COMPUTATIONAL TOOLS FOR MODELLING MOLECULAR NETWORKS IN BIOLOGICAL SYSTEMS

by

Jason W. Zwolak

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

APPROVED:

Layne T. Watson, co-chair	John J. Tyson, co-chair
Lenwood S. Heath	Clifford A. Shaffer

Jill Sible

December 15, 2004
Blacksburg, Virginia

Key words: Cell Cycle Models, Ordinary Differential Equations, Parameter Estimation, Steady State Solutions, Global Optimization.

COMPUTATIONAL TOOLS FOR MODELLING MOLECULAR NETWORKS IN BIOLOGICAL SYSTEMS

by

Jason W. Zwolak

(ABSTRACT)

Theoretical molecular biologists try to understand the workings of cells through mathematics. Some theoreticians use systems of ordinary differential equations (ODEs) as the basis for mathematical modelling of molecular networks. This thesis develops algorithms for estimating molecular reaction rate constants within those mathematical models by fitting the models to experimental data. An additional step is taken to fit non-timecourse experimental data (e.g., transformations must be performed on the ODE solutions before the experimental and simulation data are similar, and therefore, comparable). VTDIRECT is used to perform (a deterministic direct search) global estimation and ODRPACK is used to perform (a trust region Levenberg-Marquardt based) local estimation of rate constants. One such transformation performed on the ODE solutions determines the value of the steady state of the ODE solutions. A new algorithm was developed that finds all steady state solutions of the ODE system given that the system has a special structure (e.g., the right hand sides of the ODEs are rational functions). Also, since the rate constants in the models cannot be negative and may have other restrictions on the values, ODRPACK was modified to address this problem of bound constraints. The new Fortran 95 version of ODRPACK is named ODRPACK95.

ACKNOWLEDGMENTS

I can never give the volume of thanks due for those who supported me in this work. Thank you Dr. Watson for the many hours of your time you dedicated to helping and teaching me. Thank you Dr. Tyson for providing a wonderful working environment. Thank you both for being role models. Thank you Dr. Heath, Dr. Shaffer, and Dr. Sible for serving on my committee.

Thank you Kathy Chen for being a special friend. Thanks to all my friends in the dancing and music communities for supplying many hours of fun. Thanks to my mother, father, and brother for being a wonderful family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: Introduction

A recent trend in molecular biology has led theoretical biologists to a greater dependence on computers to do mathematical calculations and simulations. These theoretical biologists create mathematical models of molecular networks and use the models to achieve greater understanding or to make predictions. The work described in this thesis was done for those theoretical biologists using ordinary differential equations (ODEs) to build their models.

Theoreticians derive the models from wiring diagrams of networks of proteins and/or genes (but, proteins for the models in this thesis) such as the simple hypothetical example given in Fig. 1.1. In this example the fictitious protein X is transformed into the fictitious protein Y, and conversely, protein Y is transformed into protein X. (In reality these reactions would not likely occur without the involvement of other proteins, but this will do for an example.) The theoretician must represent the rate of change of X to Y, and the reverse rate of change, Y to X. Often, mass action kinetics are used such that the reaction X transforms into Y occurs at the rate k_1X and the reverse occurs at the rate k_2Y. A system of rate equations (ODEs) representing the time rate of change of X and Y is written as

$$\frac{dX}{dt} = -k_1X + k_2Y, \tag{1.1}$$

$$\frac{dY}{dt} = k_1X - k_2Y, \tag{1.2}$$

where the constants k_1 and k_2 are to be determined and X and Y appear as variables. Since X and Y are neither created (synthesized) nor destroyed (degraded), there is the conservation relation $X + Y = T$, where T is some constant representing the fact that X and Y are not synthesized or degraded but can merely transform between each other (e.g., every loss of a unit of X results in a gain of a unit of Y and vice versa). The theoretician must supply the rate constants and initial conditions before the equations can be solved. The experiments being modeled usually supply relative or absolute initial concentrations of the proteins. These initial concentrations can also be used to determine the constants in the conservation relations. Say the initial concentration of X is 1 and the initial concentration of Y is 0. Now Eqs. (1.1)–(1.2) can be rewritten as

$$X(0) = 1, \tag{1.3}$$

$$Y(0) = 0, \tag{1.4}$$

$$T = X + Y = 1, \tag{1.5}$$

$$\frac{dX}{dt} = -k_1X + k_2Y = k_2 - (k_1 + k_2)X, \tag{1.6}$$

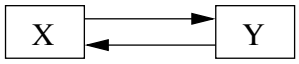

Fig. 1.1. A simple example wiring diagram of the fictitious proteins X and Y. X transforms into Y according to the arrow pointing right and Y transforms into X according to the arrow pointing left.

where Y can be written as $Y = 1 - X$ (as chosen here, but the alternative $X = 1 - Y$ is also possible), and k_1 and k_2 are still unknown rate constants. Theoreticians can, however, estimate the rate constants by experimental data by adjusting the rate constants until the model matches the experimental data. This process is known as parameter estimation and is an iterative process of adjusting rate constants (parameters) and comparing simulations and experiments with the goal of matching simulations to experiments as closely as possible. This is precisely the problem addressed in this thesis and the basis for all work described here.

There are existing tools that can be used to estimate parameters or to aid parameter estimation. One such tool theoreticians use is XPP/XPPAUT (Ermentrout, 2002). XPP cannot estimate parameters in ODE models, but it can aid theoreticians in their search for rate constants. XPP provides a platform and reasonable interface for integrating (simulating) systems of ODEs (models). The theoretician then has the task of comparing the simulation output (timecourses of protein concentrations) to experimental data and manually adjusting rate constants until simulations match experiments. XPP also provides sophisticated bifurcation analysis that helps theoreticians understand the model better and therefore better guess the rate constants. WinPP (a Microsoft Windows version of XPP) was used in this way to develop the models described in Chen et al. (2000) and Chen et al. (2004).

Computers can do more than simulate models, while leaving parameter estimation to theoreticians. Gepasi (Mendes, 1993) can simulate models and perform parameter estimation using a number of different algorithms. Gepasi is, however, limited in the types of experimental data it can handle; Gepasi can only handle timecourse data (values of protein concentrations (ODE variables) with respect to time). In many cases, experimental data are not timecourses of protein concentrations [(Chen et al., 2000), (Chen et al., 2004), (Marlovits et al., 1998), (Novak and Tyson, 1993)]. The experimental data, for example, can be a time lag for a protein to reach 50% active, or a threshold for a protein activity level that, when crossed, activates a second protein.

There is a large effort to create a community of software developers and theoretical biologists called BioSPICE (Kumar and Feidler, 2003). This community aspires to create (and has created already) many tools to solve many problems such as the parameter estimation problem described here. The BioSPICE community has tools to build and simulate

models, but they do not yet have tools to perform parameter estimation. Much of the work described in this thesis was funded by the DARPA BioSPICE project and is included in the code base for BioSPICE as part of JigCell (Allen et al., 2003). Although JigCell includes much of the code used for this work, there still remains a lot of work before the parameter estimation tools described here can be used within the BioSPICE and JigCell frameworks.

Sometimes estimating parameters involves finding steady states in the model to compare to experiments. These solutions can also be used in bifurcation analysis. A steady state solution consists of values for the variables that make the right hand sides evaluate to 0. Since the right hand sides are the rates of change of the variables, this means the variables will not change their values (e.g., after the variables reach a steady state they will remain at that steady state forever). A steady state solution for the model in Eq. (1.3)–(1.6) is

$$0 = k_2 - (k_1 + k_2)X, \tag{1.7}$$

$$(k_1 + k_2)X = k_2, \tag{1.8}$$

$$X = \frac{k_2}{k_1 + k_2}. \tag{1.9}$$

$$Y = 1 - \frac{k_2}{k_1 + k_2}. \tag{1.10}$$

When the steady state values are substituted into Eq. (1.6), the rate of change of X, and therefore Y, becomes 0. Finding steady state solutions is important because sometimes the experimental data is a steady state. Therefore, the steady states of the model must be determined in order to compare the model to the experiment. This thesis describes an algorithm that finds all steady state solutions to chemical kinetic models (models based on ODEs using mass action and Michaelis-Menten kinetics).

Another desirable task is to limit rate constants to positive numbers while estimating parameters. The package VTDIRECT, used here for global searches of parameter space, already limits the rate constants to a user defined range of valid values (this limitation is called bound constraints). The package ODRPACK, used here for local searches of parameter space, does not support bound constraints on parameters. This thesis describes a modified version of ODRPACK, called ODRPACK95, which supports bound constraints and is written the Fortran 95 language (instead of FORTRAN 77 as was the original ODRPACK).

The chapters of this thesis are published, in-press, or submitted journal articles. The journal articles have been left intact. This means that each chapter can stand alone and some information is duplicated. This also means that the equivalent of a literature review is spread between this introduction and the introduction of each chapter. Chapter 2 describes parameter estimation performed on a model of the frog egg. The parameter estimation uses ODRPACK for local optimization and LSODAR to integrate the system of ODEs. The estimated parameters are similar to those of Marlovits et al. (1998). Chapter 3 describes a global search of parameter space on the same frog egg model. The global search uses VTDIRECT and refines the parameter estimates with ODRPACK. The estimated parameters are different from those in Marlovits et al. (1998) and led to the discovery of a simplified model. Chapter 4 describes the algorithm to find all steady states of chemical kinetic models. Chapter 5 describes ODRPACK95.

CHAPTER 2: Locally Optimized Parameters for a Frog Egg Model

2.1. INTRODUCTION

The ultimate goal of molecular cell biology is to understand how the information stored in the genome is read out and put into action as the physiological behavior of a living cell. At one end, genes provide the "code" to synthesize polypeptide chains, and at the other end, networks of interacting proteins govern how a cell moves, feeds, responds, and reproduces. The "last step" of computational molecular biology is to derive the physiological properties of a cell from the wiring diagrams of its underlying molecular regulatory circuits (Tyson et al., 2001). The approach we take is deterministic modeling of the regulatory system by ordinary differential equations (ODEs). This approach provides direct, quantitative connections between mechanistic details and observable cellular behavior, but it is constrained by the difficulty of assigning numerical values to the many kinetic parameters (rate constants and binding constants) that appear in the governing ODEs. In this chapter, we show how to assign "optimal" values to the kinetic parameters that appear in the governing differential equations.

Common practice among modelers is to "twiddle" the parameters, in order to bring numerical solutions of the ODEs in reasonable agreement with a set of experimental observations. If so, the model is confirmed, and "guesstimates" of the kinetic parameters are established. If no amount of twiddling can bring the model into agreement with the data, then the modeler must consider changing the wiring diagram, which changes the governing ODEs and starts the parameter twiddling process anew (von Dassow et al., 2000). For the most part, the parameter estimation phase is still done by hand, even for quite complex models (Chen et al., 2000), (Teusink et al., 2000), (Hynne et al., 2001). The data to be fit are often diverse types and rarely straightforward measurements of dynamical variables as a function of time. Usually, the observations are not repeated enough times for reliable statistical estimates to be made. The observations on which a model is based are often highly variable in quality and reliability. For these reasons, the modeler must often make many mental "tradeoffs" in evaluating the suitability of a model plus parameter values in fitting a set of experiments. While computational biologists were gaining experience with large ODE models of complex regulatory systems, it was reasonable to survey parameter values manually, but eventually, hand crafting of models must be replaced—or supplemented—by automatic parameter estimation (Moles et al., 2003).

We use classical optimization methods to estimate rate constants in a model of the nuclear replication-division cycle in frog egg extracts. The model is simple enough to be readily understood yet complex enough to illustrate the many challenges of fitting molecular mechanisms of cell behavior to real biochemical and physiological data. The algorithms we employ are sufficiently powerful and general to attack more challenging and important problems of cell cycle modeling in the future.

We start by testing our tools against simple models of the cell cycle. The cell cycle is the sequence of events by which a cell replicates all its components and divides them between two daughter cells, so that each daughter has the information and machinery necessary to repeat the process (Murray and Hunt, 1993). The most important component is the genome, which must be accurately replicated during interphase (each chromosome becoming two identical sister chromatids) and carefully partitioned during mitosis (sister chromatids moving to opposite poles of the mitotic spindle). These basic events, DNA synthesis and mitosis, are triggered in alternating fashion by a complex network of molecular interactions involving cyclin-dependent protein kinases (Cdks). The first 10–12 mitotic cycles of a fertilized egg provide a favorite experimental system for studying this biochemical network because the cycles of DNA synthesis (S phase) and mitosis (M phase) proceed rapidly, without the intervening gaps (G1 and G2) characteristic of somatic cell cycles. Furthermore, for eggs of the African clawed frog, *Xenopus laevis*, protocols have been developed for making cell-free extracts that carry out DNA synthesis and nuclear division in a colloidal suspension unhindered by plasma membranes.

Twenty-five years of careful biochemical studies have revealed a relatively simple control system for the nuclear division cycle in frog egg extracts (Fig. 2.1), which is only a subset of the reaction network in intact cells. Novak and Tyson (1993) published a thorough theoretical study of the mechanism, comparing it mostly to qualitative observations of mitotic cycles in egg extracts. At the time, Novak and Tyson postulated a set of rate constants that gave a reasonable, semi-quantitative fit of the model to experimental data on the activation of M-phase promoting factor (MPF, a dimer of Cdk1 and cyclin B). Marlovits et al. (1998) re-evaluated the model in light of more quantitative studies on MPF activation in frog egg extracts. By back-of-an-envelope calculations on the available kinetic data, they proposed a refined parameter set, not too much different from the original guesses of Novak and Tyson.

In this chapter, we present the frog-egg model and the experimental data in full quantitative detail, and take on the challenge of finding the parameter set that optimally fits the model to the data. To express the goodness of fit of the model to the data, we use an objective function based on "orthogonal distance" between model predictions and experimental observations. Orthogonal distance allows us to weight independently the likely errors in all measured quantities. To minimize the weighted sum of orthogonal distances, we could use many different methods—local or global, deterministic or nondeterministic. We use an algorithm (Levenberg-Marquardt) for local, gradient-based optimization—the best currently known deterministic local optimization algorithm for nonlinear least-squares problems. Using Marlovits et al. (1998) for an initial parameter set, we show that the Levenberg-Marquardt algorithm converges quickly to a (locally) optimal parameter set not too far away, and the regression error is significantly improved. We also characterize the basin of attraction near this optimal solution.

