia[3]);
  printf("Rs = %.3e,%d\n", rs, ia[4]);
  printf("Pm = %.3e,%d\n", J, ia[5]);
}



//
// Identify the induction motor.
//
#define NMAX 50
#define NMIN 5

void Identify(Matrix &Z, double T)
{
  int N = rows(Z);
  Matrix P(N*2,10);           // Workspace...
  Vector a(5);
  int i,j;
  int ia[10];
  double omega = 2.0*M_PI*60.0;
  double V = Z(1,5);
  double tau = .002;

  for(i = 1, j = 1; i <= N; i++,j += 2) {
    P(j  ,is) = Z(i,1);
    P(j+1,is) = Z(i,2);
    P(j  ,vs) = Z(i,3);
    P(j+1,vs) = Z(i,4);
    P(j  ,sj) = 0.0;
    P(j+1,sj) = -Z(i,5);
  }

  printf("iqs(1) = %lf\n", Z(1,1));

  printf("\n\nIdentifying...\n");
  printf("N   = %d\n", N);
  printf("tau = %lf\n", tau);
  printf("V = %lf \n", V);

/*
  Vector p(rows(Z));
  for(i = 1; i <= length(p); i++)
   p(i) = Z(i,1);

  printf("Plot of IQS \n");
  plot(p);
*/

  printf("T   = %lf\n", T);
  printf("tau = %lf\n", tau);
  printf("1/tau = %lf\n", 1/tau);

  // Initialize the objective function.
```

```
  func(P,T,V,tau,V,0);

  // make a parameter guess.
  InitialGuess(Z,a,ia);

  //
  // Initialize stuff to be fitted.
  int n,q;
  int N2 = 64;

  Vector x(N2), y(N2), sig(N2);

  double f0,f1;

/*
  for(j = 1,i = 1; i <= N2; i++,j++) {
    x(i) = (double)(j);
    sig(i) = 1.0;
  }
*/

  for(i = 1; i <= N2; i++) {
    x(i) = 1 + (2*N - N2/2 + i) % (2*N - 1);
    sig(i) = 1.0;

//    if( x(i) == 1 || x(i) == 2)
//      sig(i) = .001;
  }


/*
  for(j = N2/2,i = 1; i <= N2/2; i++,j--) {
    x(i) = (double)(j);
    sig(i) = 1.0;
  }
  for(j = 2*N; i <= N2; i++,j--) {
    x(i) = (double)(j);
    sig(i) = 1.0;
  }
*/

/*  f0 = (floor(x(i)/2)-1)/((N/2)*T);
    f1 = (floor((N-x(i+1))/2)+1)/((N/2)*T);      */
/*
  printf("plot of std deviations, modified for pole location of observer...\n");
  plot(sig);
 */

  // Test func out...

  Vector dyda(50);
  for(i = 1; i <= length(x); i++)
    func(x(i), a, y(i), dyda, i==1);
```

```
  printf("plot of errors returned by func\n");
  plot(y);
  y = 0.0;

  // End of func test.

  printf("\nStarting Non-linear least squares routine...\n");

  y = 0.0;              // What we want to achieve.

  Vector chi(NMAX);
  Matrix covar(length(a),length(a));
  Matrix alpha(length(a),length(a));
  Vector atry(a),beta(a),da(a);
  Matrix oneda(a.q_no_elems(),1);
  double chisq, alambda = -1.0;
  double ochisq, olambda;

  printf("\n");


  for(i = 1; i <= NMAX; i++) {
    mrqmin(x, y, sig, a, ia, covar, alpha, chisq, alambda, atry,
beta, da, oneda, func);

    printf("Chisq[%d] = %lf\n", i, chisq);
    printf("Alambda[%d] = %lf\n", i, alambda);

    // Record our convergence success
    // so it can be plotted...
    chi(i) = log( chisq );

    // Terminate?
    if( i > NMIN ) {
      if((chisq-ochisq)/chisq < 1e-5 && alambda <= olambda) {
printf("\n\nStopping. i = %d \n", i);
printf("Chisq = %lf \n", chisq);
break;
      }
    }
    ochisq  = chisq;
    olambda = alambda;
  }

  printf("\n");

  //
  // get covariance matrix of the parameters.
  //
  alambda = 0.0;
  mrqmin(x, y, sig, a, ia, covar, alpha, chisq, alambda, atry,
beta, da, oneda, func);

  dump("parameters", a, "%.3e");
```

```
  dump("covariance matrix", covar, "%.1e");


  for(i = 1; i <= length(x); i++)
    func(x(i), a, y(i), dyda, i==1);

  printf("plot of errors returned by func\n");
  plot(y);
  y = 0.0;

  Vector xyz(N);

  printf("Slip\n");

  for(i = 1; i <= N; i++)
    xyz(i) = -P(i*2,sj);

  plot(xyz);

/*
  Vector pchi(i);
  for(j = 1; j < i; j++)
    pchi(j) = chi(j);

  plot(pchi);
*/
}




//
// this is the "func" that numerical recipes sees.
//
void func(double x, Vector &a, double &y, Vector &dyda, int first)
{
  func(x,a,y,dyda,first,NULL);
}


//
// this is the initialization call.
//
void func(Matrix &P, double t, double v, double tau, double vqs, double vds)
{
  Vector x(6);

  x(1) = t;
  x(2) = v;
  x(3) = tau;
  x(4) = vqs;
  x(5) = vds;

  func(t,x,v,x,0,&(P));
```

```
}


//
// the real "func"
//


#define err(n)    ((*P)( (n), e))
#define e_p(n,i)  ((*P)( (n), e+(i)))

void func(double x, Vector &a, double &y, Vector &dyda, int first, Matrix *_P)
{
  static double T;        // Private copies of key parameters.
  static double tau;
  static Matrix *P;

  // initialize local variables. this should
  // never happen on calls from mrqmin.
  if((void *)_P != NULL) {
    P     = _P;
    T     = a(1);
    tau   = a(3);
    return;
  }

  // If this is the first call with NEW parameters, we need to
  // do some one-tie setup calculations.
  if(first == 1) {
    dump("Parameters sent to observer:\n", a, "%.3e");
    Observer((*P),a,T);
    Errors((*P),a,T,tau);
    Derivatives((*P),a,T,tau);
  }

  int i,n = (int)x;

  // Return stuff to minimization routine.
  y      =  err(n);
  for(i = 1; i <= 5; i++)
    dyda(i) = e_p(n,i);
}

#undef err
#undef e_p
```

make file

```
identify : identify.o window.o integrate.o lambda.o fourier.o mrqmin.o
```

```
estimate.o sysidtools.o matrix1.o matrix2.o vector.o myenv.o
tools.o simulate.o
g++ -o identify identify.o window.o integrate.o fourier.o
estimate.o lambda.o mrqmin.o  matrix1.o matrix2.o vector.o
myenv.o tools.o sysidtools.o simulate.o -lm -lg++

identify.o: identify.cc linalg/linalg.h
g++ -o identify.o -c -ggdb -DSTANDALONE identify.cc

lambda.o: lambda.cc lambda.h
g++ -c lambda.cc

integrate.o: integrate.cc integrate.h
g++ -c integrate.cc

window.o: window.cc window.h
g++ -c window.cc

mrqmin.o: mrqmin.cc mrqmin.h
g++ -c mrqmin.cc

fourier.o: fourier.cc fourier.h
g++ -c fourier.cc

matrix1.o: linalg/matrix1.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -ggdb -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/matrix1.cc

estimate.o: estimate.cc estimate.h
g++ -c estimate.cc

sysidtools.o: sysidtools.cc sysidtools.h
g++ -c sysidtools.cc

matrix2.o: linalg/matrix2.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/matrix2.cc

vector.o:  linalg/vector.cc linalg/linalg.h linalg/myenv.h
g++ -c -O -Wall -Wpointer-arith -fforce-mem -fforce-addr
-felide-constructors linalg/vector.cc

myenv.o:   linalg/myenv.cc  linalg/myenv.h
g++ -c linalg/myenv.cc

tools.o:   tools.cc linalg/myenv.h linalg/linalg.h tools.h
g++ -c -ggdb tools.cc

simulate.o: simulate.cc linalg/linalg.h
g++ -c -ggdb simulate.cc
```

map.h

```
// Standard column locations.
#define is    (1)
#define ir    (2)
#define vs    (3)
#define sj    (4)
#define e     (5)
#define e_xm  (6)
#define e_xl  (7)
#define e_rr  (8)
#define e_rs  (9)
#define e_J   (10)


// Standard Parameter mapping.
#define xm   (a(1))
#define xl   (a(2))
#define xl   (a(2))
#define rr   (a(3))
#define rs   (a(4))
#define J    (a(5))
```

E.2 estimate.cc

The following code may contain portions derived from Numerical Recipes

C++ source

```
//
// Estimate.cc
//
// Estimate module for induction motor identification code.
//
// Last Modified:   October, 1996.
//
//
#include <math.h>
#include "linalg/myenv.h"
#include "linalg/linalg.h"
#include "tools.h"
#include "sysidtools.h"
#include "simulate.h"
#include "lambda.h"
#include "fourier.h"
#include "integrate.h"
#include "window.h"
```

```
#include "map.h"

void mulj(Vector &x);

void detrend(Vector &x, int w1, int w2);

void plotspect(Vector &x, int N);
void alambda(Vector &x, double tau, double T);


void dottimes(Vector &x, Vector &y);
void complex2real(Vector &x, Vector &real, Vector &imag);
void real2complex(Vector &real, Vector &imag, Vector &x);
void cpv(Vector &from, Vector &to);
void Deconvolve(Vector &eq, double T) ;
void Slipest(Matrix &B, Vector &a, double T);

void gamma(Vector &x, double eps, double rx0, double ix0);
static void gammaderivs(double t, Vector &s, Vector &ds);

void Window(Vector &x);