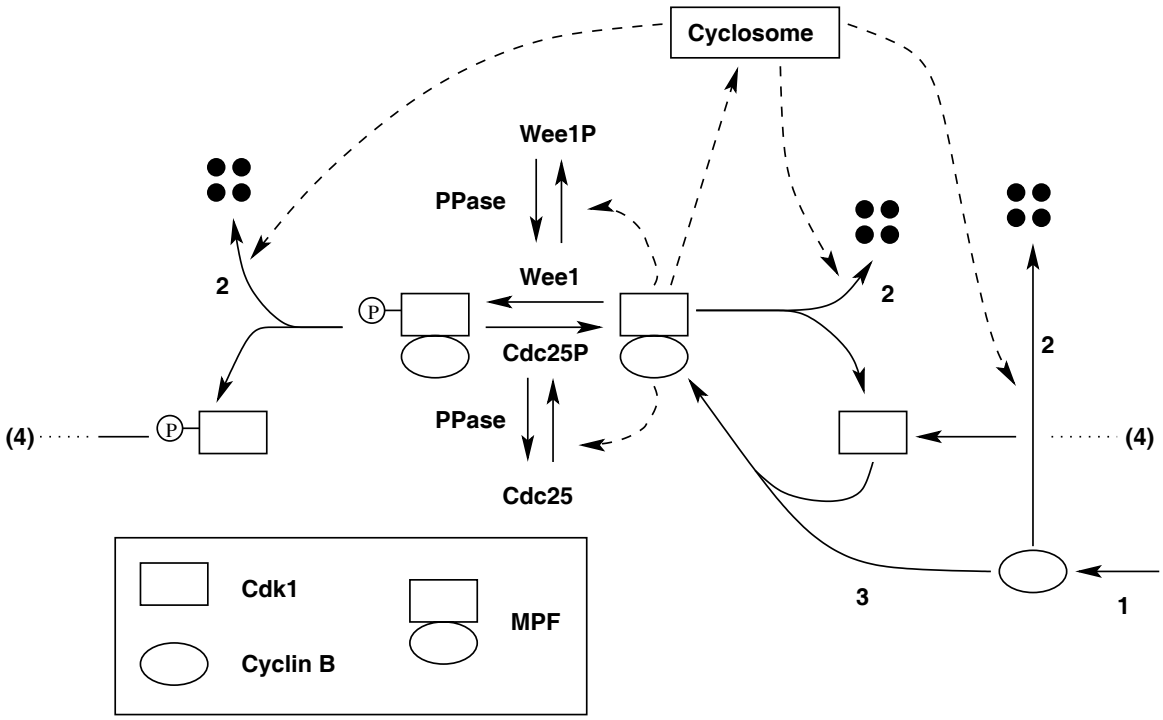

Fig. 2.1. The biochemical control system for MPF activation in frog egg extracts. MPF = mitosis-promoting factor = dimer of a cyclin-dependent kinase, Cdk1 (rectangle), and a B-type cyclin (oval). Rising MPF activity drives an extract through DNA synthesis, nuclear envelope breakdown, chromosome condensation, and alignment of replicated chromosomes on the mitotic spindle. Falling MPF activity allows for sister chromatid separation, nuclear reassembly, and licensing of DNA for another round of replication. Cycles of MPF activation and inactivation are driven by phases of cyclin synthesis and degradation, and by phases of Cdk1 phosphorylation (by Wee1 and Myt1, together shown as Wee1) and dephosphorylation (by Cdc25). (Note that there are two phosphorylation sites: Thr161 and Tyr15. The Thr161 phosphorylation site is ignored in this model.) The newly fertilized egg has a large supply of Cdk1, the enzymes Wee1 and Cdc25, and proteasomes (the protein complexes that promote cyclin B degradation in the steps labeled 2). The only missing component is cyclin B. As cyclin is synthesized (step 1), it combines rapidly (step 3) with Cdk1 monomers to form active MPF dimers. The active dimers, however, are rapidly phosphorylated by Wee1 to a less active form. During this phase of the cycle, cyclins are relatively stable because the proteasome is inactive. For the extract to enter mitosis, the inhibitory phosphate groups must be removed from Cdk1 by Cdc25. Activation of MPF is an autocatalytic process because active MPF activates Cdc25 and inhibits Wee1 (PPase is a phosphatase that opposes MPF in these reactions). As MPF activity rises, nuclei are driven into mitosis and proteasomes are activated. The proteasomes promote rapid cyclin degradation, which destroys MPF activity and allows nuclei to finish mitosis and prepare for a new round of DNA replication.

2.2. MODEL

The reaction network in Fig. 2.1 can be converted into a set of ODEs for the time rates of change of concentrations of all the proteins in the mechanism. Before doing so, we make a number of simplifications to the mechanism. All the experiments analyzed in this chapter are carried out in extracts supplemented with cycloheximide, an inhibitor of protein synthesis. Hence the extract does not synthesize cyclin from its own store of cyclin mRNA, so we set $k_1 = 0$ and ignore this reaction. The experimenter adds to the extract a known amount of exogenously produced cyclin. In all cases, the added cyclin protein has been genetically engineered to be resistant to proteasome-mediated degradation. Hence $k_2 = 0$, and we can ignore all the degradation steps in the mechanism. Because cyclin is neither synthesized nor destroyed, the total concentration of cyclin protein in the extract is constant:

$$[\text{monomeric cyclin}] + [\text{MPF}] + [\text{preMPF}] = [\text{total cyclin}] = \text{constant},$$

where MPF is the active Cdk1:cyclin dimer and preMPF refers to the phosphorylated (low activity) form of the dimer. Next, we make the assumptions (well supported by experiments) that (1) the binding of cyclin monomers and Cdk1 monomers is very fast (k_3 is large) and (2) [total Cdk1] > [total endogenous cyclin]. Hence [monomeric cyclin] << [total cyclin], and the conservation condition on cyclin subunits becomes

$$[\text{MPF}] + [\text{preMPF}] = [\text{total cyclin}].$$

With these simplifications and assumptions, the mechanism in Fig. 2.1 can be described by three ODEs

$$\frac{dM}{dt} = \big(v_d'(1-D) + v_d''D\big)(C_T - M) - \big(v_w'(1-W) + v_w''W\big)M, \tag{2.1}$$

$$\frac{dD}{dt} = v_d\left(\frac{M(1-D)}{K_{md} + (1-D)} - \frac{\rho_d D}{K_{mdr} + D}\right), \tag{2.2}$$

$$\frac{dW}{dt} = v_w\left(-\frac{MW}{K_{mw} + W} + \frac{\rho_w(1-W)}{K_{mwr} + (1-W)}\right), \tag{2.3}$$

where

$$\begin{aligned} M &= [\text{MPF}]/[\text{total Cdk1}], \\ D &= [\text{Cdc25P}]/[\text{total Cdc25}], \\ W &= [\text{Wee1}]/[\text{total Wee1}], \\ C_T &= [\text{total cyclin}]/[\text{total Cdk1}]. \end{aligned}$$

[MPF], [Cdc25P] (phosphorylated [Cdc25]), and [Wee1] represent the concentration of the active forms of MPF, Cdc25, and Wee1, respectively, and [total cyclin], [total Cdc25], and [total Wee1] represent the total concentrations (including both active and inactive forms) of MPF, Cdc25, and Wee1. In Eqs. (2.1)–(2.3), time is expressed in minutes, and all

concentrations are dimensionless numbers, having been scaled relative to some appropriate reference concentration. In frog egg extracts, [Cdk1] is typically close to 100 nM [(Solomon et al., 1990), (Wei et al., 2002)]. Total concentrations for Cdc25, Wee1 and PPase are unknown, so we set each to 1 AU ("arbitrary unit"). All vs are pseudo-first-order rate constants (units = time^{-1}). All Ks are dimensionless Michaelis constants, i.e., ratios of true Michaelis constants ($\hat{K}$, units = concentration) to reference concentrations. The ρs are dimensionless numbers expressing the activity of the phosphatase (PPase, in Fig. 2.1) relative to MPF. The rate constants refer to the following reactions:

v'_d — dephosphorylation of preMPF by Cdc25 in its less-active form,
v''_d — dephosphorylation of preMPF by Cdc25 in its active form (phosphorylated),
v'_w — phosphorylation of MPF by Wee1 in its less-active form (phosphorylated),
v''_w — phosphorylation of MPF by Wee1 in its active form,
v_d — phosphorylation of Cdc25 by MPF,
$v_d\rho_d$ — dephosphorylation of Cdc25 by PPase,
v_w — phosphorylation of Wee1 by MPF,
$v_w\rho_w$ — dephosphorylation of Wee1 by PPase,

where PPase is generic name assigned to phosphatase(s) of Cdc25 and Wee1 (there is no claim here on the real names of this phosphatase(s)).

Table 2.1 indicates how the parameters in Eqs. (2.1)–(2.3), the vs and Ks without hats, are related to the fundamental kinetic parameters, the $\hat{v}$s and $\hat{K}$s, and a "dilution factor" μ. This dilution factor must be included in our calculations, because in some experiments a buffered solution of proteins is added to the frog egg extract, increasing the volume of the extract and thereby diluting all the endogenous proteins in the extract. Dilution changes the values of the scaled parameters from one experiment to the next, and must be taken into account when trying to fit the model to real data. The dilution factor, constrained to be greater than 1, allows more variance in the parameters to fit the data. For the parameter estimation done in this thesis the dilution factor remained close to 1, even though no upper bound was placed on it.

2.3. EXPERIMENTS

Our goal is to obtain the "best" estimates of the rate constants in Table 2.1 from the experimental data presented in Fig. 2.2. The figure also presents the best-fitting curves derived from ODEs (2.1)–(2.3), and the optimal parameter values (Table 2.2). In Table 2.3 we describe how each experiment is to be simulated by the model equations (2.1)–(2.3).

Experiments A–H in Fig. 2.2 are straightforward measurements of enzyme activity as functions of time. The data in Figs. 2.2I and 2.2J are more indirect and require further explanation.

Table 2.1. Estimated parameter values (Marlovits et al., 1998). μ = dilution factor (see text).

Unscaled Parameters		Scaled Parameters	
Name	**Value**	**Name**	**Value**
$\hat{v}'_d$	0.017 $\mathrm{AU_d^{-1}min^{-1}}$	$v'_d = \mu\hat{v}'_d[\mathrm{totalCdc25}]$	0.017 $\mathrm{min^{-1}}$
$\hat{v}''_d$	0.17 $\mathrm{AU_d^{-1}min^{-1}}$	$v''_d = \mu\hat{v}''_d[\mathrm{totalCdc25}]$	0.17 $\mathrm{min^{-1}}$
$\hat{v}'_w$	0.01 $\mathrm{AU_w^{-1}min^{-1}}$	$v'_w = \mu\hat{v}'_w[\mathrm{totalWee1}]$	0.01 $\mathrm{min^{-1}}$
$\hat{v}''_w$	1.0 $\mathrm{AU_w^{-1}min^{-1}}$	$v''_w = \mu\hat{v}''_w[\mathrm{totalWee1}]$	1.0 $\mathrm{min^{-1}}$
$\hat{v}_d$	0.02 $\mathrm{AU_d\ nM^{-1}min^{-1}}$	$v_d = \hat{v}_d[\mathrm{totalCdk1}]/[\mathrm{totalCdc25}]$	2.0 $\mathrm{min^{-1}}$
$\hat{v}_w$	0.02 $\mathrm{AU_w\ nM^{-1}min^{-1}}$	$v_w = \hat{v}_w[\mathrm{totalCdk1}]/[\mathrm{totalWee1}]$	2.0 $\mathrm{min^{-1}}$
$\hat{v}_{dr}$	0.1 $\mathrm{AU_d\ AU_p^{-1}min^{-1}}$	$\rho_d = (\hat{v}_{dr}/\hat{v}_d)[\mathrm{PPase}]/[\mathrm{totalCdk1}]$	0.05
$\hat{v}_{wr}$	0.1 $\mathrm{AU_w\ AU_p^{-1}min^{-1}}$	$\rho_w = (\hat{v}_{wr}/\hat{v}_w)[\mathrm{PPase}]/[\mathrm{totalCdk1}]$	0.05
$\hat{K}_{md}$	0.1 $\mathrm{AU_d}$	$K_{md} = \hat{K}_{md}/(\mu[\mathrm{totalCdc25}])$	0.1
$\hat{K}_{mdr}$	1.0 $\mathrm{AU_d}$	$K_{mdr} = \hat{K}_{mdr}/(\mu[\mathrm{totalCdc25}])$	1.0
$\hat{K}_{mw}$	0.1 $\mathrm{AU_w}$	$K_{mw} = \hat{K}_{mw}/(\mu[\mathrm{totalWee1}])$	0.1
$\hat{K}_{mwr}$	1.0 $\mathrm{AU_w}$	$K_{mwr} = \hat{K}_{mwr}/(\mu[\mathrm{totalWee1}])$	1.0
[totalCdk1]	100 nM		
[totalCdc25]	1 $\mathrm{AU_d}$		
[totalWee1]	1 $\mathrm{AU_w}$		
[PPase]	1 $\mathrm{AU_p}$		

A fundamental proposal of the Novak and Tyson (1993) model of MPF oscillations in frog egg extracts is that the processes of MPF activation (dephosphorylation of preMPF by Cdc25) and MPF inactivation (phosphorylation of MPF by Wee1) are jump transitions on a hysteresis loop. To see this, we solve Eqs. (2.1)–(2.3) for the steady-state (ss) concentrations of MPF, Cdc25, and Wee1, and plot M_{ss} as a function of C_T (total cyclin added to the extract, a parameter easily controlled by the experimentalist). The graph (Fig. 2.3), called a one-parameter bifurcation diagram, shows that, for $C_I < C_T < C_A$, the MPF control system exhibits bistability: two stable steady states (nodes) separated by an unstable steady state (a saddle point). One node has low MPF activity (most Cdk1:cyclin dimers are in the less active, phosphorylated form) and the other node has high MPF activity. The low-activity state corresponds to "interphase" (when DNA is being replicated) and the high-activity state corresponds to "mitosis" (when chromosomes are aligned on the mitotic spindle and then sister chromatids are separated). At the limits of the bistable region, the stable steady states are annihilated by coalescence with the unstable steady state (called a "saddle-node" bifurcation).

In a classic experiment, Solomon et al. (1990) prepared an egg extract in interphase (no endogenous cyclin) and supplemented it with exogenous, non-degradable cyclin. They

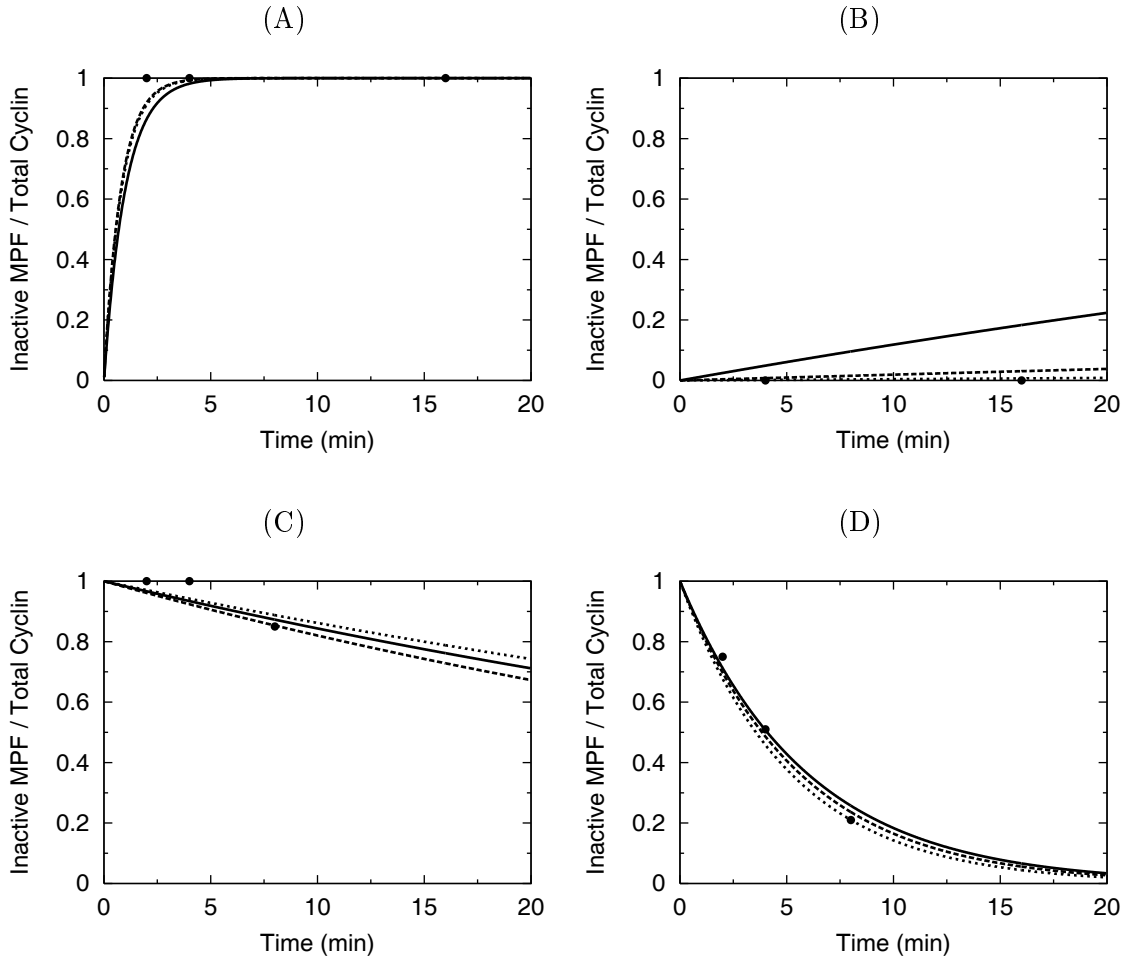

Fig. 2.2. Experimental data (•) used for parameter estimation, simulations (... and - - -) generated by the optimal parameters with Ks fixed and constrained, respectively, and simulations (—) generated by the Marlovits et al. (1998) parameters. (A) Kumagai and Dunphy (1995), Fig. 3C. Phosphorylation of MPF during interphase, when Wee1 is more active. (B) Kumagai and Dunphy (1995), Fig. 3C. Phosphorylation of MPF during mitosis, when Wee1 is less active. (C) Kumagai and Dunphy (1995), Fig. 4B. Dephosphorylation of preMPF during interphase, when Cdc25 is less active. (D) Kumagai and Dunphy (1995), Fig. 4B. Dephosphorylation of preMPF during mitosis, when Cdc25 is more active. (E) Kumagai and Dunphy (1992), Fig. 10A. Phosphorylation of Cdc25 during mitosis, when MPF is more active. (F) Kumagai and Dunphy (1992), Fig. 10A. Dephosphorylation of Cdc25 during interphase. (G) Tang et al. (1993), Fig. 2. Phosphorylation of Wee1 during mitosis, when MPF is more active. (H) Tang et al. (1993), remark in text (p. 3430). Dephosphorylation of Wee1 during interphase. (I) Moore (1997), time lag for MPF activation. (J) Moore (1997), thresholds for MPF activation (↑) and inactivation (↓).

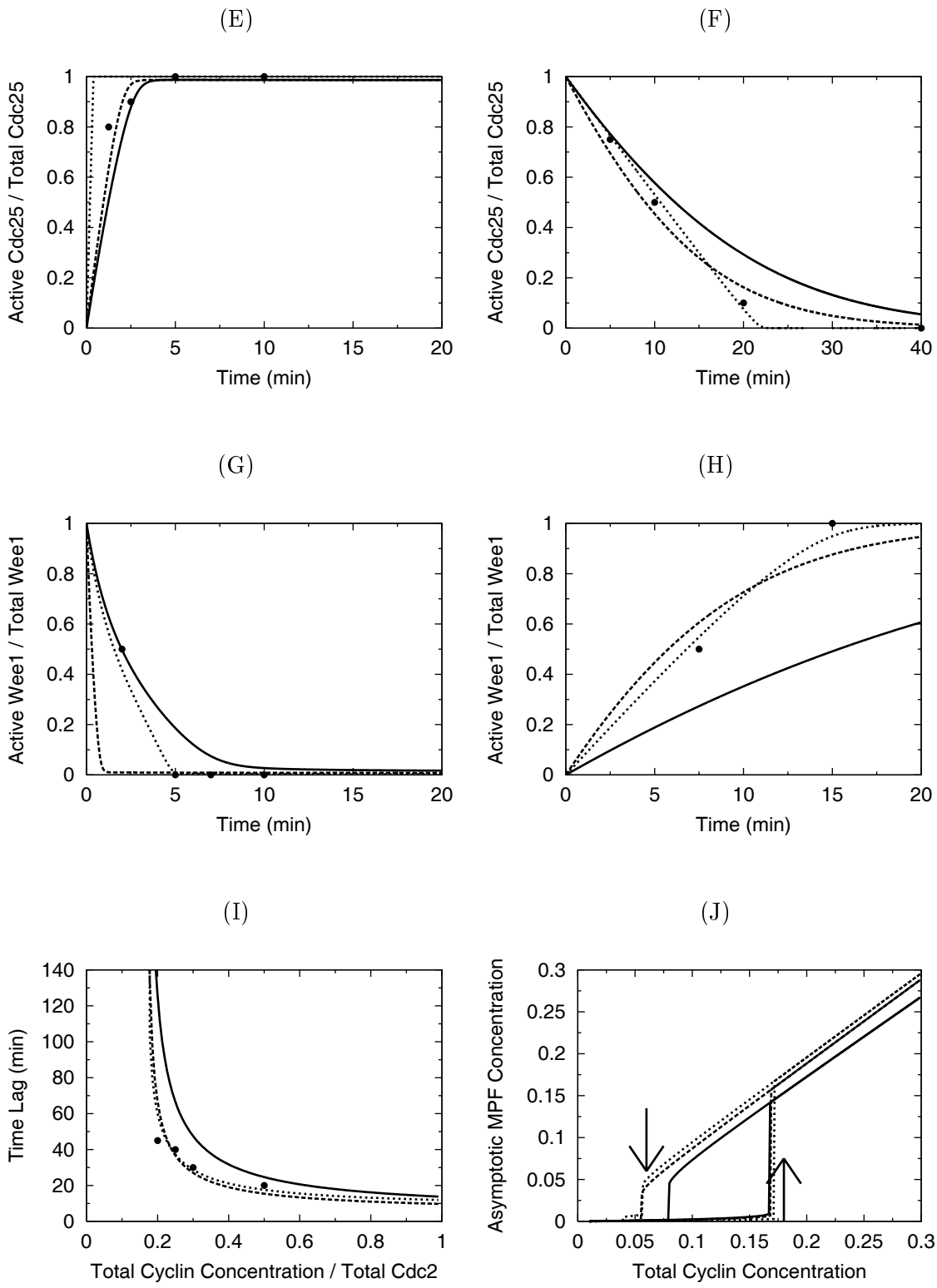

Fig. 2.2. Continued.

Table 2.2. Initial and final estimated parameter values. The initial values come from Marlovits et al. (1998), and the final values are our optimal parameter set for fitting the data in Fig. 2.2. We present two "optimal" parameter sets: one with fixed Ks (the Michaelis constants are fixed at the values chosen by Marlovits et al. (1998)), and one with constrained Ks (the Michaelis constants are allowed to vary, but they may not fall below 0.01). The last row reports the value of the error function E for each of the three parameter sets.

Rate Constant	Initial	Fixed Ks	Constrained Ks
v'_d	0.017	0.0198	0.0148
v''_d	0.17	0.181	0.195
ρ_d	0.05	0.0537	0.0044
v'_w	0.01	0.00	0.00
v''_w	1.0	1.256	1.213
ρ_w	0.05	0.0303	0.036
K_{md}	0.1	0.1	0.01
K_{mdr}	1.0	1.0	0.01
K_{mw}	0.1	0.1	0.01
K_{mwr}	1.0	1.0	0.072
v_d	2.0	2.563	10.7
v_w	2.0	8.775	2.344
$E(\beta)$	68.9	9.58	5.02

found that if $C_T < C_A$, then the extract remains in interphase, with little detectable MPF activity. However, if $C_T > C_A$, then the extract will eventually enter mitosis (after a time lag), and the amount of MPF activity generated by the extract will increase with increasing C_T. These observations are consistent with the theoretical picture in Fig. 2.3.

Novak and Tyson drew two further conclusions from Fig. 2.3. (1) As C_T is lowered toward the threshold ($C_T \to C_A^+$), the time lag before MPF activation should increase abruptly. (2) There should be a different threshold C_I for MPF inactivation in extracts that are transiting from mitosis back to interphase ($C_T = C_I$ for the inactivation threshold in Fig. 2.3). The inactivation threshold can be measured by supplementing an interphase extract with a combination of exogenously produced, degradable and nondegradable cyclins. Let C_T be the total concentration of added cyclin and f be the fraction that is nondegradable. If $C_T > C_A$, then the extract will enter mitosis, activate cyclosomes, and destroy the degradable fraction of cyclins. If the amount of cyclin remaining (fC_T) is greater than the inactivation threshold (C_I) then the extract will remain in mitosis. But if $fC_T < C_I$, then the extract will exit mitosis and return to interphase.

In a recent paper, Sha et al. (2003) have tested these two predictions of the Novak-Tyson model. Some early experiments (unpublished) from Jonathan Moore are presented in Figs. 2.2I and 2.2J. Fig. 2.2I shows the time lag for MPF activation, which clearly

Table 2.3. Parameter values, initial conditions, and weights assigned to each experiment in Fig. 2.2.

		Data		**Weights**		**Params**		**Initial Conditions**		
Exp	**Notes**	x	y	w_δ	w_ϵ	μ	C_T	M	D	W
A	(1)	t	$L(t)$			1	0	0	0	1
		2	1	1	100					
		4	1	1	100					
		16	1	1	100					
B	(1)	t	$L(t)$			1	1	1	1	0
		4	0	1	100					
		16	0	1	100					
C	(2)	t	$L_p(t)$			1	0	0	0	1
		2	1	1	100					
		4	1	1	100					
		8	0.85	1	100					
D	(2)	t	$L_p(t)$			1	1	1	1	0
		2	0.75	1	100					
		4	0.51	1	100					
		8	0.21	1	100					
E		t	$D(t)$			0.83	0.25	0.25	0	0
		1.25	0.8	1	100					
		2.5	0.9	1	100					
		5	1	1	100					
		10	1	1	100					
F		t	$D(t)$			0.83	0	0	1	0
		5	0.75	1	100					
		10	0.5	1	100					
		20	0.1	1	100					
		40	0	1	100					
G		t	$W(t)$			0.67	0.254	0.254	1	1
		2	0.5	1	100					
		5	0	1	100					
		7	0	1	100					
		10	0	1	100					
H		t	$W(t)$			0.67	0	0	1	0
		7.5	0.5	1	100					
		15	1	1	100					

Table 2.3. Continued.

		Data		Weights		Params		Initial Conditions		
Exp	**Notes**	x	y	w_δ	w_ϵ	μ	C_T	M	D	W
I	(3,4)	C_T	Lag			-	-	0	0	1
		0.2	45	2500	0.01					
		0.25	40	2500	0.01					
		0.3	30	2500	0.01					
		0.5	20	2500	0.01					
J	(3,5)	-	Thresh			-	-			
		-	0.18	-	2500			0	0	1
		-	3	-	11			1	1	0

Notes

(1) For experiments A and B, Eqs (2.1)–(2.3) are supplemented by

$$\frac{dL}{dt} = (v'_w(1-W) + v''_w W)(1-L), \quad L(0) = 0.$$

(2) For experiments C and D, Eqs (2.1)–(2.3) are supplemented by

$$\frac{dL_p}{dt} = (v'_d(1-D) + v''_d D)L_p, \quad L_p(0) = 1.$$

(3) For experiments I and J, μ is considered to be an adjustable parameter. Initially, it is set to 1, and it reaches an optimal value of 1.015. This means that Moore's extracts have effectively the same dilution as Dunphy's.

(4) See Appendix A for a description of how to compute lag times for MPF activation.

(5) See Appendix B for a description of how to compute thresholds for MPF activation and inactivation.

increases as C_T decreases. Fig. 2.2J represents the thresholds for MPF activation and inactivation. These observations placed the activation threshold C_A between 0.15 and 0.20 and the inactivation threshold C_I about three-fold lower (0.06). Other experiments confirming predictions of the Novak-Tyson model can be found in Sha et al. (2003) and Pomerening et al. (2003).

2.4. METHODS

We use the public domain software package ODRPACK to estimate the parameters of our model by minimizing the weighted sum of orthogonal distances between the experimental data and the predictions of the model. The model equations are integrated by LSODAR, a public domain software package that efficiently solves stiff and nonstiff systems of ODEs.

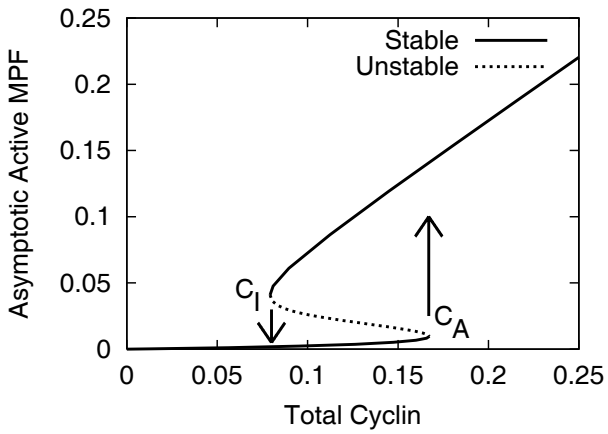

Fig. 2.3. Steady-state MPF activity versus total cyclin concentration. This S-shaped curve was computed from Eqs. (2.1)–(2.3) by setting all time derivatives equal to 0 and solving the resulting nonlinear algebraic equations for M as a function of C_T. All other parameters in the equations are given the values estimated by Marlovits et al. (1998), see Table 2.1. C_A = activation threshold, C_I = inactivation threshold.

LSODAR was chosen for its ability to solve stiff ODE systems and find roots. Like most models of molecular regulatory systems, the frog egg extract model is stiff, i.e. it includes reactions that proceed on time scales with different orders of magnitude. LSODAR performs well on such models because it automatically switches between stiff and non-stiff methods allowing integration to proceed as fast as possible. The root finding capability of LSODAR is also useful for computing timelags (discussed more later).

Orthogonal distance regression (ODR) was chosen to quantify goodness of fit because our data (Fig. 2.2) contain considerable error in the independent variables (abscissa) as well as the dependent variables (ordinate). Carroll, Ruppert, and Stefanski (1995) have argued that ODR is not a reliable method to obtain statistically sound estimates of parameters in cases where the analyst has many replicates of inherently noisy correlations. This is not the case here, where we have few measurements (often crudely taken, because the experiments are technically challenging) of strongly correlated variables, from which we want to derive trends and quantify rate constants. We are not attempting to put error bars on our estimates, which would be unsupportable from the data at hand. We do claim, however, that ODR captures accurately what modelers have in mind by "goodness of fit". By minimizing orthogonal distance, we obtain fits to the data (Fig. 2.2) that anyone would consider improvements over the original "guesstimates" of Marlovits et al. (1998).

The separate panels of Fig. 2.2 are especially sensitive to particular rate constants in the model: A—v''_w, B—v'_w, C—v'_d, D—v''_d, E—v_d, F—$v_d\rho_d$, G—v_w, H—$v_w\rho_w$. Clearly, there is enough data in these figures to estimate these eight rate constants with some reliability. In addition, the "physiological" measurements in Figs. 2.2I and 2.2J provide additional constraints on these rate constants. On the other hand, the data does not provide strong constraints on the Michaelis constants (K_{md}, K_{mdr}, K_{mw}, K_{mwr}). For this reason, we do not try to estimate the Ks from the data.