/*

   extra routines for slip estimatation.
*/
double rtnewt(double (*func)(double), double x1,
double x2, double xacc, double init);
double rtbis(double (*func)(double), double x1, double x2, double xacc);
double funky(double x);
double funky(double beta, Vector *_torque, Vector *_z, double _T,
double _s0, int _i1, int _i2);



//
// Observer: This code estimates the rotor flux linkages
// using the first line of the motor model.
//
void Observer(Matrix &B, Vector &a, double T)
{
  double omega = M_PI*2.0*60.0;
  int N = rows(B);
  int M = N/2;
  int i,j;
  double t;
  Vector eq(N);

  // Setup "input"
  getcol(B,vs,eq);
  agetcol(B,is,eq,-rs);

  Deconvolve( eq , T );
```

```
  for(i = 1; i <= rows(B); i++) {
    eq(i) = B(i,ir) = eq(i)/xm - ((xm+xl)/xm)*B(i,is);
    B(i,e)  = xm*B(i,is) + (xm+xl)*B(i,ir);
  }

  /* Estimate the slip here... */
  Slipest(B,a,T);

  // Debugging code.
  // printf("Rotor currents...\n");
  // plot(eq);
}



/* Interface to numerical recipes routine... */
double funky(double x)
{
  funky(x,NULL,NULL,0,0,0,0);       // Mask for real function.
}

/* the real function.                          */
double funky(double beta, Vector *_torque, Vector *_s, double _T,
double _s0, int _i1, int _i2)
{
  static Vector *torque;
  static Vector *s;
  static double gamma;
  static double T,s0;
  static int N,index;
  int i;
  double dsdt;

  // Initialization
  if((void *)_torque != NULL ) {
    T      = _T;
    torque = _torque;
    s      = _s;
    s0     = _s0;


    gamma = 0;
    for(i = _i1; i <= _i2; i++)
      gamma += ((*torque)(i)) / (1.0-s0);

    gamma /= (_i2-_i1+1);
    index = (int)(_i1+_i2)/2;

    N      = length((*torque));
    return 0.0;
  }

  // Initial condition.
```

```
  (*s)(1) = 1.0;

  // Integrate according to the mechanical model.
  for(i = 1; i <= N-1; i++) {
    dsdt = -beta*(*torque)(i) + beta*gamma*(1.0-(*s)(i));
    (*s)(i+1) = (*s)(i) + dsdt*T;
  }

  // Bring this quantity to zero.
  return (*s)(index) - s0;
}


//
// Estimate the slip.
//
// This now accomodates inertia and drag using a shooting method.
//
void Slipest(Matrix &B, Vector &a, double T)
{
  int i,j,N = rows(B);
  double omega = 2*M_PI*60;
  static double beta = 0.05;
  Vector slipper(N/2);
  Vector torque(N/2);

  // Stuff the torque in here...
  for(i = 1, j = 1; i <= N; i += 2, j++)
    torque(j) = ((B(i,vs)*B(i,is)+B(i+1,vs)*B(i+1,is))
 - rs*(B(i,is)*B(i,is)+B(i+1,is)*B(i+1,is))) / omega;

//  printf("hi\n");

  // Pass parameters to funky for initialization.
  funky(0,&(torque),&(slipper),T,.005,12000,12208);

//  printf("hi\n");

  // Find the root. This is a 1-D shooting
  // method.
  beta = rtnewt(funky,0.0,1.0,1.0e-4,beta);

  printf("beta = %lf\n", beta);

//  plot(slipper);

  // Stuff the matrix.
  for(i = 1, j = 1; i <= N; i += 2, j++) {
    B(i,  sj) = 0.0;
    B(i+1,sj) = -slipper(j);
  }
}
```

```
void Deconvolve(Vector &eq, double T)
{
  Vector x(2*length(eq));
  double t;
  double omega = 2.0*60.0*M_PI;
  int M = length(eq);
  int N = length(x);
  int i,j;

  cpv(eq,x);                        // Copy Vector.

  // Get the frequencies, given the bilateral transform.
  for(i = 1; i <= M/2; i++) {
    eq(i)     = (2.0/T)*tan(  ((double)i-1)*M_PI/M );
    eq(M-i+1) = (2.0/T)*tan(  -((double)i)*M_PI/M  );
  }

  // Forward transform...
  fft(x,FORWARD);

 // plotspect(x,1024);


  // deconvolve.
  for(i=1, j=1; i <= N; i += 2, j++) {
    t       = x(i);
    x(i)   = -x(i+1);
    x(i+1) = t;
    x(i  ) = -x(i)   / ((double)M*(eq(j)/omega-1.0));
    x(i+1) = -x(i+1) / ((double)M*(eq(j)/omega-1.0));

  //  x(i)   = x(i)/(double)M;
  //  x(i+1) = x(i+1)/(double)M;
  }

  fft(x,INVERSE);                   // take the inverse transform.

  cpv(x,eq);                        // Copy Vector.
}


//
// Copy Vectors.
//
void cpv(Vector &from, Vector &to)
{
  int i,N = min(length(from),length(to));

  // Change to mem
  for(i=1;i<=N;i++)
    to(i) = from(i);

  // memcpy( ptrto, ptrfrom, sizeof(double)*N);
}
```

```
//
// Compute errors, easiest possible way.
// Fourier transform of the errors goes in Column e of B
//
void Errors(Matrix &B, Vector &a, double T, double tau)
{
  int i,j,N = rows(B);
  Vector Psi(N),err(N);
  double omega = M_PI*2.0*60.0;
  double tmp;

  // Compute the objective function
  //
  getcol(B,e,Psi);            // This is Psi...
  getcol(B,sj,err);           // Get s    -> err
  dottimes(err,Psi);          // Get sPsi -> err
  agetcol(B,ir,err,rr);       // sPsi + rr*ir -> err
  for(i = 1; i <= N; i++)
    err(i) = tau*err(i) - (Psi(i) = Psi(i)/omega);
  alambda(err,tau,T);         // lambda(sPsi + rr*ir);
  err += Psi;

 //  plot(err);

  // Window the errors. This reduces the
  // spectral leakage of our pole.
  Window( err );

//  plot(err);

  // Take the transform.
  fft(err,FORWARD);

  //  Debugging code.
/*
  printf("DFT of errors...\n");
  plotspect(err,1000);
  plotspect(err,400);
  plotspect(err,200);
*/

/*  plotspect(err,100);

  detrend(err,60,80);

  plotspect(err,100);

*/
  setcol(err,B,e);             // B(:,e) <= err(:)
}
```

```
void Window(Vector &x)
{
  HanningWindow(x);
}



//
// Put the partials of the DFT of the errors into the Matrix B.
//
// The partials of the DFT of the errors is the DFT of the partials.
//
void Derivatives(Matrix &B, Vector &a, double T, double tau)
{
  int i,j,k,N = rows(B);
  Vector x(N),y(N),tm(N);
  double omega = M_PI*2.0*60.0;

  ///////////////
  // Compute e_rr
  getcol(B,ir,x,tau);            // x       <=   ir*tau
// alambda(x,tau,T);              // x       <=   lambda( x )
  Window(x);                     // Window it.
  fft(x,FORWARD);                // x       <=   FFT (x)
  setcol(x,B,e_rr);              // e_rr    <=   x

  ///////////////
  // Compute e_xm
/*  for(i = 1; i <= N; i++) {
    y(i) = B(i,is) - ((xm+xl)/xm)*B(i,ir);
    x(i) = y(i) / omega;
  }
//  agetcol(B,sj,tm);
  getcol(B,sj,tm);
  dottimes(tm,y);
  for(i = 1; i <= N; i++)
    y(i) = (-B(i,ir)*rr/xm + tm(i))*tau - x(i);
  alambda(y,tau,T);
  x += y;
  Window(x);
  fft(x,FORWARD);
  setcol(x,B,e_xm);
*/
  getcol(B,is,x,-1.0/xm);                  // x has   -(ir+is)/xm
  agetcol(B,ir,x,-1.0/xm);
  getcol(B,sj,tm, xl);
  dottimes(tm,x);                          // tm has (p/w - sj) part.
  x *= rr;                                 // rr part.
  x += tm;                                 // sum
  Window(x);
  fft(x,FORWARD);
  setcol(x,B,e_xm,tau);
```

```
  ////////////////
  // Compute e_xl
  getcol(B,is,x,-(1.0+xl/xm));
  getcol(B,is,y,-rr/xm);
  agetcol(B,ir,x);
  getcol(B,sj,tm);
  dottimes(tm,x);
  y += tm;
  Window(y);
  fft(y,FORWARD);
  setcol(y,B,e_xl,tau);

  ////////
  // Compute e_rs
  getcol(B,is,x,-(xm+xl)/xm);
  getcol(B,sj,tm);
  dottimes(x,tm);                 // x has part due to s term.
  getcol(B,is,y,-rr/xm);
  x += y;
  mulj(x);
  Window(x);
  fft(x,FORWARD);
  setcol(x,B,e_rs,tau);

  ///////
  // Compute e_J
/*
   double slip;