2.4.1. ODRPACK

In general, a model predicts the values of dependent variables (y) from independent variables (x) and parameter values (β):

$$y_i = f_i(x_i; \beta), i = 1 \ldots n,$$

where y, x, and β are (in general) all vectors and f_i is the model for the ith datum. Suppose the experimental data can be expressed by vectors y_i and x_i, $i = 1, \ldots, n$. If the model fits the data perfectly, for a specific parameter vector $\hat{\beta}$, then $y_i = f_i(x_i; \hat{\beta})$ for all i. It is always the case that there are discrepancies between the model and observations because of errors in the measurements and/or inadequacies of the model. ODRPACK constructs an objective function that is a scalar measure of the goodness-of-fit of the model to the data and then minimizes this function over a domain of parameter values.

ODRPACK does not assume that all measurement errors are in the dependent variables. Rather, it seeks to minimize the weighted sum of orthogonal distances between the model and the data:

$$E_{min} = \min_{\beta,\delta} \left(\sum_{i=1}^{n} \epsilon_i^T w_{\epsilon_i} \epsilon_i + \delta_i^T w_{\delta_i} \delta_i \right), \tag{2.4}$$

subject to the constraints

$$\epsilon_i = f_i(x_i + \delta_i; \beta) - y_i, i = 1, \ldots, n. \tag{2.5}$$

In Eq. (2.4), ϵ_i and δ_i are the vectors of residuals for the dependent and independent variables, respectively, and w_{δ_i} and w_{ϵ_i} are matrices of weighting factors for the errors. The weighting factors are used to translate all observables into a common "currency" and to express the user's confidence in the reliability of different observations. The solution of problem (2.4)–(2.5) consists of $\tilde{\beta}, \tilde{\delta}, \tilde{\epsilon}, E_{min}$, giving the optimal parameter vector, the minimal discrepancies between model and observations, and a scalar measure of the overall goodness-of-fit. Convergence to the minimum solution $\tilde{\beta}$, $\tilde{\delta}$, $\tilde{\epsilon}$, and E_{min} is achieved by adjusting β and δ, where δ is treated like β—as an independent unknown.

In our problem, all our data points are ordered pairs (x_i, y_i) of scalar quantities, and our objective function is simply

$$E = \sum_{i=1}^{n} w_{\epsilon_i} \epsilon_i^2 + w_{\delta_i} \delta_i^2.$$

In Table 2.3 we collect all the information necessary to compute E from the model (Eqs. (2.1)–(2.3)) and the data (Fig. 2.2).

ODRPACK uses a trust region Levenberg-Marquardt method with scaling to minimize E (Boggs et al., 1992). In doing so, ODRPACK needs to calculate Jacobian matrices (partial derivatives of the weighted vector (ϵ, δ) with respect to β and δ). ODRPACK can calculate these matrices by finite differences or by a user-supplied routine. Finite differences were used here. The ODRPACK code is described in detail in Boggs, Byrd, and Schnabel (1987). A brief outline is given here that helps explain how the δs are treated and what needs to be calculated to update the parameters. Note that when calculating the Jacobian matrix, the partials with respect to the δs must be calculated and the weighted δs are included in the list of functions to be minimized (along with $\sqrt{w_{\epsilon_i}}(f_i - y_i)$, $i = 1, \ldots, n$). The Jacobian matrix of the weighted error vector (ϵ, δ) will have the structure

$$J = \begin{bmatrix} G & V \\ 0 & D \end{bmatrix},$$

where G, V, and D are defined in the pseudocode below.

do until parameters converge, sum of squares converge, or iteration limit reached.

$G_{ij} = \sqrt{w_{\epsilon_i}}\partial f_i(x_i + \delta_i; \beta)/\partial\beta_j$, $i = 1, \ldots, n$, $j = 1, \ldots, p$, where p is the number of parameters; calculate the Jacobian matrix like in ordinary least squares.

$V = \mathrm{diag}\{\sqrt{w_{\epsilon_i}}\partial f_i(x_i + \delta_i; \beta)/\partial\delta_i,\ i = 1, \ldots, n\}$; the partials of f with respect to the δs. Note that f_i only depends on δ_i. This realization makes ODRPACK very efficient (i.e., the time complexity is reduced from quadratic to linear with respect to n).

$D = \mathrm{diag}\{\sqrt{w_{\delta_i}},\ i = 1, \ldots, n\}$; the partials of the weighted δs with respect to each δ. Note again that $w_{\delta_i} * \delta_i$ only depends on δ_i giving a diagonal matrix and a similar reduction in time complexity as with V.

$P = V^T V + D^2$;

Formulate the linear least squares problem (derived from the linearization of E)

$$\min_s \left\| \left((I - VP^{-1}V^T)^{\frac{1}{2}} Gs\right) - (I - VP^{-1}V^T)^{-\frac{1}{2}}\left(-\epsilon + VP^{-1}(V^T\epsilon + D\delta)\right) \right\|^2,$$

and solve for s with a QR factorization of the coefficient matrix of s. Note that Boggs, Byrd, and Schnabel (1987) realized the ODR problem can be solved efficiently this way instead of solving the normal equations with the full Jacobian matrix J.

$t = -P^{-1}(V^T\epsilon + D\delta + V^T Gs)$;

Use s and t to update β and δ, respectively. The Levenberg-Marquardt method starts with the steepest decent method and smoothly changes to Newton's method, where s and t are simply added to β and δ, as the solution is approached. ODRPACK uses a trust region implementation of the Levenberg-Marquardt method which reduces the step size based on the confidence in a model of the objective function. See Moré and Wright (1993) for details on how parameters are updated in the Levenberg-Marquardt algorithm.

end do

2.4.2. LSODAR

All solutions of the ODEs (2.1)–(2.3) were computed by LSODAR, a variant of LSODE [(Radhakrishnan and Hindmarsh, 1993), (Hindmarsh, 1980), (Hindmarsh, 1983)], which automatically switches between stiff and nonstiff methods and has a root finder. LSODAR starts with a nonstiff method and switches to a stiff method if necessary. LSODAR's root finder is used in this application to find the time lag for MPF activation.

For nonstiff problems, LSODAR uses Adams-Moulton (AM) of orders 1 to 12. For stiff problems, LSODAR uses backward differentiation formulas (BDF) of orders 1 to 5. With both methods, LSODAR varies the step size and order. LSODAR switches from AM to BDF when AM is no longer stable for the problem or cannot meet the accuracy requirements efficiently (Petzold, 1983).

The root finder in LSODAR is based on ZEROIN (Shampine and Allen, 1973). ZEROIN is based on code by Dekker (1969). LSODAR detects a root when the sign changes for the user-defined subroutine GEX.

The tolerances are set to 10^{-12} for both relative and absolute error. A tolerance of 10^{-10} is used when calculating a root for a function of the form $M(t) - M_{root}$, where M_{root} is the value of the function $M(t)$ for which a time, t, is desired.

2.4.3 Weights

The choice of weights, w_{ϵ_i} and w_{δ_i}, should express the relative reliabilities of the measurements. Ideally, if we have many replicated measurements, we can compute the reliabilities, and hence the weights, from the variance of the measurements. In our case, we do not have this luxury, and we must provide some informal estimate of our confidence in the data. The weights are also used to quantify our assessment of the relative importance of different experiments.

To these ends, we suggest weights of the form

$$w_{\epsilon_i} = \frac{\alpha_i}{\sigma_{\epsilon_i}^2}, \qquad w_{\delta_i} = \frac{\alpha_i}{\sigma_{\delta_i}^2},$$

where α_i is a dimensionless number that assigns a relative importance to the ith data vector; σ_{ϵ_i} and σ_{δ_i} are constants, carrying the same units as y_i and x_i, respectively, that reflect our relative uncertainties in the measurements (ideally, they should be estimates of the standard deviation of the measurements).

In our case, we choose $\alpha_i = 1$ for all the weights. Our choices of σ_{ϵ_i} and σ_{δ_i} represent a subjective estimate of the standard deviation of each measurement. In Table 2.3, we collect all the information necessary to compute E from the model (Eqs. (2.1)–(2.3)) and the data (Fig. 2.2).

2.5. RESULTS

We have fitted the data in Fig. 2.2 to two different sets of "optimal" parameter values; the results are reported in Fig. 2.2 and Table 2.2. In the first set of optimized parameter values, we fixed the four Michaelis constants at the values chosen by Marlovits et al. (1998). In the second set, we allowed the Michaelis constants to adjust to the data, provided none of them fall below 0.01. Small values for K imply an enzyme with very high affinity for substrate and generate rate laws with sharp "kinks." In our opinion, $K < 0.01$ is biochemically unrealistic, even though the kinks provide moderately better fits to the data. Whether the Ks are fixed or constrained, the optimal values of the other rate constants are in reasonable agreement amongst themselves and compared with the initial estimates of Marlovits et al. (1998) (Table 2.2). By examining the curves in Fig. 2.2, one sees clearly that the guesstimates of Marlovits et al. (1998) were pretty good, but ODRPACK was able to find nearby parameter sets that give a better fit of the model to the data.

For both optimization procedures, ODRPACK drove v'_w to very small values, so we fixed $v'_w = 0$ in both cases. In the "fixed K" case, ODRPACK determined that $\rho_d \approx \rho_w$, which gives a simple S-shaped bifurcation curve, as in Fig. 2.3. For the "constrained K" case, ODRPACK finds an optimal parameter set with $\rho_d \approx \rho_w/10$, in which case the steady-state bifurcation curve (Fig. 2.3) is double-S-shaped. This effect can be seen in Fig. 2.2(J), at low cyclin concentrations, where there appear to be two jumps in MPF activity. The experimental data are not good enough to distinguish between these two possibilities. For the sake of simplicity, we prefer the "fixed K" parameter values.

To ensure that ODRPACK found a local minimum of the objective function, we calculated $E(\beta)$ in a neighborhood of β^0, where $\beta^0 = \{p_1^0, p_2^0, \ldots, p_8^0\}$ is the set of optimal parameter values. (For fixed K, there are eight parameters to be optimized: the seven adjustable rate constants in Table 2.2 plus the dilution factor, μ, as described in footnote (3) to Table 2.3.) The evaluations were made on a regular grid of points at $0.975p_i^0$, $1.0p_i^0$, and $1.025p_i^0$ in each of the eight directions. All $6,561$ evaluations of E satisfied $E(\beta) > E(\beta^0)$, confirming that ODRPACK had indeed found a local minimum of the objective function.

In addition, we fitted these $6,561$ values of E to a function of the form

$$\tilde{E}(\beta) = E(\beta^0) + \sum_{i,j=1}^{8} a_{ij} \left(\frac{p_i - p_i^0}{p_i^0} \right) \left(\frac{p_j - p_j^0}{p_j^0} \right).$$

The matrix of coefficients, $A = [a_{ij}]$, is symmetric and, hence, can be diagonalized by an orthogonal matrix Q; furthermore, all the eigenvalues of A are real (Gantmacher, 1977). That is to say, $Q^T A Q = \Lambda = diag(\lambda_1, \ldots, \lambda_8)$, where $Q^T Q = I$, $\lambda_1, \ldots, \lambda_8$ are real numbers, and the columns of Q (call them q_i) are the eigenvectors of A. With this information, we can

estimate the deviations of E from $E(\beta^0)$ as follows. Let π be a vector whose components are $\pi_i = (p_i - p_i^0)/p_i^0, i = 1, \ldots, 8$. Then

$$\tilde{E}(\beta) - E(\beta^0) = \pi^T A \pi = \pi^T Q Q^T A Q Q^T \pi = \chi^T \Lambda \chi,$$

where $\chi = Q^T \pi$. Hence, the eigenvalues and eigenvectors of A give us useful information about the shape of E in the neighborhood of $E(\beta^0)$ (provided, of course, that $\tilde{E}$ is a good approximation to E). In our case, the eigenvalues and eigenvectors of A are reported in Table 2.4. Because all the eigenvalues are positive, $\tilde{E}$ (considered as a function of π_is) is an eight-dimensional bowl with a local minimum at the origin ($\pi_i = 0$ for all i). Furthermore, the bowl is very flat (i.e., quality of the fit is not sensitive to modest changes in parameter values), in the following sense. $\tilde{E}$ changes most rapidly in the direction of the first eigenvector, because $\lambda_1 = 908.7$ is the largest eigenvalue. For a 2.5% change in parameter values in the direction of q_1, we estimate that $\tilde{E} - E^0 = 0.57$ (Fig. 2.4), where E^0 is the error at β^0. Since $E^0 = 9.58$, this is only a 6% increase in error. If we dare to extrapolate beyond the data set, then a 4% change in parameter values (in the direction of q_1) would lead to only a 15% increase in error. Hence, we conclude that ODRPACK has indeed found a robust, locally optimal parameter set for this problem.

Table 2.4. The eigenvalues and eigenvectors of A (see text).

i	λ_i	q_{i1} (v'_d)	q_{i2} (v''_d)	q_{i3} (ρ_d)	q_{i4} (v''_w)	q_{i5} (ρ_w)	q_{i6} (v_d)	q_{i7} (v_w)	q_{i8} (μ)
1	908.7	0.3305	0.3779	−0.328	−0.653	−0.192	0.0491	0.0336	−0.415
2	168.0	0.0410	0.1622	0.5665	−0.118	−0.371	0.6804	−0.094	0.1641
3	135.1	0.1371	−0.049	0.2365	−0.098	0.7422	0.3153	0.4535	−0.238
4	60.0	−0.049	0.5087	−0.114	−0.024	−0.039	−0.112	0.5868	0.6052
5	26.3	0.1775	−0.749	−0.153	−0.387	−0.199	0.0837	0.2954	0.3156
6	10.8	0.8153	0.0691	0.0483	0.1061	0.2272	−0.099	−0.353	0.3606
7	6.3	0.0489	0.0092	−0.669	0.4024	0.0066	0.6215	−0.012	0.0140
8	2.8	0.4112	−0.057	0.1649	0.4754	−0.425	−0.138	0.4765	−0.383

2.6. DISCUSSION

Computational models of cell growth and division involve mathematical representation of a complex network of biochemical reactions within cells. These reactions can be described by a system of nonlinear ordinary differential equations, according to the principles of biochemical kinetics. Rate constants and binding constants enter as parameters in the differential equations, and must be estimated by fitting solutions of the equations to experimental data.

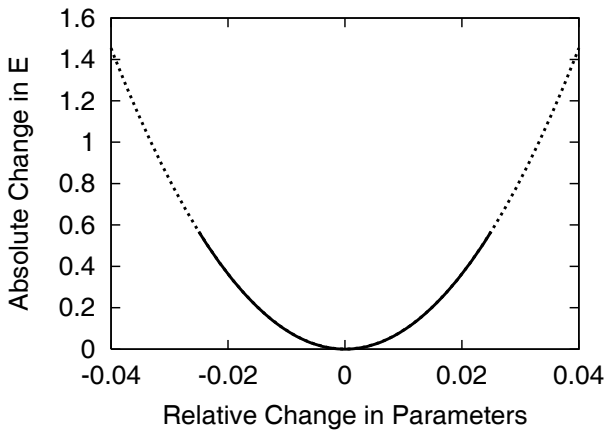

Fig. 2.4. Absolute changes in the error function with respect to relative changes in the parameters in the direction of q_1. q_1 is the eigenvector corresponding to the maximum eigenvalue. Therefore, this plot represents the maximum change in the error function with respect to the parameters in the neighborhood of the optimal parameters. The solid line (—) represents the range of the fitted data. The dotted line (...) represents an extrapolation of the quadratic fit. The error function is plotted for the "fixed K" case ($\lambda_1 = 909$). The curve is almost the same for the "constrained K" case ($\lambda_1 = 1168$).