  // Put d slip/ dJ in X
  for(i = 1; i <= N; i += 2) {
    slip = -B(i+1,sj);

    x(i) = (slip-1)/J;
    x(i+1) = (slip-1)/J;
  }

  getcol(B,is,y,-xm);
  agetcol(B,ir,y,-(xl+xm));
  mulj(y);
  dottimes(x,y);
  Window(x);
  fft(x,FORWARD);
  setcol(x,B,e_J,tau);
*/
}

void plotspect(Vector &x, int N)
{
  int i,j;
  Vector spect(N);
```

```
  for(j = 1, i = length(x)-N/2; i <= length(x); i++, j++)
    spect(j) = x(i);

  for(i = 1; j <= N; j++, i++)
    spect(j) = x(i);

 plot(spect);
}


// void lambda(Vector &x, double tau, double T, double eps,
// double rx0, double ix0);

//
// apply lambda to complex vector.
//
void alambda(Vector &x, double tau, double T)
{
  lambda(x,tau,T,1.0e-6,0.0,0.0);
}


//
// Interpret Vector as complex; multiply by j.
//
void mulj(Vector &x)
{
  int i,N = length(x);
  double tmp;

  for(i = 1; i <= N; i += 2) {
    tmp    = x(i);
    x(i)   = -x(i+1);
    x(i+1) = tmp;
  }
}



//  x <- x .* sj;
//
// Do elementwise multiplication of X and SJ
//
// Vectors are interpreted as being complex.
//
void dottimes(Vector &x, Vector &y)
{
  int i,N = length(x);
  double xi;

  for(i=1; i <= N; i += 2) {
    x(i) = (xi=x(i))*y(i) - x(i+1)*y(i+1);
    x(i+1) = x(i+1)*y(i) + xi*y(i+1);
  }
```

```
}

void complex2real(Vector &x, Vector &real, Vector &imag)
{
  int i,j,N = length(x);

  for(i = 1, j = 1; i <= N; i += 2, j++) {
    real(j) = x(i);
    imag(j) = x(i+1);
  }
}

void real2complex(Vector &real, Vector &imag, Vector &x)
{
  int i,j,N = length(x);

  for(i = 1, j = 1; i <= N; i += 2, j++) {
    x(i) = real(j);
    x(i+1) = imag(j);
  }
}




static double ri1,ii1,ri2,ii2;

////////////////////////////
// Apply to complex vector.
//
void gamma(Vector &x, double eps, double rx0, double ix0)
{
  int N = length(x);
  Vector st(2);
  Vector ds(2);
  int nok,nbad;
  int i,j;

  st(1) = rx0;
  st(2) = ix0;

  for(i = 1; i <= N; i += 2) {
    ri1 = x(i  );
    ii1 = x(i+1);

    if( i+3 <= N ) {
      ri2 = x(i+2);
      ii2 = x(i+3);
    } else {
      ri2 = ri1;
      ii2 = ii1;
    }
```

```
    // Integrate (in dimensionless time...)
    odeint(st,0.0,1.0,eps,1.0,0.0,nok,nbad,gammaderivs,rkqs);

    x(i)    = st(1);
    x(i+1)  = st(2);
  }
}

// Calculate derivatives, imaginary simulation.
static void gammaderivs(double t, Vector &st, Vector &ds)
{
  double ri,ii;
  static double omega = M_PI*2.0*60.0;

  ri = t*ri2 + (1.0-t)*ri1;        // t is dimensionless.
  ii = t*ii2 + (1.0-t)*ii1;        // linear interpolate me.

  ds(1) = omega*(ri - st(2));
  ds(2) = omega*(ii + st(1));
}


//
// Interpolate in the interval
//
//
void detrend(Vector &x, int w1, int w2)
{
/*
  int i,j;
  int N = (w2-w1) << 1;
  Matrix A(N,2);
  Vector b(N);
  Vector c(2);

  // Stuff the matrices.
  for(i = -w2, j = 1; i <= -w1; i++, j++) {
    A(j,1) = 1.0;
    A(j,2) = i;
    b(j) =   x(length(x) + i);
  }

  for(i = w1; i <= w2; i++, j++) {
    A(j,1) = 1.0;
    A(j,2) = i;
    b(j) =   x(i);
  }

  normaleqn(A,c,b);

  // Subtract the interpolant.
  for(i = -w2; i <= 0; i++)
    x(i+length(x)) -= c(2)*i+c(1);
```

```
  for(i = 1; i <= w2; i++)
    x(i) -= c(1)*i+c(1);
*/
}


#define JMAX 20
double rtnewt(double (*func)(double), double x1, double x2, double xacc,
double init)
{
  double df,dx,f,rtn,inc;
  int j;

  rtn = init;                                // Initial guess
  dx  = 1.0e-3;

  for (j = 1; j <= JMAX; j++) {
    f = (*func)(rtn);
    df = (*func)(rtn + dx) - f;

    inc = f*(dx/df);

    rtn -= inc;

    if ((x1-rtn)*(rtn-x2) < 0.0)
      return rtbis(func,x1,x2,xacc);

    if (fabs(inc) < xacc)
      return rtn;
   }

  return rtbis(func,x1,x2,xacc);
}
#undef JMAX



#define JMAX 40
double rtbis(double (*func)(double), double x1, double x2, double xacc)
{
  int j;
  double dx,f,fmid,xmid,rtb;

  f=(*func)(x1);
  fmid=(*func)(x2);
  if (f*fmid >= 0.0) {
    printf("Root must be bracketed for bisection in rtbis");
    return 0.0;
  }
  rtb = f < 0.0 ? (dx=x2-x1,x1) : (dx=x1-x2,x2);
  for (j=1;j<=JMAX;j++) {
    fmid=(*func)(xmid=rtb+(dx *= 0.5));
    if (fmid <= 0.0) rtb=xmid;
    if (fabs(dx) < xacc || fmid == 0.0)
```

```
      return rtb;
  }
  printf("Too many bisections in rtbis");
  return 0.0;
}
#undef JMAX

/*

//
// This is probably the most computationally efficient way
// to do this.
//
void fftlambda(double tau, double T, Vector &x)
{
  int N = length(x), i;
  Vector xzp(N << 1);
  double rp,ip,den,s;

  // Create zero padded X.
  for(i = 1; i <= N; i++) {
    xzp(i+N) = x(i);
    xzp(i) = 0.0;
  }

  // Take the DFT
  fft(xzp,FORWARD);

  // Apply the lambda operator in the frequency
  // domain.
  for(i = 1; i <= (N<<1); i += 2) {
    s = M_PI*2.0*(((i<=(N+1))?(i-1):((i-1)-(N<<1)))>>1)/(N*T);

    den = (1.0+tau*tau*s*s);
    rp = xzp(i+0) - tau*s*xzp(i+1);
    ip = xzp(i+1) + tau*s*xzp(i+0);
    xzp(i+0) = rp/den;
    xzp(i+1) = ip/den;
  }

  // Inverse DFT
  fft(xzp,INVERSE);
  xzp *= (1.0/N);

  // Copy uncorrupted data.
  for(i = 1; i <= N; i++)
    x(i) = zxp(i+N);
}

*/
```

include file

```
void Errors(Matrix &B, Vector &p, double T, double tau);
void Derivatives(Matrix &B, Vector &p, double T, double tau);
void Observer(Matrix &B, Vector &p, double T);
```

Appendix F

General purpose codes

This appendix contains listings of the general purpose codes used throughout this thesis. Many of these codes are somewhat modified versions of like-named routines in *Numerical Recipes in C.*

F.1 `fourier.cc`

This module contains basic FFT tools.

The following code may contain portions derived from Numerical Recipes C++ Source

```
#include <math.h>
#include "linalg/myenv.h"
#include "linalg/linalg.h"
#include "tools.h"
#include "fourier.h"

void four1(double *data, unsigned int nn, int isign);

#ifdef STANDALONE
main()
{
  Vector x(1024*8);
  int i;

  for(i = 1; i <= length(x); i++)
    x(i) = sin(2*M_PI*i/250) + .3*sin(2*M_PI*i/20);
```

```
  plot(x);                        // Plot this.
  realfft( x, FORWARD );          // Take the forward transform
  plot(x);                        // Plot result.

  for(i = 500; i <= 1500; i++)
    x(i) = 0.0;

  realfft(x, INVERSE );          // Inverse FFT
  x *= 2.0/length(x);            // Scale.

  plot(x);

  printf("have a nice day!\n");
}
#endif




//
// Compute the DTFT of two real-valued vectors simulatneously.
//
// length(fft1) = 2*length(data1)
//
void twofft(Vector &_data1, Vector &_data2, Vector &_fft1, Vector &_fft2)
{
  unsigned long n = length(_data1);
  double *data1 = _data1.get_pointer();
  double *data2 = _data2.get_pointer();
  double *fft1  = _fft1.get_pointer();
  double *fft2  = _fft2.get_pointer();

  if( n != length(_data2) ) {
    printf("twofft: Error. input vectors must have same length!\n");
    return;
  }

  if( 2*n > min(length(_fft1), length(_fft2))) {
    printf("Error: output arrays must have twice input array length.\n");
    return;
  }

  unsigned long nn3,nn2,jj,j;
  double rep,rem,aip,aim;

  nn3=1+(nn2=2+n+n);
  for (j=1,jj=2;j<=n;j++,jj+=2) {
    fft1[jj-1]=data1[j];
    fft1[jj]=data2[j];
  }
  four1(fft1,n,1);
  fft2[1]=fft1[2];
  fft1[2]=fft2[2]=0.0;
```

```
  for (j=3;j<=n+1;j+=2) {
    rep=0.5*(fft1[j]+fft1[nn2-j]);
    rem=0.5*(fft1[j]-fft1[nn2-j]);
    aip=0.5*(fft1[j+1]+fft1[nn3-j]);
    aim=0.5*(fft1[j+1]-fft1[nn3-j]);
    fft1[j]=rep;
    fft1[j+1]=aim;
    fft1[nn2-j]=rep;
    fft1[nn3-j] = -aim;
    fft2[j]=aip;
    fft2[j+1] = -rem;
    fft2[nn2-j]=aip;
    fft2[nn3-j]=rem;
  }
}


#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void realfft(Vector &_data, int isign)
{
  unsigned long n = length(_data);
  double *data = _data.get_pointer();

  unsigned long i,i1,i2,i3,i4,np3;
  double c1=0.5,c2,h1r,h1i,h2r,h2i;
  double wr,wi,wpr,wpi,wtemp,theta;

  theta= (M_PI)/(double) (n>>1);
  if (isign == FORWARD) {
    c2 = -0.5;
    four1(data,n>>1,1);
  } else {
    c2=0.5;
    theta = -theta;
  }
  wtemp=sin(0.5*theta);
  wpr = -2.0*wtemp*wtemp;
  wpi=sin(theta);
  wr=1.0+wpr;
  wi=wpi;
  np3=n+3;
  for (i=2;i<=(n>>2);i++) {
    i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
    h1r=c1*(data[i1]+data[i3]);
    h1i=c1*(data[i2]-data[i4]);
    h2r = -c2*(data[i2]+data[i4]);
    h2i=c2*(data[i1]-data[i3]);
    data[i1]=h1r+wr*h2r-wi*h2i;
    data[i2]=h1i+wr*h2i+wi*h2r;
    data[i3]=h1r-wr*h2r+wi*h2i;
    data[i4] = -h1i+wr*h2i+wi*h2r;
    wr=(wtemp=wr)*wpr-wi*wpi+wr;
    wi=wi*wpr+wtemp*wpi+wi;
```

```
  }
  if (isign == FORWARD) {
    data[1] = (h1r=data[1])+data[2];
    data[2] = h1r-data[2];
  } else {
    data[1]=c1*((h1r=data[1])+data[2]);
    data[2]=c1*(h1r-data[2]);
    four1(data,n>>1,-1);
  }
}

void fft( Vector &x, int isign)
{
  unsigned long n = length(x);
  double *data = x.get_pointer();

  four1(data,n>>1,(isign == FORWARD) ? 1 : -1);
}


// Provide a frequency "axis" for realfft.
void getfreqs(Vector &f, double T)
{
  int N = length(f), i, j;

  T /= (2.0*M_PI);            // f is rads/sec

  for(i = 3, j = 1; i <= N; i += 2, j++) {
    f(i  ) = (double)j;
    f(i+1) = (double)j;
  }
  f(1) = 0.0;
  f(2) = N/2.0;

  f *= 1.0 / (N*T);
}



/*
** Fast Fourier Transform Code (DTFT)
**
** This is written in pure C, and should not suffer a performance
** penalty due to the C++ bells and whistles.
**
*/
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr
void four1(double *data, unsigned int nn, int isign)
{
  unsigned long n,mmax,m,j,istep,i;
  double wtemp,wr,wpr,wpi,wi,theta;
  double tempr,tempi;
  n=nn << 1;
  j=1;
```

```
  // Bit reversal.
  for (i=1;i<n;i+=2) {
    if (j > i) {
      SWAP(data[j],data[i]);
      SWAP(data[j+1],data[i+1]);
    }
    m=n >> 1;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    }
    j += m;
  }

  // Butterflies.
  mmax=2;
  while (n > mmax) {
    istep=mmax << 1;
    theta=isign*( (2.0*M_PI)/mmax);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for (m=1;m<mmax;m+=2) {
      for (i=m;i<=n;i+=istep) {
j=i+mmax;
tempr=wr*data[j]-wi*data[j+1];
tempi=wr*data[j+1]+wi*data[j];
data[j]=data[i]-tempr;
data[j+1]=data[i+1]-tempi;
data[i] += tempr;
data[i+1] += tempi;
      }
      wr=(wtemp=wr)*wpr-wi*wpi+wr;
      wi=wi*wpr+wtemp*wpi+wi;
    }
    mmax=istep;
  }
}
#undef SWAP
```

Include file

```
#define   FORWARD  (1)
#define   INVERSE  (-1)
```

```
void realfft(Vector &_data, int isign);
void twofft(Vector &_data1, Vector &_data2, Vector &_fft1, Vector &_fft2);
void getfreqs(Vector &f, double T);
void fft( Vector &x, int isign);
```

F.2 lambda.cc

This module implements the λ operator described in Chapter 1.

C++ Source

```
//
//     ***
//       **
//        **
//         **
//        * **
//       *   **
//     **     ***
//
//
// Requires odeint and rkqs.
// Simulates a vector.
//
// Steven R. Shaw, Summer 1996
//
// Modified:        Fall 1996
//                  Added interpretation of vector as complex.
//

#include <math.h>
#include "linalg/linalg.h"
#include "integrate.h"
#include "lambda.h"
#include "tools.h"

// Private stuff.

static void realderivs(double t, Vector &s, Vector &ds);
static void imagderivs(double t, Vector &s, Vector &ds);


#ifdef STANDALONE

main()
{
```

```
  Vector x(1024);
  int i,N = length(x);

  for(i = 512; i<= N; i++)
    x(i) = 1.0;

  lambda(x,50.0,1.0,1.0e-6,0.0);

  plot(x);
}

#endif

//
// These guys are private.
//
static double ri1,ii1,ri2,ii2;
static double alpha;


//////////////////////
// Apply to real vector.
//
void lambda(Vector &x, double tau, double T, double eps, double rx0)
{
  int N = length(x);
  Vector s(1), ds(1);
  int nok,nbad;
  int i,j;

  s(1) = rx0;
  alpha = T/tau;

  for(i = 1; i <= N; i++) {
    ri1 = x(i);

    if( i+1  <= N)
      ri2 = x(i+1);
    else
      ri2 = ri1;

    // Integrate.
    odeint(s,0.0,1.0,eps,1.0,0.0,nok,nbad,realderivs,rkqs);

    x(i) = s(1);
  }
}




//////////////////////////
// Apply to complex vector.
//
```

```
void lambda(Vector &x, double tau, double T, double eps, double rx0, double ix0)
{
  int N = length(x);
  Vector s(2), ds(2);
  int nok,nbad;
  int i,j;

  s(1) = rx0;
  s(2) = ix0;

  alpha = T/tau;

  for(i = 1; i <= N; i += 2) {
    ri1 = x(i  );
    ii1 = x(i+1);

    if( i+3 <= N ) {
      ri2 = x(i+2);
      ii2 = x(i+3);
    } else {
      ri2 = ri1;
      ii2 = ii1;
    }

    // Integrate (in dimensionless time...)
    odeint(s,0.0,1.0,eps,1.0,0.0,nok,nbad,imagderivs,rkqs);

    x(i)    = s(1);
    x(i+1)  = s(2);
  }
}



// Calculate derivatives, real simulation.
static void realderivs(double t, Vector &s, Vector &ds)
{
  double ri;

  ri = t*ri2 + (1.0-t)*ri1;        // t is dimensionless.
  ds(1) = (ri - s(1))*alpha;
}



// Calculate derivatives, imaginary simulation.
static void imagderivs(double t, Vector &s, Vector &ds)
{
  double ri,ii;

  ri = t*ri2 + (1.0-t)*ri1;        // t is dimensionless.
  ii = t*ii2 + (1.0-t)*ii1;

  ds(1) = (ri - s(1))*alpha;
```

```
  ds(2) = (ii - s(2))*alpha;
}
```

Include file

```
// LAMBDA.H
void lambda(Vector &x, double tau, double T, double eps,
double rx0, double ix0);
void lambda(Vector &x, double tau, double T, double eps, double rx0);
```

F.3 mrqmin.cc

This module implements the Levenburg Marquardt algorithm. It is similar to the like-named Numerical Recipes routine, except that it has modifications for C++.

The following code may contain portions derived from Numerical Recipes C++ Source

```
#include <stdio.h>
#include <math.h>
#include "linalg/myenv.h"
#include "linalg/linalg.h"
#include "mrqmin.h"

//
// Keep all these guys local...
//
void mrqcof(Vector &x, Vector &y, Vector &sig, Vector &a, int ia[],
    Matrix &alpha, Vector &beta, double &chisq,
    void(*funcs)(double, Vector &, double &, Vector &, int));
void covsrt(Matrix &covar, int ma, int ia[], int mfit);
void gaussj(Matrix &a, int n, Matrix &b, int m);
void free_ivector(int *v, long nl, long nh);
int *ivector(long nl, long nh);

//
//
// This is the Levenburg-Marquardt method for non-linear least squares
//
```

```
// It is borrowed largely from the Numerical Recipes Codes,
// found in /mit/recipes on Athena.
//
// func(double x, Vector &a, double &y, Vector &dyda, int first)
// is the user supplied non-linear function containing the
// regressors.
//
// Arguments are x, Vector &a and first; the x value and the parameters vector.
// first == 1 whenever func is called with a NEW a vector. Useful for
// initializations, etc.
//
// func computes Y (the output) and dyda (partials of Y wrt a[1]...a[n]).
//
//
void mrqmin(Vector &x, Vector &y, Vector &sig, Vector &a, int ia[],
Matrix &covar, Matrix &alpha, double &chisq,
    double &alamda, Vector &atry, Vector &beta, Vector &da,
Matrix &oneda,
    void(*funcs)(double, Vector &, double &, Vector &, int))
{
  int ndata = y.q_no_elems();
  int ma    = a.q_no_elems();

  int j,k,l;
  static int mfit;
  static double ochisq;