ODRPACK, based on the orthogonal distance between experimental data and the model, is used for the nonlinear regression to estimate the unknown rate constants (ODE parameters). The ability of this algorithm to weight data values arbitrarily, and to treat both the abscissa and ordinate as uncertain, is crucial, given the sparsity and uncertainty of available biological data. Orthogonal distance conforms well to the modeler's intuition of a good fit to the type of data commonly encountered in this field.

The complete calculation is expensive, because the ODEs are stiff, and must be solved numerous times for the nonlinear regression. Some of the runs of ODRPACK for this thesis took 20 minutes to complete. Also, because of local minima, the nonlinear regression must be done from many starting points (possibly taking 20 minutes per starting point) to adequately explore the parameter space. There are potential sources for parallelism in multiple starting points for regression and in the multiple experiments being simulated.

To study realistic models of cell cycle control, more components must be added to the model, and other measurable phenomena incorporated in the cost function. As the modeling fidelity is increased, the mathematical and computational complexities of the problem grow rapidly. Efficient and accurate tools for parameter estimation will be needed to build computational models of the complex control networks operating within cells, which is one of the main goals of bioinformatics in the postgenomic era.

CHAPTER 3: Globally Optimized Parameters for a Frog Egg Model

3.1. INTRODUCTION

The physiological attributes of a cell, its abilities to move and feed, to respond to external stimuli, to grow and reproduce, to repair damage, etc., are controlled ultimately by complex networks of interacting genes, proteins and metabolites ((Bray, 1995), (Hanahan and Weinberg, 2000)). These mechanisms can be exceedingly complex, with hundreds or thousands of interacting components (Kohn, 1999), and ferreting out the implications of these "wiring diagrams" is beyond the scope of intuitive biochemical reasoning or reductionistic experimental data collection ((Hartwell et al., 1999), (Brent, 2000), (Lazebnik, 2002)). New theoretical and computational methods are needed to make sense of the data and to gain insights into the "molecular logic" of intracellular regulatory systems ((Hasty et al., 2001), (Tyson et al., 2001), (Brazhnik et al., 2002), (Kumar and Feidler, 2003)).

The gold standard of computational modeling in this domain are "bottom-up" models based on detailed biochemical kinetic descriptions of the underlying control systems, for example ((Martiel and Goldbeter, 1972), (Bray et al., 1993), (McAdams and Shapiro, 1995), (Arkin et al., 1998), (Sharp and Reinitz, 1998), (Teusink et al., 2000), (von Dassow et al., 2000), (Asthagiri and Lauffenburger, 2001), (Meinhardt and de Boer, 2001), (Chen et al., 2004)). These models are usually framed in terms of nonlinear differential equations (ordinary, partial, or stochastic). When properly formulated, such models have distinct advantages. (1) They are closely allied to real molecular processes occurring inside cells. (2) They provide quantitatively accurate accounts of the physiological properties of cells. And (3) they can provide reliable predictions of hitherto unobserved properties of normal and mutant cells (for example, (Cross et al., 2002), (Hoffmann et al., 2002), (Pomerening et al., 2003), (Sha et al., 2003)).

A major drawback of bottom-up models is that they contain many kinetic rate constants whose numerical values are unknown at the outset of the modeling exercise. The rate constants must be inferred by fitting simulations of the model to experimental observations. Parameter identification may be done by careful collection of biochemical data on component reaction steps ((Teusink et al., 2000), (Eissing et al., 2004)), by tedious fitting to a large collection of qualitative characteristics of cells ((Bray et al., 1993), (Chen et al., 2004)), or by rough agreement to a few crucial features of a cell's behavior ((Barkai and Leibler, 1997), (von Dassow et al., 2000)). When a sufficient amount of quantitatively reliable data is available, a modeler should estimate rate constants by optimizing the goodness-of-fit of the model equations to the data.

Parameter optimization begins by defining an objective function that measures the distance between the "model" and the "data". Generally, this distance is computed as a weighted sum of squares of distances between observed data points and corresponding simulated points. Once the objective function has been defined and computed, there are

many well-tested algorithms for minimizing the function over the space of model parameters (rate constants). These algorithms are generally classified as local or global, depending on the scope of their search. Local optimization algorithms stop at the first sign of a local minimum, i.e., when the value of the objective function is lower at some point in parameter space than it is at any nearby points. Local algorithms usually work by identifying "downhill" directions in parameter space and then moving down hill as fast as possible to a local minimum. Local algorithms are computationally efficient, but they often fail to find better solutions of the optimization problem (deeper pits of the objective function) in far away regions of parameter space. Global optimization algorithms have some ability to search beyond local minima to see if better fits can be found elsewhere in parameter space. Global search algorithms generally combine an exploration step (which may be quite random or systematic) with a selection step (favoring lower values of the objective function).

We have been investigating the parameter optimization problem for a particular example of bottom-up modeling: the control of cell division in frog eggs. In Zwolak et al. (2004), we used local optimization to refine a set of rate constants originally estimated by Marlovits et al. (1998) by rough fitting of a mathematical model to biochemical data obtained from frog egg extracts. In this chapter, we apply a global optimization algorithm to show that our 2004 set of rate constants is indeed a globally optimum set of parameters. As a bonus, we find that the underlying mathematical model can be simplified somewhat, by replacing a Michaelis-Menten rate law (two kinetic parameters) with a mass-action rate law (one kinetic parameter).

3.2. PROBLEM DESCRIPTION

Our goal is to find an optimal parameter set for a mathematical model of the biochemistry underlying DNA synthesis and nuclear division in frog egg extracts (Marlovits et al., 1998). The model (Fig. 2.1), proposed originally by Novak and Tyson (Novak and Tyson, 1993), consists of a reaction network (protein species interconnected by chemical reactions) whose dynamics are cast as a set of nonlinear ordinary differential equations (ODEs). We stick with an ODE representation, because ODEs are easily simulated by computers and they accurately describe the chemical kinetics of well stirred systems. We may treat the egg extracts as well stirred because diffusion and transport occur much faster than the chemical reactions under consideration. The experimental data to be fit ((Kumagai and Dunphy, 1992), (Kumagai and Dunphy, 1995), (Sha et al., 2003), (Tang et al., 1993)) are often images of "spots" on polyacrylamide gels. The intensity of each spot represents the abundance of a particular protein species in the cell. We must estimate the relative intensity of a spot (say, 60% of maximum) in order to have numerical results for computer calculations.

3.2.1 Model

The reaction network in Fig. 2.1 can be converted into a set of ODEs for the time rates of change of the concentrations of all the proteins in the reaction mechanism. Before doing so, we make appropriate simplifications to the mechanism in Marlovits et al. (1998). All the experiments analyzed in this chapter are carried out in frog egg extracts supplemented with cycloheximide, an inhibitor of protein synthesis. Hence the extract does not synthesize cyclin from its own store of cyclin mRNA, so we set $k_1 = 0$ and ignore this reaction. The experimenter adds to the extract a known amount of exogenously produced cyclin. In all cases, the added cyclin protein has been genetically engineered to be resistant to cyclosome-mediated degradation. Hence $k_2 = 0$, and we can ignore all the degradation steps in the mechanism. Because cyclin is neither synthesized nor destroyed, the total concentration of cyclin protein in the extract is constant:

$$[\text{monomeric cyclin}] + [\text{MPF}] + [\text{preMPF}] = [\text{total cyclin}] = \text{constant},$$

where MPF is the active Cdk1:cyclin dimer and preMPF refers to the phosphorylated (low activity) form of the dimer. Next, we make the assumptions (well supported by experiment) that (1) the binding of cyclin monomers and Cdk1 monomers is very fast (k_3 is large) and (2) [total Cdk1] > [total cyclin]. Hence [monomeric cyclin] << [total cyclin], and the conservation condition on cyclin subunits becomes

$$[\text{MPF}] + [\text{preMPF}] = [\text{total cyclin}].$$

With these simplifications, the mechanism in Fig. 2.1 can be described by three ODEs

$$\frac{dM}{dt} = \big(v_d'(1-D) + v_d''D\big)(C_T - M) - \big(v_w'(1-W) + v_w''W\big)M, \tag{3.1}$$

$$\frac{dD}{dt} = v_d\left(\frac{M(1-D)}{K_{md} + (1-D)} - \frac{\rho_d D}{K_{mdr} + D}\right), \tag{3.2}$$

$$\frac{dW}{dt} = v_w\left(-\frac{MW}{K_{mw} + W} + \frac{\rho_w(1-W)}{K_{mwr} + (1-W)}\right), \tag{3.3}$$

where

$$\begin{aligned} M &= [\text{MPF}]/[\text{total Cdk1}], \\ D &= [\text{Cdc25P}]/[\text{total Cdc25}], \\ W &= [\text{Wee1}]/[\text{total Wee1}], \\ C_T &= [\text{total cyclin}]/[\text{total Cdk1}]. \end{aligned}$$

In Eqs. (3.1)–(3.3), time is expressed in minutes, and all concentrations are dimensionless numbers, having been scaled relative to some appropriate reference concentration. In frog egg extracts, [Cdk1] is typically close to 100 nM ((Solomon et al., 1990), (Sha et al., 2003)). Total concentrations for Cdc25, Wee1 and PPase are unknown, so we set each to 1 AU

("arbitrary unit"). All vs are pseudo-first-order rate constants (units = min^{-1}). All Ks are dimensionless Michaelis constants, i.e., ratios of true Michaelis constants ($\hat{K}$, units = concentration) to reference concentrations. The ρs are dimensionless numbers expressing the activity of the phosphatase (PPase, in Fig. 2.1) relative to MPF. The rate constants refer to the following reactions:

v'_d — dephosphorylation of preMPF by Cdc25 in its less-active form,
v''_d — dephosphorylation of preMPF by Cdc25 in its more-active form,
v'_w — phosphorylation of MPF by Wee1 in its less-active form,
v''_w — phosphorylation of MPF by Wee1 in its more-active form,
v_d — phosphorylation of Cdc25 by MPF,
$v_d\rho_d$ — dephosphorylation of Cdc25 by PPase,
v_w — phosphorylation of Wee1 by MPF,
$v_w\rho_w$ — dephosphorylation of Wee1 by PPase.

Zwolak et al. (2004) describe how the parameters in Eqs. (3.1)–(3.3), the vs and Ks, are related to the fundamental kinetic parameters. They also explain how to introduce a dilution factor into the calculations, because in some experiments a buffered solution of proteins is added to the frog egg extract, increasing the volume of the extract and thereby diluting all the endogenous proteins in the extract. Dilution changes the values of some of the model's parameters from one experiment to the another, and must be taken into account when trying to fit the model to experimental observations.

3.2.2 Experiments

Our goal is to obtain the "best" estimates of the rate constants from the experimental data presented in Fig. 3.1. The figure also presents the best-fitting curves derived from Eqs. (3.1)–(3.3), using the optimal parameter values (Table 3.1). Experiments A–H in Fig. 3.1 are straightforward measurements of enzyme activity as functions of time. The data in Figs. 3.2I and 3.2J are more indirect and require further explanation.

A fundamental proposal of the Novak and Tyson (Novak and Tyson, 1993) model of MPF oscillations in frog egg extracts is that the processes of MPF activation (dephosphorylation of preMPF by Cdc25) and MPF inactivation (phosphorylation of MPF by Wee1) are jump transitions on a hysteresis loop. To see this, we solve Eqs. (3.1)–(3.3) for the steady-state (ss) concentrations of MPF, Cdc25, and Wee1, and plot (Fig. 3.1J) M_{ss} as a function of C_T (total cyclin added to the extract, a parameter easily controlled by the experimentalist). The resulting system has three distinct behaviors depending on the concentration of total cyclin. The system is either stable with low MPF activity and $C_T < C_I$, stable with high MPF activity and $C_T > C_A$, or bistable with high or low MPF activity

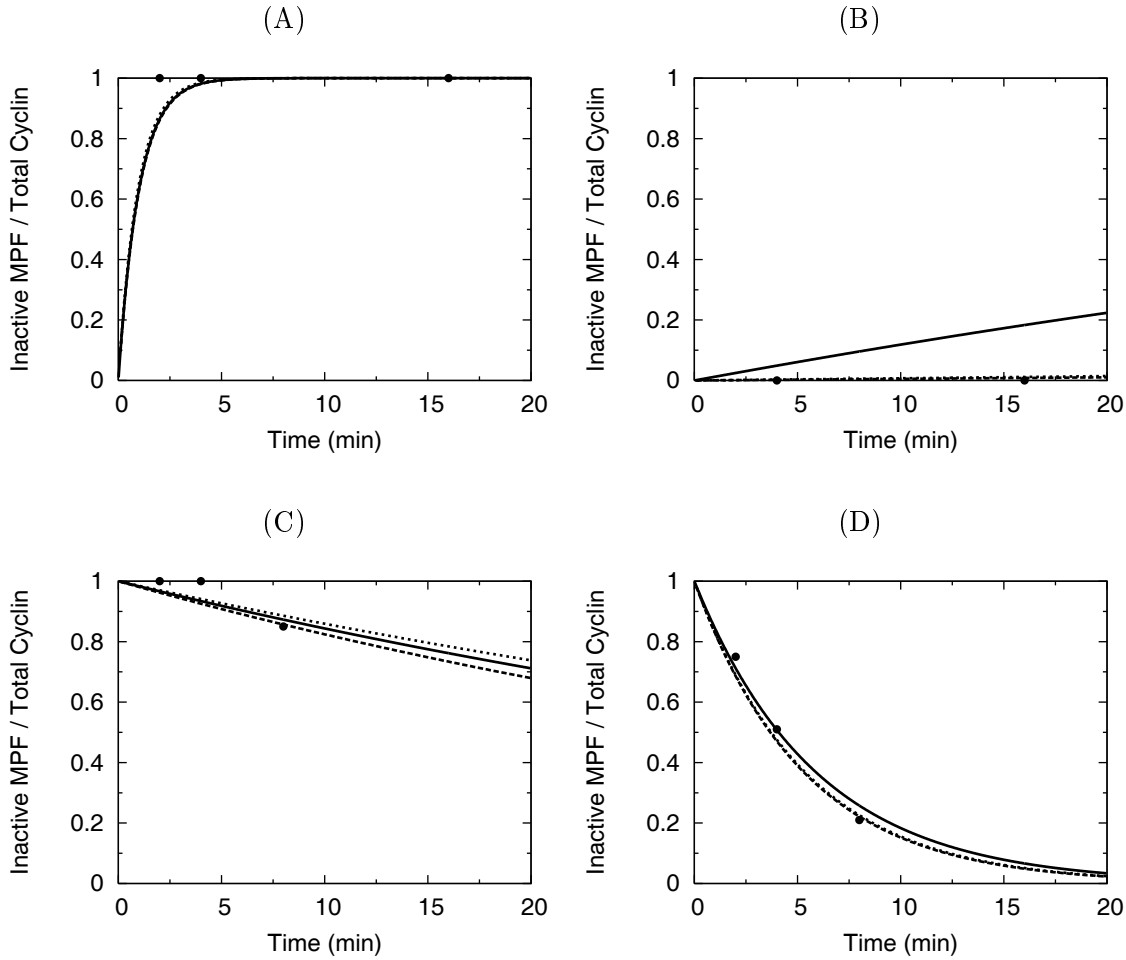

Fig. 3.1. Experimental data (•) used for parameter estimation, simulations (..... and ----) generated by the optimal points β_{Global} and $\beta_{\text{LocalOnly}}$, respectively, and simulations (——) generated by the Marlovits et al. (1998) parameters. (A) Kumagai and Dunphy (1995), Fig. 3C. Phosphorylation of MPF during interphase, when Wee1 is more active. (B) Kumagai and Dunphy (1995), Fig. 3C. Phosphorylation of MPF during mitosis, when Wee1 is less active. (C) Kumagai and Dunphy (1995), Fig. 4B. Dephosphorylation of preMPF during interphase, when Cdc25 is less active. (D) Kumagai and Dunphy (1995), Fig. 4B. Dephosphorylation of preMPF during mitosis, when Cdc25 is more active. (E) Kumagai and Dunphy (1992), Fig. 10A. Phosphorylation of Cdc25 during mitosis, when MPF is more active. (F) Kumagai and Dunphy (1992), Fig. 10B. Dephosphorylation of Cdc25 during mitosis. (G) Tang et al. (1993), Fig. 2. Phosphorylation of Wee1 during mitosis, when MPF is more active. (H) Tang et al. (1993), remark in text (p. 3430). Dephosphorylation of Wee1 during interphase. (I) Moore (1997). Time lag for MPF activation. (J) Moore (1997). Thresholds for MPF activation (↑) and inactivation (↓).