  // Initialization.
  if(alamda < 0.0) {
    for(mfit = 0,j = 1;j <= ma; j++) if (ia[j]) mfit++;
    alamda = 100.00;
    mrqcof(x,y,sig,a,ia,alpha,beta,chisq,funcs);
    ochisq = chisq;
    atry = a;
  }

  for (j=1;j<=mfit;j++) {
    for(k=1;k<=mfit;k++)
      covar(j,k) = alpha(j,k);
    covar(j,j) = alpha(j,j)*(1.0+alamda);
    oneda(j,1) = beta(j);
  }

  gaussj(covar,mfit,oneda,1);

  for(j=1; j <= mfit; j++)
    da(j) = oneda(j,1);

  if(alamda == 0.0) {
    covsrt(covar,ma,ia,mfit);
    return;
  }

  for(j = 0,l = 1; l <= ma; l++)
```

```
    if (ia[l])
      atry(l)=a(l)+da(++j);

  mrqcof(x,y,sig,atry,ia,covar,da,chisq,funcs);

  if(chisq < ochisq) {
    alamda *= 0.1;
    ochisq=chisq;

    for(j = 1; j <= mfit; j++) {
      for (k = 1; k <= mfit; k++)
alpha(j,k)=covar(j,k);
      beta(j)=da(j);
    }
    a = atry;
  } else {
    alamda *= 10.0;
    chisq   = ochisq;
  }
}



void mrqcof(Vector &x, Vector &y, Vector &sig, Vector &a, int ia[],
    Matrix &alpha, Vector &beta, double &chisq,
    void(*funcs)(double, Vector &, double &, Vector &, int))
{
  int ndata = y.q_no_elems();
  int ma    = a.q_no_elems();

  int i,j,k,l,m,mfit=0;
  double ymod,wt,sig2i,dy;

  Vector dyda(a);

  for(j = 1; j <= ma; j++)  if(ia[j]) mfit++;

  for(j = 1; j <= mfit; j++) {
    for(k = 1; k <= j; k++)
      alpha(j,k) = 0.0;
    beta(j) = 0.0;
  }

  chisq = 0.0;

  for(i = 1; i <= ndata; i++) {
    (*funcs)(x(i),a,ymod,dyda,(i==1));

    sig2i = 1.0/(sig(i)*sig(i));
    dy    = y(i)-ymod;

    for(j = 0, l = 1; l <= ma; l++) {
      if (ia[l]) {
```

```
wt = dyda(l)*sig2i;
for(j++, k = 0, m = 1; m <= l; m++)
  if (ia[m])
    alpha(j,++k) += wt*dyda(m);
beta(j) += dy*wt;
      }
    }
    chisq += dy*dy*sig2i;
  }

  for(j = 2; j <= mfit; j++)
    for(k = 1; k < j; k++)
      alpha(k,j) = alpha(j,k);
}


//
// Below here....
//

#define NR_END 1
#define FREE_ARG char*
int *ivector(long nl, long nh)
{
int *v;

v=(int *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(int)));
if (!v) {
  printf("MRQMIN: allocation failure in ivector()");
  exit(1);
}
return v-nl+NR_END;
}


void free_ivector(int *v, long nl, long nh)
{
free((FREE_ARG) (v+nl-NR_END));
}
#undef NR_END
#undef FREE_ARG

#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}

void gaussj(Matrix &a, int n, Matrix &b, int m)
{
  int *indxc,*indxr,*ipiv;
  int i,icol,irow,j,k,l,ll;
  double big,dum,pivinv,temp;

  indxc=ivector(1,n);
  indxr=ivector(1,n);
  ipiv=ivector(1,n);
```

```
  for (j=1;j<=n;j++) ipiv[j]=0;
  for (i=1;i<=n;i++) {
    big=0.0;
    for (j=1;j<=n;j++)
      if (ipiv[j] != 1)
for (k=1;k<=n;k++) {
  if (ipiv[k] == 0) {
    if (fabs(a(j,k)) >= big) {
      big=fabs(a(j,k));
      irow=j;
      icol=k;
    }
  } else if (ipiv[k] > 1) {
    printf("MRQMIN:gaussj: Singular Matrix-1");
    exit(1);
  }
}
    ++(ipiv[icol]);
    if (irow != icol) {
      for (l=1;l<=n;l++)
SWAP(a(irow,l),a(icol,l));

      for (l=1;l<=m;l++)
  SWAP(b(irow,l),b(icol,l));
    }

    indxr[i]=irow;
    indxc[i]=icol;

    if (a(icol,icol) == 0.0) {
      printf("MRQMIN:gaussj: Singular Matrix-2");
      exit(1);
    }

    pivinv=1.0/a(icol,icol);
    a(icol,icol)=1.0;
    for (l=1;l<=n;l++) a(icol,l) *= pivinv;
    for (l=1;l<=m;l++) b(icol,l) *= pivinv;
    for (ll=1;ll<=n;ll++)
      if (ll != icol) {
dum=a(ll,icol);
a(ll,icol)=0.0;
for (l=1;l<=n;l++) a(ll,l) -= a(icol,l)*dum;
for (l=1;l<=m;l++) b(ll,l) -= b(icol,l)*dum;
      }
  }
  for (l=n;l>=1;l--) {
    if (indxr[l] != indxc[l])
      for (k=1;k<=n;k++)
SWAP(a(k,indxr[l]),a(k,indxc[l]));
  }

  free_ivector(ipiv,1,n);
  free_ivector(indxr,1,n);
```

```
  free_ivector(indxc,1,n);
}

#undef SWAP(a,b)



#define SWAP(a,b) {swap=(a);(a)=(b);(b)=swap;}

void covsrt(Matrix &covar, int ma, int ia[], int mfit)
{
  int i,j,k;
  double swap;

  for(i = mfit+1; i <= ma; i++)
    for(j = 1; j <= i; j++)
      covar(i,j) = covar(j,i)=0.0;

  k = mfit;
  for(j = ma; j >= 1; j--) {
    if(ia[j]) {
      for(i = 1; i <= ma; i++)
SWAP(covar(i,k),covar(i,j));
      for(i = 1; i <= ma; i++)
SWAP(covar(k,i),covar(j,i));
      k--;
    }
  }
}
#undef SWAP
```

Include file

```
void mrqmin(Vector &x, Vector &y, Vector &sig, Vector &a, int ia[],
Matrix &covar, Matrix &alpha, double &chisq, double &alamda,
Vector &atry, Vector &beta, Vector &da, Matrix &oneda,
    void(*funcs)(double, Vector &, double &, Vector &, int));
```

F.4 sysidtools.cc

This module is a general collection of useful routines.

The following code may contain portions derived from Numerical Recipes

C++ Source

```
//
// SYSIDTOOLS.CC
//

#include <stdio.h>
#include <math.h>
#include "linalg/myenv.h"
#include "linalg/linalg.h"
#include "tools.h"
#include "sysidtools.h"

//
// Solve the least squares problem by the
// normal equations.
//
//
// The condition number of the matrix N is likely
// to be bad. Hence we will try save things to some
// extent by using iterative improvement.
//
//
void LeastSquares(Matrix &A, Matrix &x, Matrix &B)
{
  normaleqn(A,x,B);
}

//
// LeastSquares
//
// Solve the weighted least squares problem, where the
// wieghting matrix is diagonal. (i.e. the measurements
// are uncorrelated, but have individual variances)
//
//
void LeastSquares(Matrix &A, Matrix &x, Matrix &B, Vector &w)
{
  Matrix WA(A), WB(B);
  int i,j;
  int N = A.q_nrows(), MA = A.q_ncols(), MB = B.q_ncols();

  WA = A;
  WB = B;

  for(i = 1; i <= N; i++) {
    for(j = 1; j <= MA; j++)
      WA(i,j) *= w(i);

    for(j = 1; j <= MB; j++)
      WB(i,j) *= w(i);
```

```
  }

  LeastSquares(WA,x,WB);
}


//
// Solve square Vandermonde system.
//
// Routine borrowed from Numerical Recipes, Page 92.
//
void Vandermonde(Vector &x, Vector &w, Vector &q)
{
  double  b,s,t,xx;
  int i,j,k,k1,n = x.q_no_elems();
  Vector c(n);

  if (n == 1)
    w(1) = q(1);
  else {
    c(n) = -x(1);

    for (i = 2; i <= n; i++){
      xx = -x(i);
      for (j = (n+1-i); j <= (n-1); j++)
c(j) += xx*c(j+1);
      c(n) += xx;
    }

    for (i = 1; i <= n; i++) {
      xx = x(i);
      t = b = 1.0;
      s = q(n);
      k = n;
      for (j = 2; j <= n; j++) {
k1 = k-1;
b = c(k)+xx*b;
s += q(k1)*b;
t = xx * t+b;
k = k1;
      }
      w(i) = s/t;
    }
  }
}


//
// Solve overdetermined Vandermonde System
//
// Routine written by Steve Shaw, February 15, 1996.
//
// q here is a large vector, and w and x are relatively short.
//
```

```
void Vandermonde2(Vector &x, Vector &w, Vector &q)
{
  int n1 = q.q_no_elems();
  int n2 = x.q_no_elems();
  int i,j,k;
  Vector xx(n2);
  Matrix A(n2,n2);

  xx = 1.0;
  w = 0.0;

  for(i = 1; i <= n1; i++) {
    // Accumulate the matrix.
    for(j = 1; j <= n2; j++)
      for(k = 1; k <= n2; k++)
A(j,k) += xx(j)*xx(k);

    // Accumulate the RHS
    for(j = 1; j <= n2; j++)
      w(j) += q(i) * xx(j);

    // Multiply to get next power...
    for(j = 1; j <= n2; j++)
      xx(j) = xx(j) * x(j);
  }

  lusolve(A,w);
}

//
// Scammed from numerical recipes.  Converted to Vector code.
//
void rk4(Vector &y, Vector &dydx, double x, double h, Vector &yout,
 void (*derivs)(double, Vector &, Vector &))
{
  int n = y.q_no_elems();
  double xh,hh,h6;
  Vector dym(n),dyt(n),yt(n);

  hh=h*0.5;
  h6=h/6.0;
  xh=x+hh;

  yt =  dydx;
  yt *= hh;
  yt += y;

  (*derivs)(xh,yt,dyt);

  yt = dyt;
  yt *= hh;
  yt += y;

  (*derivs)(xh,yt,dym);
```

```
  yt = dym;
  yt *= h;
  yt += y;
  dym += dyt;

  (*derivs)(x+h,yt,dyt);

  // Save y_in in case y_out <=> y_in
  yt = y;

  yout = dym;
  yout *= 2;
  yout += dyt;
  yout += dydx;
  yout *= h6;
  yout += yt;
}
```

Include file

```
void Vandermonde2(Vector &x, Vector &w, Vector &q);
void Vandermonde(Vector &x, Vector &w, Vector &q);
void LeastSquares(Matrix &A, Matrix &x, Matrix &B);
void LeastSquares(Matrix &A, Matrix &x, Matrix &B, Vector &w);
void rk4(Vector &y, Vector &dydx, double x, double h, Vector &yout,
 void (*derivs)(double, Vector &, Vector &));
```

F.5 tools.cc

This module is another collection of useful routines.

The following code may contain portions derived from Numerical Recipes C++ Source

```
//
// TOOLS.CC
//

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

```
#include "linalg/myenv.h"
#include "linalg/linalg.h"
#include "tools.h"


//
// Form the inverse of A, replace A with it´s inverse.
//
void inv(Matrix &A)
{
  Matrix X(A);                      // Make a matrix that´s the same size
                                    // but is filled with zeros.
  X = A;

  lusolve(X, A.unit_matrix());
}


//
// Solve the overdetermined system Ax = B
// using the normal equations. This is the
// fastest way (in programmer time), but is not a good idea if
// A is ill-conditioned.
//
//
void normaleqn(Matrix &A, Matrix &X, Matrix &B)
{
  Matrix AT(A.q_ncols(),A.q_nrows());
  Matrix E(A.q_ncols(), A.q_ncols());

  AT = A.transpose();

  E = AT * A;
  X = AT * B;

  lusolve(E,X,3);          // LU solve with 2 iterative improvements
}




//
//   Solve the system of equations Ax = B
//
//   Uses LU decomposition and backsubstitution.
//
//   Side Effects/Notes:
//
//   A is replaced with the LU decompositi
//   B is replaced with the answers.
//   A must be square
//   A must have same number of rows as B.
void lusolve(Matrix &A, Matrix &B, int n)
{
```

```
  double b;
  int i;

  // Check compatibility.
  assure(A.q_nrows() == B.q_nrows(),
"LU Solve: Matrix A and B must have same number of Rows");
  assure(A.q_nrows() == A.q_ncols(),
"LU Solve: Matrix A must be square");

  // Allocate index
  int *indx = (int *)malloc(sizeof(int) * (A.q_nrows()+2));
  assert( indx != NULL );

  Matrix Ac = A;
  Matrix Bc = B;

  // Perform Decomposition
  ludcmp(A,indx,b);

  // Now Backsubstitute
  lubksb(A,B,indx);

  // Do required number of iterative improvements...
  for(i = 1; i <= n; i++)
    mprove(Ac, A, indx, Bc, B);

  free( indx );
}




//  LU Decomposition
//  Translated from Numerical recipes code to C++
//  Assumes use of the LINALG package.
#define TINY 1.0e-20;
void ludcmp(Matrix &A, int *indx, double &d)
{
  int n = A.q_ncols();
  int i,imax,j,k;
  double  big,dum,sum,temp;
  Vector vv(1,n);

  d=1;

  for (i=1;i<=n;i++) {
    big=0.0;
    for (j=1;j<=n;j++) {
      temp = fabs(A(i,j));
      if(temp > big)
big = temp;
    }
    assure(big != 0, "Singular Matrix in LUDCMP");
    vv(i)=1.0/big;
```

```
  }

  for (j=1;j<=n;j++) {
    // Equation 2.3.12 and 2.3.13
    for (i=1;i<j;i++) {
      sum=A(i,j);
      for (k=1;k<i;k++)
sum -= A(i,k)*A(k,j);
      A(i,j)=sum;
    }

    big=0;

    for (i=j;i<=n;i++) {
      sum=A(i,j);
      for (k=1;k<j;k++)
          sum -= A(i,k)*A(k,j);
      A(i,j)=sum;
      if((dum=vv(i)*fabs(sum)) >= big) {
big=dum;
imax=i;
      }
    }

    if (j != imax) {
      for (k=1;k<=n;k++) {
dum=A(imax,k);
A(imax,k)=A(j,k);
A(j,k)=dum;
      }
      d = -d;
      vv(imax)=vv(j);
    }

    indx[j]=imax;

    if(A(j,j) == 0)
      A(j,j) = TINY;

    if (j != n) {
      dum=1/(A(j,j));
      for (i=j+1;i<=n;i++)
A(i,j) *= dum;
    }
  }
}
#undef TINY



//  LU Backsubstitution
//  Translated from Numerical recipes code to C++
//  Assumes use of the LINALG package.
void lubksb(Matrix &A, Matrix &B, int *indx)
```

```
{
  int i,ii=0,ip,j;
  double sum;
  int k;
  int n = A.q_nrows();

  for(k = 1; k <= B.q_ncols(); k++) {
    for (i = 1; i <= n; i++) {
      ip=indx[i];
      sum=B(ip,k);
      B(ip,k)=B(i,k);

      if (ii)
for (j=ii;j<=i-1;j++)
  sum -= A(i,j)*B(j,k);
      else if (sum)
ii=i;

      B(i,k) = sum;
    }

    for (i = n; i >= 1; i--) {
      sum=B(i,k);
      for (j=i+1;j<=n;j++)
sum -= A(i,j)*B(j,k);
      B(i,k)=sum/A(i,i);
    }
  }
}


void dump(char *t, Vector &x, FILE *f)
{
  int i;

  if(t != NULL)
    fprintf(f, "%s:\n", t);

  for(i = x.q_lwb(); i <= x.q_upb(); i++)
    fprintf(f, "%lf\n", x(i) );
}

void dump(char *t, Matrix &A, char *fmt, FILE *f)
{
  int i,j;

  if(t != NULL)
    fprintf(f, "%s:\n", t);

  for(j = 1; j <= A.q_nrows(); j++) { // rows.
    for(i = 1; i <= A.q_ncols(); i++) { // columns.
      fprintf(f, fmt, A(j,i));
      fprintf(f,"\t");
    }
```

```
    fprintf(f,"\n");
  }
}


// Iterative Improvement
//
// Substantially borrowed from Numerical Recipes.
// Uses LINALG package.
//
void mprove(Matrix &A, Matrix &alud, int *indx, Matrix &b, Matrix &x)
{
  Matrix r(b);

  r = A*x;                      // Calculate residual.
  r -= b;

  lubksb(alud,r,indx);          // Backsubsitute.

  x -= r;                       // Make correction.
}


void plot( Matrix &A )
{
  FILE *f = (FILE *)fopen("plot.stuff", "w");
  int N = A.q_ncols();
  int i;

  fprintf(f, "plot \"plot.tmp\" using 1\n");

  for(i = 2; i <= N; i++)
    fprintf(f, "replot \"plot.tmp\" using %d\n", i);

  fprintf(f, "pause -1");
  fclose(f);

  f = fopen("plot.tmp", "w");
  dump((char *)NULL, A, "%.3e", f);
  fclose(f);

  system("gnuplot plot.stuff");
}



void plot(Vector &v)
{
  FILE *f = (FILE *)fopen("plot.stuff", "w");
  int N = v.q_no_elems();

  if(f == NULL) return;

  fprintf(f, "plot \"plot.tmp\" using 1\n");
```

```
  fprintf(f, "pause -1\n");
  fclose(f);

  f = fopen("plot.tmp", "w");
  dump((char *)NULL, v, "%.3e", f);
  fclose(f);

  system("gnuplot plot.stuff");
}


//
// RATINT
//
#define TINY 1.0e-25
void ratint(Vector &xa, Vector &ya, double x, double &y, double &dy)
{
  int m,i,ns=1;
  double w,t,hh,h,dd;

  int n = xa.q_no_elems();

  Vector c( n);
  Vector d( n);


  hh=fabs(x-xa(1));

  for (i=1;i<=n;i++) {
    h=fabs(x-xa(i));
    if (h == 0.0) {
      y=ya(i);
      dy=0.0;
    } else if (h < hh) {
      ns=i;
      hh=h;
    }
    c(i)=ya(i);
    d(i)=ya(i)+TINY;
  }
  y=ya(ns--);

  for (m=1;m<n;m++) {
    for (i=1;i<=n-m;i++) {
      w=c(i+1)-d(i);
      h=xa(i+m)-x;
      t=(xa(i)-x)*d(i)/h;
      dd=t-c(i+1);
      if (dd == 0.0) printf("Error in routine ratint\n\n");
      dd=w/dd;
      d(i)=c(i+1)*dd;
      c(i)=t*dd;
    }
    y += (dy=(2*ns < (n-m) ? c(ns+1) : d(ns--)));
```

```
  }
}
#undef TINY



/* Matrix column tools */

// Some utility routines
// #define min(a,b) ((a)<(b)?(a):(b))

void getcol(Matrix &B, int col, Vector &x)
{
  int i,j,N = min(length(x),rows(B));

  for(i = 1; i <= N; i++)
    x(i) = B(i,col);
}

void getcol(Matrix &B, int col, Vector &x, double a)
{
  int i,j,N = min(length(x),rows(B));

  for(i = 1; i <= N; i++)
    x(i) = a*B(i,col);
}


void setcol(Vector &x, Matrix &B, int col)
{
  int i,j,N = min(length(x),rows(B));

  for(i = 1; i <= N; i++)
    B(i,col) = x(i);
}

void setcol(Vector &x, Matrix &B, int col, double a)
{
  int i,j,N = min(length(x),rows(B));

  for(i = 1; i <= N; i++)
    B(i,col) = a*x(i);
}


// Same, but += not =

void agetcol(Matrix &B, int col, Vector &x)
{
  int i,j,N = min(length(x),rows(B));

  for(i = 1; i <= N; i++)
    x(i) += B(i,col);
```

```
}

void agetcol(Matrix &B, int col, Vector &x, double a)
{
  int i,j,N = min(length(x),rows(B));

  for(i = 1; i <= N; i++)
    x(i) += a*B(i,col);
}

void asetcol(Vector &x, Matrix &B, int col)
{
  int i,j,N = min(length(x),rows(B));

  for(i = 1; i <= N; i++)
    B(i,col) += x(i);
}


void asetcol(Vector &x, Matrix &B, int col, double a)
{
  int i,j,N = min(length(x),rows(B));

  for(i = 1; i <= N; i++)
    B(i,col) += a*x(i);
}
```