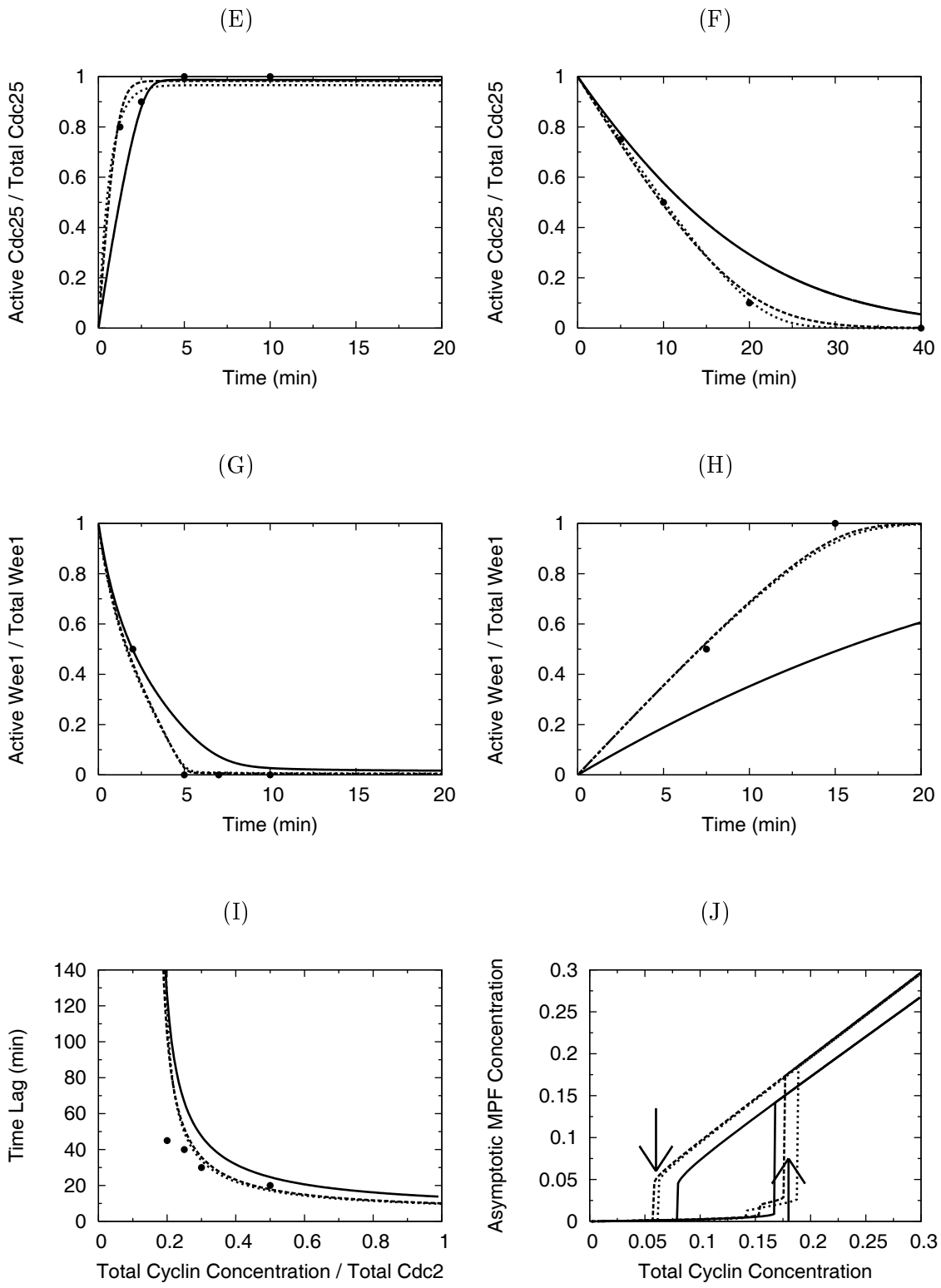

Fig. 3.1. Continued.

Table 3.1. Selected points in parameter space used and discovered in this work. All the points were obtained by fitting the experimental data summarized in Zwolak et al. (2004). The Marlovits et al. (1998) point was fit by hand. The point $\beta_{\text{LocalOnly}}$ is an optimal point obtained after local optimization was performed with $\beta_{\text{Marlovits}}$ as an initial point. β_{Global} is the result of global optimization followed by local optimization using the best point returned by the global optimizer. β_{Global3} is the result of local optimization performed on the third best point returned by the global optimizer. It is presented here for comparison to β_{Global}, as they are very similar and suggest there may be a manifold of solutions. β_{Simple} is the best point from global and local optimization on the simplified Frog Egg model with slightly different parameter names given that the model is different.

Rate Constant	$\beta_{\text{Marlovits}}$	$\beta_{\text{LocalOnly}}$	β_{Global}	β_{Global3}	β_{Simple}
v'_d	0.017	*0.0193*	0.0156	0.0153	*0.0152*
v''_d	0.17	*0.189*	0.187	0.187	*0.187*
ρ_d	0.05	0.0126	0.0013	0.0004	-
v'_w	0.01	5.1×10^{-6}	4.4×10^{-9}	7.8×10^{-7}	6.4×10^{-7}
v''_w	1.0	*0.986*	1.05	1.05	*1.05*
ρ_w	0.05	*0.0376*	0.0369	0.0366	*0.0366*
K_{md}	0.1	0.356	5.73	20.7	-
K_{mdr}	1.0	*0.257*	0.130	0.115	*0.111*
K_{mw}	0.1	*0.0141*	0.0187	0.0194	*0.0196*
K_{mwr}	1.0	*0.0596*	0.0736	0.0728	*0.0736*
v_d	2.0	5.57	44.4	152.7	-
v_w	2.0	*2.10*	2.19	2.20	*2.21*
v_{md}	-	-	-	-	7.24
v_{dr}	-	-	-	-	0.0573
$E(\beta)$	0.6001	0.03569	0.03173	0.03166	0.03165

depending on whether MPF activity was initially high or low and $C_I < C_T < C_I$ ((Novak and Tyson, 1993), (Zwolak et al., 2004)).

The predicted range of cyclin concentrations for which the MPF control system is bistable has recently been confirmed experimentally by Sha et al. (2003) and Pomerening et al. (2003). The observed range is indicated by the two arrows in Fig. 3.1J. The absolute amount of exogenously prepared, nondegradable cyclin that must be added to the extract at each transition point varies from one experiment to the next (e.g., 20–40 nanomol/L at the activation threshold), depending upon uncontrollable features of the procedure for preparing cyclin protein. Presumably, in some preparations, a fraction of the cyclin is inactive, so more prepared cyclin must be added to the extract to induce the transition. Comparing experiments, we find that the minimum amount of nondegradable cyclin needed

to induce MPF activation is 16nM, so we take this value as the activation threshold. In all experiments, the ratio of inactive threshold to the activation threshold is always 1/3 (regardless of their absolute magnitudes), so we take this ratio as our second experimental datum.

Novak and Tyson (1993) predicted further that the time lag for activation of MPF should increase dramatically as the cyclin concentration approaches the activation threshold (16nM) from above. This prediction was confirmed by Sha et al. (2003), and their data are presented in Fig. 3.1I.

In what follows, we shall refer to this collection of equations and experimental data as the "Xenopus model," and attack the problem of finding a globally optimum solution to the data-fitting problem.

3.3. METHODS AND ALGORITHMS

Parameter estimation for problems like this is a complicated business, demanding a variety of software tools for diverse tasks and careful setup of the numerical parameters in each tool. In our case, the tasks are global optimization, local optimization, integration of stiff ODEs, and "transformation" of simulation output (e.g., MPF activity as a function of time) into the format of complex observations (e.g., MPF thresholds). The first three tasks have mature tools publicly available. The fourth task is highly specific to the scientific situation and requires specialized software. For this problem, the difficult transformations involved experiments I and J in Fig. 3.1. The strategy of these transformations is described in Zwolak et al. (2004).

With these transformations in hand, we calculate the weighted sum of the squared differences of the calculated properties and the observed properties (Fig. 3.1) and provide this sum as the objective function to public domain software for numerical optimization. We use VTDIRECT (He et al., 2002) to perform a global search of parameter space, followed by ODRPACK (Boggs et al., 1989), to refine the parameter values for our model. The model equations are integrated by LSODAR (Hindmarsh, 1983), a public domain software package that efficiently solves stiff and nonstiff systems of ODEs. The VTDIRECT, ODRPACK, and LSODAR codes were chosen, after consideration of many packages, for their suitability to this problem. VTDIRECT was chosen because it efficiently allocates CPU time between exploring parameter space and converging on good regions of parameter space. ODRPACK was chosen for its efficient handling of orthogonal distances (error in both the dependent and independent variables); a necessity for the time lags discussed later. LSODAR was chosen because the ODEs here are stiff and roots need to be found for the transforms described later.

Figure 3.2 shows how the code works together. For global optimization the code operates in the right hand side of Fig. 3.2 until VTDIRECT finishes. Then the code operates on the left hand side to refine the solution VTDIRECT returned. Note that the objective function is built-in to ODRPACK but has to be supplied to VTDIRECT. For this problem the ODRPACK objective function was used for VTDIRECT as well.

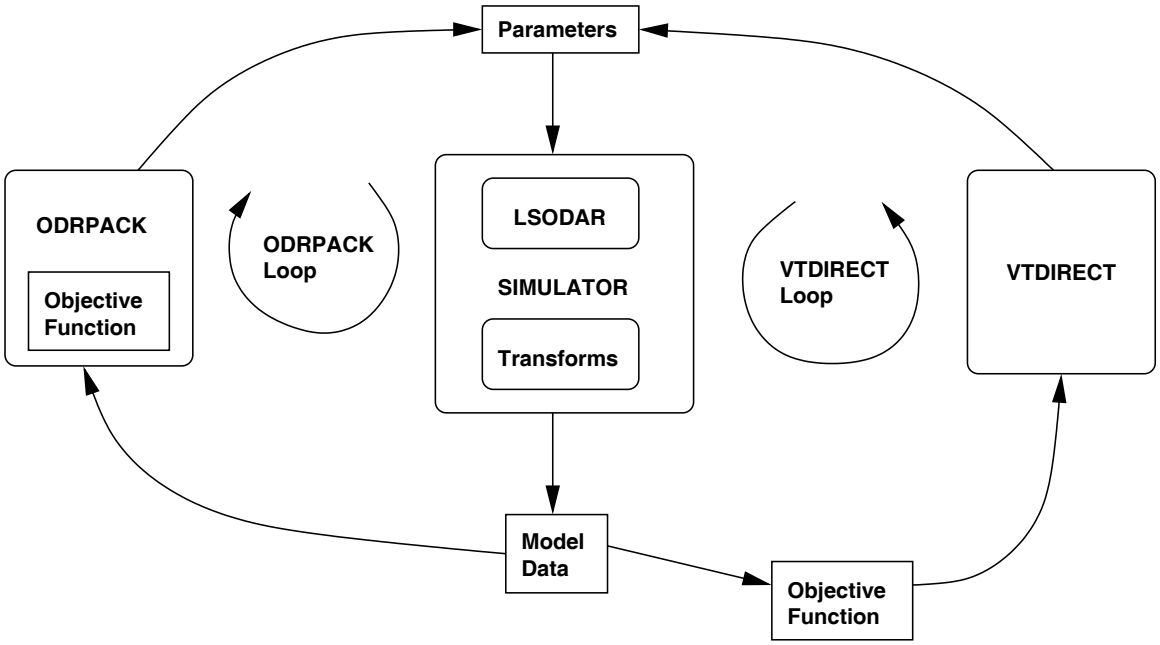

Fig. 3.2. The system architecture showing how VTDIRECT, ODRPACK, LSODAR, and the transforms interact.

3.3.1 Objective Function

In general, a model predicts the values of dependent variables (y) from independent variables (x) and parameter values (β):

$$y_i = f_i(x_i; \beta), \qquad i = 1 \ldots n,$$

where y, x, and β are (in general) all vectors and f_i is the model for the ith datum. Suppose the experimental data can be expressed by vectors y_i and x_i, $i = 1, \ldots, n$. If the model fits the data perfectly, for a specific parameter vector $\hat{\beta}$, then $y_i = f_i(x_i; \hat{\beta})$ for all i. It is always the case that there are discrepancies between the model and observations because of errors in the measurements and/or inadequacies of the model. The objective function is a scalar measure of goodness-of-fit where lower values represent a better fit of model to data. Optimizers search for parameters to minimize the objective function.

We do not assume that all measurement errors are in the dependent variables, and in fact, our model and data suggest error in the independent variable exists. For example, when the threshold for MPF activation in the model is greater than any of the time lag data points, the error in the dependent variable, time, is infinite. This occurs even if the data point is close to the timelag curve in the independent variable, total cyclin concentration.

For reactions such as this, we seek to minimize the weighted sum of squares (WSOS) of the orthogonal distances between the model and the data:

$$E_{min} = \min_{\beta,\delta} \left(\sum_{i=1}^{n} w_{\epsilon_i} \epsilon_i^2 + w_{\delta_i} \delta_i^2 \right), \tag{3.4}$$

subject to the constraints

$$\epsilon_i = f_i(x_i + \delta_i; \beta) - y_i, \qquad i = 1, \ldots, n. \tag{3.5}$$

In Eq. (3.4), ϵ_i and δ_i are the residuals for the dependent and independent variables, respectively, and w_{δ_i} and w_{ϵ_i} are weights for the errors. The weights are used to translate all observables into a common "currency" and to express the user's confidence in the reliability of different observations. The solution of problem (3.4)–(3.5) consists of $\tilde{\beta}$, $\tilde{\delta}$, $\tilde{\epsilon}$, E_{min}, giving the optimal parameter vector, the minimal discrepancies between model and observations, and a scalar measure of the overall goodness-of-fit. Convergence to the minimum solution $\tilde{\beta}$, $\tilde{\delta}$, $\tilde{\epsilon}$, and E_{min} is achieved by adjusting β and δ, where δ is treated like β—as an independent unknown.

We suggest weights of the form

$$w_{\epsilon_i} = \frac{\alpha_i}{1 + y_i^2}, \qquad w_{\delta_i} = \frac{\alpha_i}{1 + x_i^2},$$

where α_i is a dimensionless number that assigns a relative importance to the ith data vector. The terms x_i^2 and y_i^2 in the denominator make the squared residuals relative and the addition of 1 ensures that experimental data close to zero will not increase the weights to unreasonably high values. In our case, we choose $\alpha_i = 1$ for all data except the thresholds. For the thresholds we choose $\alpha_i = 3$ because these two data points are actually estimated from many separate observations, and thus should be given more significance in the optimization procedure.

3.3.2 VTDIRECT

VTDIRECT [(He et al., 2002), (He et al., 2004), (Watson and Baker, 2001)] implements the DIRECT algorithm described in Jones et al. (1993) and outlined here. The algorithm divides the search space (a p-dimensional box, assuming β is a p-vector) into boxes and systematically subdivides the boxes in search of regions of parameter space where the objective function values are small. The algorithm is deterministic, globally convergent, and (in a certain sense) computationally efficient. VTDIRECT calls a user-supplied objective function to evaluate points in the search space. Only these function evaluations are used; VTDIRECT does not use derivative information. Multiple points in parameter space

are returned as potentially optimal with one point having the best value of the objective function.

The algorithm operates on a list of boxes and objective function values calculated at the center of each box. Initially there is a single box, encompassing the region of parameter space to be searched. In the main loop, the algorithm decides which boxes from its list to subdivide, divides them into smaller boxes, evaluates E at the center of each new box, adds the new boxes to its list, and repeats. (Note that δ and ϵ are not optimized by VTDIRECT. Instead, at each point β_i that VTDIRECT requests, the orthogonal distance (δ_i, ϵ_i) is calculated using Levenberg-Marquardt, and the WSOS is returned.) Fig. 3.3 illustrates how VTDIRECT might proceed on a two-dimensional parameter space after initialization, one iteration, five iterations, and ten iterations.

VTDIRECT selects boxes to be divided from its list of boxes by computing the convex hull on the lower right envelope of a scatter plot of function values versus box diameters; all boxes on the convex hull are divided. Figure 3.4A shows a possible scatter plot of boxes and the convex hull that VTDIRECT would compute with this scatter plot. The box diameter used in this plot is the length of the longest line that fits in the box (the line goes from a corner through the box center and to the opposite corner). VTDIRECT has a parameter ϵ that controls the bias to wards exploration or convergence. ϵ's effects can be seen graphically in Fig. 3.4B. Implementation aside, the role of ϵ can be understood as a fictitious box B^* inserted into the list of boxes with a diameter of 0 and a function value of $E^* := E_{min} - \epsilon|E_{min}|$, where E_{min} is the smallest function value evaluated so far. Although B^* will never be divided, it is used in the convex hull and may cause small boxes to be passed up for division on the current iteration.

The main loop continues until one of these stopping criteria are met: iteration limit, function evaluation limit, minimum box diameter tolerance, or relative change in E_{min} tolerance. VTDIRECT then returns the box B_{min} containing the parameter vector β_{min} associated with E_{min} as the best parameter values thus far. Optionally, more parameter vectors can be returned with a user-supplied minimum separation ρ. These additional vectors are selected by removing all boxes within the minimum separation distance from B_{min}, including B_{min}, then returning the next best box. Then, repeat by removing boxes within the minimum separation distance of the next best box and return a new next best box until all boxes have been removed from the box list.

We performed a few trial runs to see how fast VTDIRECT converges for our particular problem. We used these results to determine good parameters for VTDIRECT, considering that local optimization by ODRPACK would be run on each of the points returned by VTDIRECT. We used 0.001 as the tolerance for the relative change in E_{min} and left the other stopping criteria to their defaults (no limits). We used 0.01 for ϵ and 1/3 for ρ, the minimum separation relative to the initial box diameter. For the objective function, we used the same WSOS function (above) for both VTDIRECT and ODRPACK. Later we modified ϵ, ρ, and the function evaluation limit while trying to rediscover the $\beta_{\text{LocalOnly}}$ point (see the Results section).

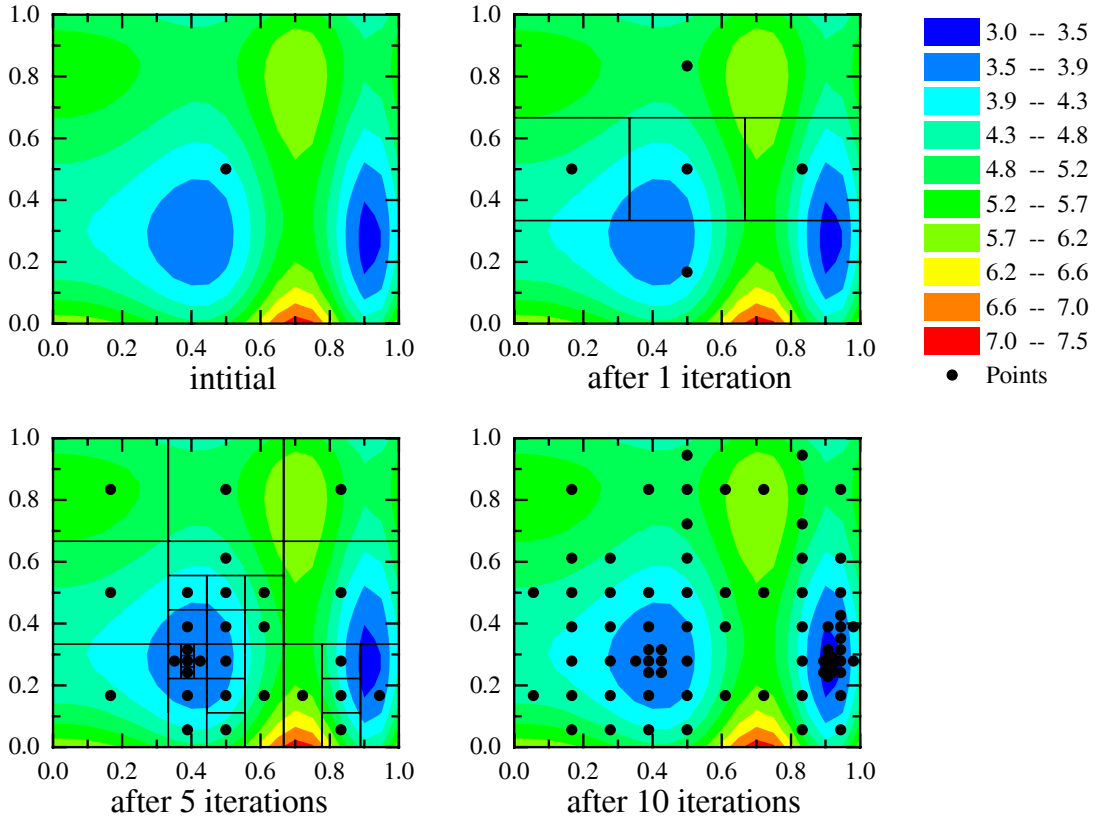

Fig. 3.3. A pictorial example of rectangle divisions made by VTDIRECT for a simple 2-dimensional example problem from Watson and Baker (2001). VTDIRECT quickly detects the local minimum after 5 iterations. After 10 iterations, VTDIRECT has refocused its efforts on the global minimum. Notice that VTDIRECT spends more time requesting function evaluations where the objective function is smaller. For this reason, VTDIRECT will tend to form "clusters" of function evaluations around local minima as well as the global minimum.

3.3.3 ODRPACK

ODRPACK uses a trust region Levenberg-Marquardt method with scaling to minimize E (Boggs et al., 1992). In doing so, ODRPACK needs to calculate Jacobian matrices (partial derivatives of the weighted vector (ϵ, δ) with respect to β and δ). ODRPACK can calculate these matrices by forward differences, centered differences, or by a user-supplied routine. Forward differences were used here. The default tolerances and scaling were used, and the maximum number of iterations was set to 10,000. The ODRPACK code is described in detail in Boggs, Byrd, and Schnabel (1987) and summarized for this problem in Zwolak et al. (2004).

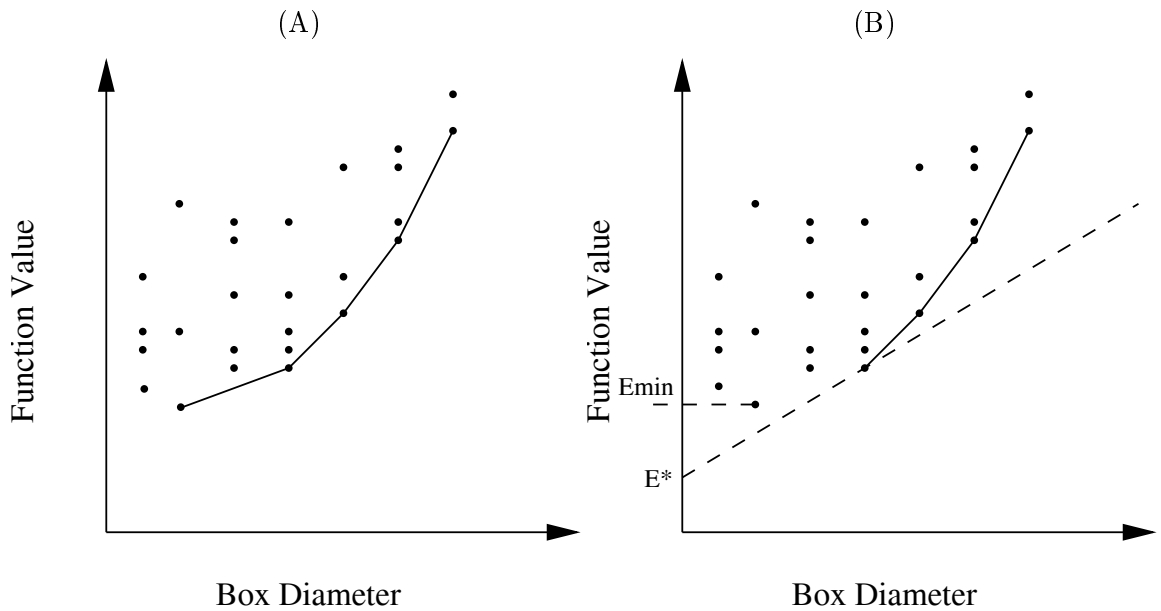

Fig. 3.4. These scatter plots represent a possible collection of boxes that VTDIRECT may have after some iterations. The objective function value E at the center of each box is plotted versus the box diameter (length of the longest line the box can contain). The points on the solid black line represent the boxes VTDIRECT will divide in the current iteration. (A) shows VTDIRECT's default behavior, which is to compute the convex hull on the lower right envelope of the scatter plot. (B) shows VTDIRECT's behavior when ϵ is set to a positive value. E^* is $E_{min} - \epsilon|E_{min}|$.

3.3.4 LSODAR

All solutions of the ODEs (3.1)–(3.3) were computed by LSODAR, a variant of LSODE ((Radhakrishnan and Hindmarsh, 1993), (Hindmarsh, 1980), (Hindmarsh, 1983)), which automatically switches between stiff and nonstiff methods and has a root finder. LSODAR starts with a nonstiff method and switches to a stiff method if necessary. LSODAR's root finder is used in this application to find the time lag for MPF activation. For nonstiff problems, LSODAR uses Adams-Moulton (AM) of orders 1 to 12. For stiff problems, LSODAR uses backward differentiation formulas (BDF) of orders 1 to 5. With both methods, LSODAR varies the step size and order. LSODAR switches from AM to BDF when AM is no longer stable for the problem or cannot meet the accuracy requirements efficiently (Petzold, 1983). The root finder in LSODAR is based on ZEROIN (Shampine and Allen, 1973). ZEROIN is based on code by Dekker (1969). LSODAR detects a root when the sign changes for the user-defined subroutine GEX. The tolerances are set to 10^{-10} for both relative and absolute error. A tolerance of 10^{-10} is used when calculating a root for a function of the form $M(t) - M_{root}$, where M_{root} is the value of the function $M(t)$ for which a time, t, is desired.

3.4. RESULTS

3.4.1 Global optimization of the Xenopus model

In Chapter 2 (Zwolak et al., 2004), we estimated best-fitting parameter values for the Xenopus model (called "the frog egg model" there) by local optimization, using the Marlovits et al. (1998) parameters as a starting point for ODRPACK. In this chapter we report on global optimization over a parameter range passed to VTDIRECT. The results of global optimization became starting points for ODRPACK to refine the parameter estimates. This strategy uncovered several local minima and the "global" minimum. The global minimum we found is different from the local minimum close to the Marlovits et al. (1998) starting point. The value of the global minimum is slightly smaller (by 9%) than the value of the local minimum found in Zwolak et al. (2004). The parameters found by global optimization suggest a mathematical simplification of the model, and we recomputed the global minimum for the simplified model.

Table 3.2 contains the WSOS for all the initial and final points used in the parameter estimation. A baseline run was performed with $\beta_{\text{Marlovits}}$ as the initial parameters using local optimization and the computation converged to $\beta_{\text{LocalOnly}}$, which is similar to results from Zwolak et al. (2004), but not identical, because we are using different weights in this chapter. Global parameter estimation was run with the initial range in Table 3.3 and a relative minimum separation of 1/3, and VTDIRECT found four points in parameter space satisfying the minimum separation. Those four points were then given to the local optimizer (ODRPACK) to be refined and yielded the parameter vectors labelled β_{Global}, β_{Global2}, β_{Global3}, and β_{Global4} in Table 3.2. Notice that β_{Global} is the optimal parameter set (that is it has the smallest objective function value) for the Xenopus model, where $\beta_{\text{Global2}}, \beta_{\text{Global3}}$, and β_{Global4} are locally optimal, globally suboptimal solutions.

Table 3.2. Weighted sum of squares (WSOS) of the residuals for local parameter estimation with various starting points from the global optimizer. Also included is the Marlovits et al. (1998) initial guess.

Point	WSOS
$\beta_{\text{Marlovits}}$	0.60014
$\beta_{\text{LocalOnly}}$	0.0356919
β_{Global}	0.03173179
β_{Global2}	0.2515503
β_{Global3}	0.0316594
β_{Global4}	0.14134885
$\beta_{\text{GlobalContinued}}$	0.03165119
β_{Simple}	0.0316514
β_{Simple2}	0.2516655
β_{Simple3}	0.0316593

β_{Global2} and β_{Global4} were thrown out because of their high WSOS compared to β_{Global} and β_{Global3}. β_{Global} and β_{Global3} have similar parameter values as can be seen in Table 3.1. These points in parameter space were explored further.

3.4.2 Simplification of the Xenopus model

The points β_{Global} and β_{Global3} differ considerably from $\beta_{\text{LocalOnly}}$ in that the parameters K_{md} and v_d are much larger in the "global" vectors than in the "local only" vector. These two parameters characterize the Michaelis-Menten function used to describe the kinetics of Cdc25 phosphorylation by active MPF. The value of K_{md} is more than five times larger than the maximum concentration of Cdc25 (scaled to be 1). Hence, substrate concentration is always $\ll K_{md}$, and the enzyme (MPF) is operating in the linear range of its kinetic rate law. From a mathematical perspective,

$$\frac{v_d M(1-D)}{K_{md} + (1-D)}$$

becomes

$$\frac{v_d M(1-D)}{K_{md}}$$

for $K_{md} \gg (1-D)$, and then the ratio v_d/K_{md} can be simplified to a single constant v_{md}. Figure 3.5 demonstrates that this simplification of the rate law is entirely justified for values of inactive Cdc25 concentration in its operational range, 0–1.