Include file

```
void inv(Matrix &A);
void mprove(Matrix &A, Matrix &alud, int *indx, Matrix &b, Matrix &x);
void lusolve(Matrix &A, Matrix &B, int n = 0);
void ludcmp(Matrix &A, int *indx, double &d);
void lubksb(Matrix &A, Matrix &B, int *indx);
void dump(char *t, Vector &x, FILE *f = stdout);
void dump(char *t, Matrix &m, char *fmt, FILE *f = stdout);
void normaleqn(Matrix &A, Matrix &X, Matrix &B);
void plot( Matrix &A );
void plot(Vector &v);

/* Column/Matrix/Vector tools */
void getcol(Matrix &B, int col, Vector &x);
void getcol(Matrix &B, int col, Vector &x, double a);
void setcol(Vector &x, Matrix &B, int col);
void setcol(Vector &x, Matrix &B, int col, double a);
void agetcol(Matrix &B, int col, Vector &x);
void agetcol(Matrix &B, int col, Vector &x, double a);
```

```
void asetcol(Vector &x, Matrix &B, int col);
void asetcol(Vector &x, Matrix &B, int col, double a);


/*
void HammingWindow( Vector &x );
void BartlettWindow( Vector &x );
void HanningWindow( Vector &x );
*/


void ratint(Vector &xa, Vector &ya, double x, double &y, double &dy);



#define length(x)  ((x).q_no_elems())
#define cols(x)    ((x).q_ncols())
#define rows(x)    ((x).q_nrows())


#define min(a,b)   ((a) < (b) ? (a) : (b))
#define max(a,b)   ((a) > (b) ? (a) : (b))
```

F.6 window.cc

This module implements some signal-processing type windows for reducing spectral leakage in analysis of finite length signals.

C++ Source

```
/*
 Windowing functions.

 Implementations of the various window functions.
 Hanning, Hamming, Bartlett and Blackman

*/

#include <stdio.h>
#include <math.h>
#include "linalg/myenv.h"
#include "linalg/linalg.h"

#include "tools.h"
#include "window.h"
```

```
// Apply a Hanning Window to the Vector x
//
void HanningWindow(Vector &x)
{
  int i,N = length(x);
  double alpha = 2*M_PI/(N-1);

  for(i = 1; i <= N; i++)
    x(i) *= (.5 - .5*cos(alpha*(i-1)));
}

// Apply a hamming window to the vector x
//
void HammingWindow( Vector &x )
{
  int i,N = length(x);
  double alpha = M_PI*2.0/(N-1);

  for(i = 1; i <= N; i++)
    x(i) *= (.54 - .46*cos(alpha*(i-1)));
}

// Apply a Bartlett window to the vector x
//
void BartlettWindow( Vector &x )
{
  int i,N = length(x);
  double omega = 2.0/(N-1);

  for(i = 1; i <= N/2; i++)
    x(i) *= (i-1.0)*omega;

  for(; i <= N; i++)
    x(i) *= (1.0-(i-1.0)*omega);
}

// Apply a Blackman window to the vector x
//
void BlackmanWindow( Vector &x )
{
  int i,N = length(x);
  double alpha = M_PI*2.0/(N-1);

  for(i = 1; i <= N; i++)
    x(i) *= (.42 - .5*cos(alpha*(i-1)) + .08*cos(2*alpha*(i-1)));
}
```

Include file

```
/*
 Windowing functions.
*/

void BlackmanWindow( Vector &x );
void HanningWindow(  Vector &x );
void HammingWindow(  Vector &x );
void BartlettWindow( Vector &x );
```

Appendix G

Data file tools

This appendix contains several utilities for manipulating data files. Application of the utilities is shown in the shell script below, which was used to translate the raw Tektronix .WFM format files containing the induction motor transient measurements presented in Chapter 5.

G.1 BASH Script to translate Tektronix .wfm files to .dqo files

```
convert tek00005.wfm 50.0 .07 > t1.csv
convert tek00004.wfm 50.0 .07 > t2.csv
convert tek00003.wfm 50.0 .07 > t3.csv

catl t1.csv t2.csv > out
catl out t3.csv > out1
clip out1 410 > out
taxis out 4e-5 -.056517 > si.dat

convert tek00006.wfm -100.0 .07 > t1.csv
clip t1.csv 410 > out
taxis out 4e-5 -.056517 > sr.dat

convert tek00007.wfm -100.0 .07 > t1.csv
convert tek00008.wfm -100.0 .07 > t2.csv
```

```
convert tek00009.wfm -100.0 .07 > t3.csv

catl t1.csv t2.csv > out
catl out t3.csv > out1
taxis out1 4e-5 -.056517 > sv.dat

frame sv.dat > out1
cols out1 1 3 2 4 > sv.dqo

frame si.dat > out1
cols out1 1 3 2 4 > si.dqo
```

G.2 File manipulation utilities

The file manipulation utilities include `clip`, `catl`, `convert`, `frame` and `taxis`. `clip` removes records from the beginning of a data file and is useful for aligning files according to a synchronizing signal. `catl` concatonates files, like `cat`, except that the files are joined line by line. `convert` translate binary Tektronix format .WFM files to ASCII. `convert` scales the .WFM file according to the units per division and zeros the output according to values measured in the pretrigger interval. `frame` converts between data files containing either lab or *dq* frame data. Finally, `taxis` is a program that inserts a "time-axis" in a data file.

```
/*
   clip.c

    Steven Shaw.

   This program is useful for syncing datasets.

   It reads a data file in the format [data1] .. [datan]
   and puts on the standard output the lines [data1] .. [datan]
   only for n > N

   USAGE:

   CLIP [FILE] [N] > OUT
*/
```

```
#include <stdio.h>
#include <string.h>

void main(int argc, char *argv[])
{
  static char line1[4096];
  FILE *f1 = fopen(argv[1],"r");
  long n,n0;

  if(f1 == NULL) {
    printf("clip: could not open file\n");
    return;
  }

  n0 = strtol(argv[2],NULL,0);
  n  = 0;

  while(!feof(f1)) {
    fgets(line1,4096,f1);

    if(!feof(f1) && n > n0)
printf("%s", line1);

    n++;
  }

  fclose(f1);
}
```

```
/*
   catl.c


 Steven R. Shaw.


   concatonate lines

   Usage:

   catl [file1] [file2]

   catl concatonates [file1] and [file2] line by line on
   the standard output.
*/

#include <stdio.h>
#include <string.h>
```

```
void main(int argc, char *argv[])
{
  static char line1[4096],line2[4096];
  FILE *f1 = fopen(argv[1],"r");
  FILE *f2 = fopen(argv[2],"r");

  if(f1 == NULL || f2 == NULL) {
    printf("catl: could not open file\n");
    return;
  }

  while(!feof(f1) && !feof(f2)) {
    fgets(line1,4096,f1);
    fgets(line2,4096,f2);

    if(!feof(f1) && !feof(f2)) {
      strtok(line1,"\n");
      strtok(line2,"\n");
      printf("%s\t%s\n",line1,line2);
    }
  }

  fclose(f1);
  fclose(f2);
}
```

```
/*
   Program to convert .WFM files to .CSV files.

   Steven R. Shaw.
*/

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

/* Prototypes. */
void cnvrt(FILE *f,double upd,double pre);
#define MAXN (32768)

/* Static data. */
static double data[MAXN];

/* main program */
void main(int argc, char *argv[])
{
  FILE *f;
  double upd,pre;

  if( argc != 4 ) {
```

```
    printf("Usage: \n");
    printf("convert [file] [units_per_division] [pre_trigger] \n");
    return;
  }

  /* Open the input file. */
  f = fopen(argv[1], "rb");
  if(f == NULL) {
    printf("Error opening %s \n", argv[1]);
    return;
  }

  /* Units per division */
  upd = strtod(argv[2],NULL);

  /* Pretrigger -> used to subtract DC bias... */
  pre = strtod(argv[3],NULL);

/*
  printf("Converting %s with pretrigger of %lf %% \n", argv[1], pre*100);
  printf("Scale factor = %lf\n", upd);
*/
  cnvrt(f,upd,pre);

  fclose(f);
}

/* This routine converts the TEK .WFM file to ASCII */
void cnvrt(FILE *f,double upd,double pre)
{
  char  title[64];
  long  stuff[64];
  long i,k,N;
  short w;
  double mean;

  fread(title, sizeof(char),  16, f);
  fread(stuff,sizeof(double), 16, f);

  for(i = 1; i <= MAXN && !feof(f); i++) {
    fread(&w, sizeof(short), 1, f);
    data[i] = (double)w;
  }

  /* Compute the pretrigger average, set to zero... */
  N = i - 20;
  k = N*pre;
  for(i = 1, mean = 0; i <= k; i++)
    mean += data[i];
  mean /= (double)k;

  /* Subtract, scale and print...                    */
  upd *= 5.0 / 32768;
  for(i = 1; i <= N; i++) {
```

```
    data[i] = (data[i]-mean)*upd;
    printf("%.5e\n",data[i]);
  }
}
```

```
/*
   frame.c

   Steven Shaw.
   Convert ABC variables to DQ0 variables and vice versa.
*/


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

/* Prototypes.       */
void abcdq(FILE *f);
void dqabc(FILE *f);

void main(int argc, char *argv[])
{
  FILE *f;
  char *filen = argv[1];         /* Default */
  int todq = 1;
  int i;

  /* Print usage information */
  if(argc > 3 || argc < 2) {
    printf("Usage:\n");
    printf("frame [-abc] [-rev] [file]\n");
    printf("-abc causes frame to convert to the abc frame on stdout.\n");
    printf("Atherwise, conversion to dq0 frame is performed.\n");
    return;
  }

  /* Parse command line.     */
  for(i = 1; i < argc; i++) {
    if( *argv[i] != '-')
      filen = argv[i];
    else {
      todq = strcmp(argv[i],"-abc");
        }
  }

  /* Open up input file.     */
  f = fopen(filen, "r");
```

```
  if(f == NULL) {
    printf("error opening file %s\n", filen);
    return;
  }

  /* Convert, one way or
     the other               */
  if(todq != 0)
    abcdq(f);
  else
    dqabc(f);

  fclose(f);
}

#define OMEGA (M_PI*2.0*60.0)
#define PHI    (M_PI*2.0/3.0)

/*
   go to dq0 frame
*/
void abcdq(FILE *f )
{
  double t,a,b,c,d,q,o;
  double beta;

  while(!feof(f)) {
    fscanf(f,"%lf  %lf  %lf  %lf\n", &t, &a, &b, &c);

    /* synchronous frame    */
    beta = OMEGA*t;

    /* parks transform      */
    d = (2.0/3.0)*(a*cos(beta)+b*cos(beta-PHI)+c*cos(beta+PHI));
    q = (2.0/3.0)*(a*sin(beta)+b*sin(beta-PHI)+c*sin(beta+PHI));
    o = (1.0/3.0)*(a+b+c);

    /* to stdout            */
    printf("%.4e %.4e %.4e %.4e\n", t, d, q, o);
  }
}


/*
   go to abc frame
*/
void dqabc(FILE *f)
{
  double t,a,b,c,d,q,o;
  double beta;

  while(!feof(f)) {
    fscanf(f,"%lf  %lf  %lf  %lf\n", &t, &d, &q, &o);
```

```
    /* synchronous frame    */
    beta = OMEGA*t;

    /* parks transform      */
    a = d*cos(beta)     + q*sin(beta)     + o;
    b = d*cos(beta-PHI) + q*sin(beta-PHI) + o;
    c = d*cos(beta+PHI) + q*sin(beta+PHI) + o;

    /* to stdout            */
    printf("%.4e %.4e %.4e %.4e\n", t, a, b, c);
  }
}
```

```
/*
   taxis.c

   Steven R. Shaw.

   Attach a time axis to a data file, put on stdout.

   usage:

   taxis [file] [T] [T0]
*/

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

void main(int argc, char *argv[])
{
  double T,T0,t;
  long   i;
  FILE *f;
  static char lines[4096];

  if( argc != 4 ) {
    printf("Usage:\n");
    printf("taxis [file] [t] [t0]\n");
    printf("Creates a ´time axis´ on the stdout.\n");
    printf("N points spaced by T starting at T0.\n");
    return;
  }

  /* Open file */

  f = (FILE *)fopen(argv[1], "r");
  if( f == NULL ) {
    printf("error opening %s\n", argv[1]);
    return;
```

```
  }

  /* Get parameters of time axis */
  T  = strtod(argv[2],NULL);
  T0 = strtod(argv[3],NULL);

  i = 0;

  while(!feof(f)) {
    /* Get a line         */
    fgets(lines,4096,f);

    t = i*T + T0;
    i++;

    if(!feof(f))
      printf("%.5e \t%s", t, lines);
  }

  fclose(f);
}
```

Bibliography

[1] W. C. Beattie and S. R. Matthews. Impedance measurement on distribution networks. In *Proceedings of the 29th Universities Power Engineering Conference*, pages 117–120, September 1994.

[2] Philip R. Bevington and D. Keith Robinson. *Data Reduction and Error Analysis for the Physical Sciences.* McGraw-Hill, second edition, 1992.

[3] Richard M. Bozorth. *Ferromagnetism.* D. Van Nostrand Company, Princeton, New Jersey, 1959.

[4] K. R. Cho, J. H. Lang, and S. D. Umans. Detection of broken rotor bars in induction motors using parameter and state estimation. In *Proceedings of IEEE IAS Annual Meeting*, December 1989.

[5] Kyong Rae Cho. Detection of broken rotor bars in induction motors using parameter and state estimation. Master's thesis, MIT, June 1989.

[6] Claude Cohen-Tannoudji, Bernard Diu, and Frank Laloë. *Quantum Mechanics*, volume 1. John Wiley and Sons, 1977. An excellent discussion of the properties of the commutator appears in the Appendix.

[7] Alfio Consoli, Luigi Fortuna, and Antonio Gallo. Induction motor identification by a microcomputer-based structure. *IEEE Transactions on Industrial Electronics*, 1985.

[8] Gene H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins, second edition, 1989.

[9] Glenn V. Gordon and Montgomery T. Shaw. *Computer Programs for Rheologists.* Hanser, 1994.

[10] Rolf Johansson. *System Modeling and Identification.* Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[11] Paul C. Krause. *Analysis of Electric Machinery.* McGraw-Hill Book Company, 1986.

[12] S. B. Leeb. *A Conjoint Pattern Recognition Approach to Nonintrusive Load Monitoring.* Phd, MIT, Department of Electrical Engineering and Computer Science, February 1993.

[13] S. B. Leeb and J. L. Kirtley. A multiscale transient event detector for nonintrusive load monitoring. In *Proceedings of the IEEE*, November 1993.

[14] S. B. Leeb and J. L. Kirtley. A transient event detector for nonintrusive load monitoring. Technical report, U.S. Patent Number 5,483,153, Issued January 1996.

[15] Rob Lepard. Power quality prediction based on determination of supply impedance. Master's thesis, MIT, 1996.

[16] Kazuaki Minami. Model-based speed and parameter tracking for induction machines. Master's thesis, MIT, May 1989.

[17] Zbigniew H. Nitecki and Martin M. Guterman. *Differential Equations with Linear Algebra.* CBS College Publishing, 1986.

[18] A. V. Oppenheim and R. W. Schaefer. *Discrete-Time Signal Processing.* Prentice Hall, 1989.

[19] W. H. Press, S. A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992.

[20] C. R. Rao and Helge Toutenburg. *Linear Models* Least Squares and Alternatives. Springer, 1995.

[21] Gian Carlo Rota. *Probability.* Unpublished version of a book to appear, used in the MIT class 18.313., 1993.

[22] Seth Robert Sanders. State estimation in induction machines. Master's thesis, MIT, June 1985.

[23] Steven R. Shaw, Robert F. Lepard, and Steven B. Leeb. Desire: A power quality prediction system. In *Proceedings of the North American Power Symposium*, September 1996.

[24] William McC. Siebert. *Circuits, Signals, and Systems.* The MIT Press, 1986.

[25] G. R. Slemon. *Magnetoelectric Devices: Transducers, Transformers and Machines.* John-Wiley and Sons, 1966.

[26] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer Verlag, second edition, 1992.

[27] Gilbert Strang. *Linear Algebra and its Applications.* Harcourt Brace Jovanovich, third edition, 1988.

[28] K. J. Åström. Maximum likelihood and prediction error methods. *Automatica*, 16:551–574, 1980.

[29] Miguel Vélez-Reyes. Speed and parameter estimation for induction machines. Master's thesis, MIT, May 1988.

[30] Miguel Vélez-Reyes. *Decomposed Algorithms for Parameter Estimation.* PhD thesis, MIT, September 1992.

[31] Miguel Vélez-Reyes Kazuaki Minami George C. Verghese. Recursive speed and parameter estimation for induction machines. In *Proceedings of IEEE Industry Applications Society Annual Meeting*, 1989.

[32] Mark Steven Welsh. Detection of broken rotor bars in induction motors using stator current measurements. Master's thesis, MIT, May 1988.