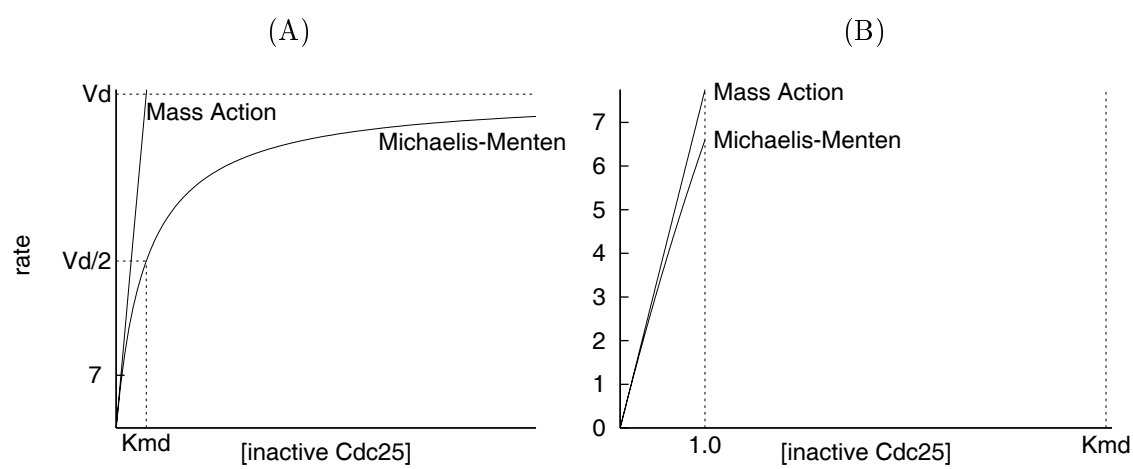

Fig. 3.5. The reaction rate term containing K_{md} from Eq. (3.2) versus inactive Cdc25 using the point β_{Global} from Table 3.1. (A) shows the rate of mass action kinetics and Michaelis-Menten kinetics to a range 0–80 for [inactive Cdc25]. (B) shows a blow up of (A) for values of [inactive Cdc25] from 0 to K_{md}. Here, the difference between mass action and Michaelis-Menten kinetics is insignificant for valid values of [inactive Cdc25], from 0 to 1.

The new model is

$$\frac{dM}{dt} = \big(v_d'(1-D) + v_d''D\big)(C_T - M) - \big(v_w'(1-W) + v_w''W\big)M, \tag{3.6}$$

$$\frac{dD}{dt} = v_{md}M(1-D) - \frac{v_{dr}D}{K_{mdr}+D}, \tag{3.7}$$

$$\frac{dW}{dt} = v_w\left(-\frac{MW}{K_{mw}+W} + \frac{\rho_w(1-W)}{K_{mwr}+(1-W)}\right), \tag{3.8}$$

with the following new parameters:

v_{md} — mass action rate constant, replacing $v_d/(K_{md}+(1-D))$,
v_{dr} — dephosphorylation of Cdc25 by PPase, replacing $v_d\rho_d$.

After this simplification, the points β_{Global} and β_{Global3} are seen to be the same solution: the values of (v_{md}, v_{dr}) are $(7.75, 0.0074)$ and $(7.36, 0.0079)$ for β_{Global} and β_{Global3}, respectively.

The reader may be interested to know why the Michaelis-Menten rate law was used to begin with. The Michaelis-Menten rate law used in the forward and reverse reactions for Cdc25 gives a switch like behavior with active MPF at the controls of that switch. When active MPF concentration reaches a critical point (in this model, the point is ρ_d), Cdc25 is "switched on", that is, it becomes active. This behavior is necessary for MPF to activate after a time lag. The Michaelis-Menten rate law was also used because it is known that MPF activates Cdc25 and inactivates Wee1. This, however, only justifies the use of Michaelis-Menten kinetics for one direction on Cdc25 and Wee1. The other direction was included to "sharpen" the switch. The replacement of Michaelis-Menten kinetics for Cdc25 activation with mass action kinetics makes Cdc25 gradually activate as MPF moves through the switch instead of sharply activating. The time lag for MPF activation still exists because of the switch behavior from the Michaelis-Menten kinetics in Wee1.

Global and local parameter estimation was then performed with the new equations and parameters. First, local optimization was run with β_{Global} as the starting point. The local optimization yielded a point close to the starting point. The WSOS of this point can be seen in Table 3.2 as $\beta_{\text{GlobalContinued}}$. Second, global optimization was run with the same ranges specified in Table 3.3 and returned multiple points with relative minimum separation of 1/3, each of which was then used as the starting point for local optimization. The WSOS of the three points returned are in Table 3.2 labeled β_{Simple}, β_{Simple2}, and β_{Simple3}. β_{Simple2} was discarded for its high WSOS. β_{Simple} and β_{Simple3} are the same point (which can be a result of VTDIRECT returning two points in the same basin of attraction resulting in ODRPACK converging on the same point). β_{Simple}, recorded in Table 3.1, is similar to β_{Global} and β_{Global3}.

Table 3.3. Ranges of parameters used for VTDIRECT while globally searching parameter space. The upper bounds were picked conservatively in case a better fit far from the Marlovits initial parameters exists.

Parameter	Lower	Upper
v'_d	0	1
v''_d	0	10
v'''_d	0	1
v'_w	0	1
v''_w	0	10
v'''_w	0	1
K_{md}	0	10
K_{mdr}	0	100
K_{mw}	0	10
K_{mwr}	0	100
v_d	0	100
v_w	0	100
μ	1	10

3.4.3 The basin of attraction of $\beta_{\text{LocalOnly}}$

We explored the parameter space further in an attempt to rediscover the solution $\beta_{\text{LocalOnly}}$. The search space was cut by a factor greater than 2^{13}, by reducing the upper bound on each parameter by about half, the ϵ parameter to VTDIRECT was increased giving more weight to wards exploration, and the stopping criterion was set to 10,000 function evaluations. Six points were returned with a relative minimum separation of 1/5 (which is conveniently 1/2 the distance between β_{Global} and $\beta_{\text{LocalOnly}}$). Of these points, two were closer to $\beta_{\text{LocalOnly}}$ than to β_{Global}. One point was an absolute distance of 15 from $\beta_{\text{LocalOnly}}$ and 33 from β_{Global}, and the other was 30 from $\beta_{\text{LocalOnly}}$ and 40 from β_{Global}. Only the first of the two was evaluated further. Local optimization was run and it converged to β_{Global}. We could not find the point $\beta_{\text{LocalOnly}}$ with global optimization even though we used our knowledge of the location of $\beta_{\text{LocalOnly}}$ in our search. None of the points returned by the global optimizer followed by the local optimizer were close to $\beta_{\text{LocalOnly}}$. VTDIRECT explored tens of thousands of points and we expected some of the better points to fall in the basin of attraction of $\beta_{\text{LocalOnly}}$. The best points fell into the basin of attraction of β_{Global}. After removing all points around β_{Global} within the minimum separation distance, the next best point fell into the basin of attraction of β_{Global2}, and similarly, the next best points fell into the basins of attraction of β_{Global3} and β_{Global4}.

Why did our attempts to find $\beta_{\text{LocalOnly}}$ fail? What properties of the objective function around $\beta_{\text{LocalOnly}}$ could explain our results? Possibly, the basin of attraction of $\beta_{\text{LocalOnly}}$ is

near one of the other basins and small enough to fit within the minimum separation distance. Perhaps, the $\beta_{\text{LocalOnly}}$ basin is not small but is mostly shallow and has a steep drop off near $\beta_{\text{LocalOnly}}$. Perhaps, the $\beta_{\text{LocalOnly}}$ basin is very small, in which case it may have been missed altogether by VTDIRECT. Or perhaps, the best points of the $\beta_{\text{LocalOnly}}$ basin fall within the minimum separation distance of the point returned by the global optimizer for β_{Global}, and the points outside that minimum separation distance are considerably higher than other points outside that distance.

We counted the number of points VTDIRECT evaluated around β_{Global} and around $\beta_{\text{LocalOnly}}$ within a ball of radius one half the distance between β_{Global} and $\beta_{\text{LocalOnly}}$. About 700 out of the 10,000 points from the last global optimization fell within the ball around $\beta_{\text{LocalOnly}}$, and about 3000 fell within the ball around β_{Global}. VTDIRECT was dividing boxes more heavily around β_{Global} than $\beta_{\text{LocalOnly}}$. This means there are more promising boxes near β_{Global} than $\beta_{\text{LocalOnly}}$. We can infer that the parameter space around β_{Global} has a larger range with good values of the objective function since more boxes were divided. Although not directly correlated, we conjecture that the basin of attraction for β_{Global} is in some sense larger and/or deeper than the basin of $\beta_{\text{LocalOnly}}$. $\beta_{\text{LocalOnly}}$ was obtained using $\beta_{\text{Marlovits}}$ as the starting point to ODRPACK. In support of this conjecture, we point out that $\beta_{\text{LocalOnly}}$ and $\beta_{\text{Marlovits}}$ are a distance of 3.7 apart, whereas $\beta_{\text{LocalOnly}}$ and its closest point returned by VTDIRECT are a distance of 15 apart. By comparison convergence to β_{Global} was obtained from points as far away as 80.

Figure 3.6 provides some views of the objective function over slices of the parameter space between several locally optimal points. The objective function was calculated in four different planes in parameter space specified by three points each. The points used are $\beta_{\text{LocalOnly}}$, β_{Global}, β_{Global2}, and β_{Global4}. (β_{Global3} is not used because it is similar to β_{Global}.) Figures 3.6A and 3.6B show that $\beta_{\text{LocalOnly}}$ and β_{Global} are in a valley together; there is not much of an increase in the objective function between them as compared to next to them (to wards β_{Global2} and β_{Global4}). Furthermore, the basin of attraction of β_{Global} seems larger than that of $\beta_{\text{LocalOnly}}$ (the objective function values around β_{Global} are smaller than that of $\beta_{\text{LocalOnly}}$ given the same radius around them). Although inconclusive by itself, Fig. 3.6 provides evidence that β_{Global} has a larger basin of attraction with better objective function values than $\beta_{\text{LocalOnly}}$. These properties would bias VTDIRECT and ODRPACK to wards β_{Global}, as we have seen in our parameter estimation runs.

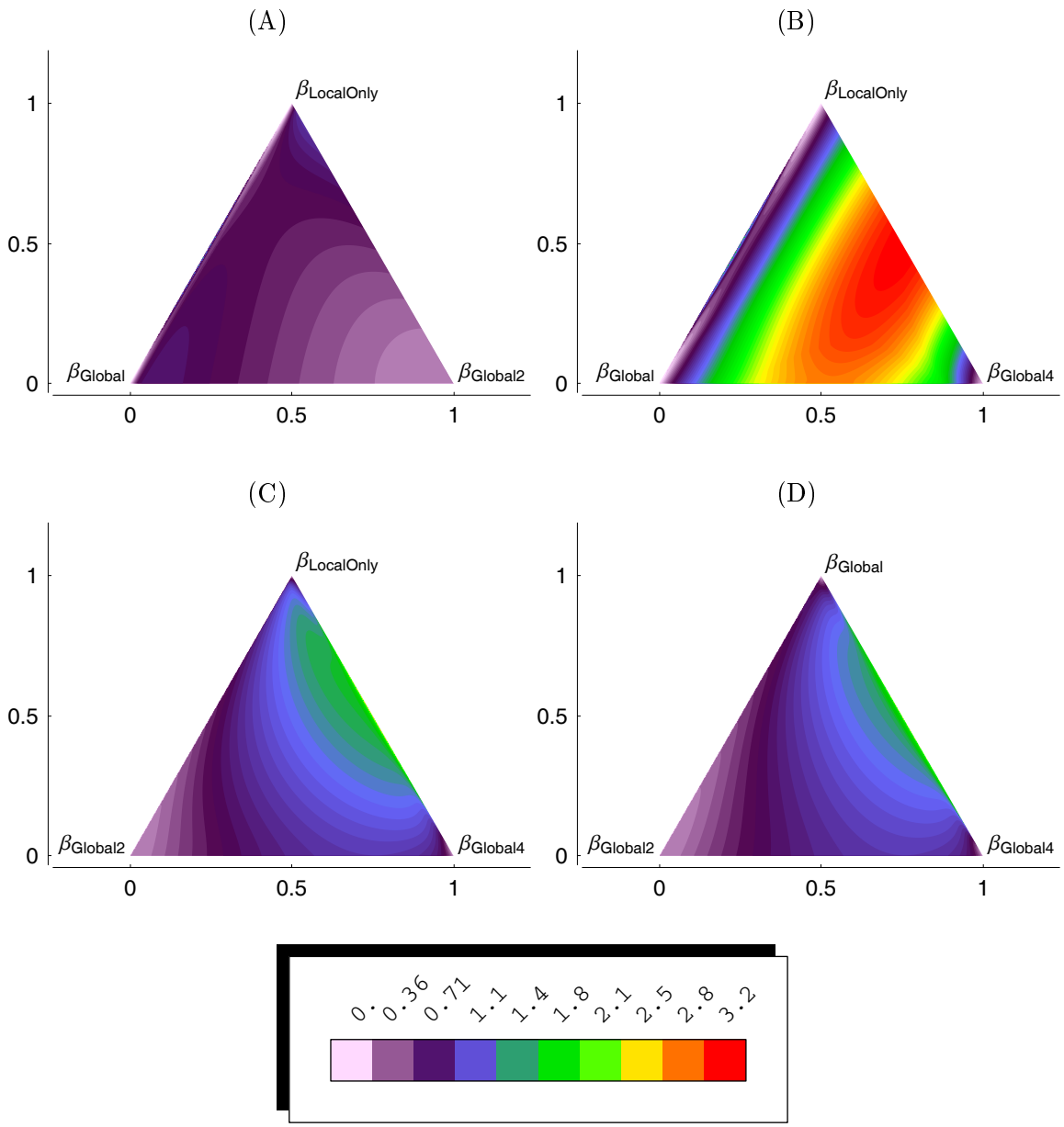

Fig. 3.6. Contour plots of two-dimensional cuts in parameter space between four groups of three points each. All the unique points returned by the global followed by local optimization are used from the first run of VTDIRECT. These plots provide hints why β_{Global} is repeatedly reached by our optimizers while $\beta_{\text{LocalOnly}}$ is reached only with local optimization and $\beta_{\text{Marlovits}}$ as the starting point.

3.5. DISCUSSION

The "bottom-up" approach to systems biology attempts to build accurate and realistic mathematical models of the molecular machinery underlying a certain aspect of cell physiol-

ogy. These models contain many kinetic parameters (rate constants, binding constants, etc.) that must be estimated by comparing model simulations to experimental measurements. Although this procedure of estimating the kinetic parameters from the very experiments that the model is trying to explain is often criticized as being circular reasoning or vacuous curve-fitting ("with four parameters I can fit an elephant"), the fact is that all kinetic parameters are ultimately estimated, in one way or another, by fitting the consequences of kinetic rate laws to experimental data. The question is not whether the parameters are estimated from the data or not, but whether we have sufficient experimental observations both to estimate the parameters and to provide meaningful tests of the mechanism. Novak and Tyson, and their colleagues, have been studying these issues for many years on a model of DNA synthesis and nuclear division in frog eggs and frog egg extracts. In Round One, Novak and Tyson (Novak and Tyson, 1993) proposed the model and estimated the kinetic constants from a consideration only of very general qualitative features of the control system (steady states, oscillations, thresholds, etc.). They made three qualitative predictions about the control system ("hysteresis", "slowing down", and "checkpoint elevation") that were confirmed ten years later [Sha et al. (2003), Pomerening et al. (2003)]. In addition, in the mid 1990s, there appeared several biochemical studies ((Kumagai and Dunphy, 1992), (Kumagai and Dunphy, 1995), (Moore, 1997), and (Tang et al., 1993)) of rates of component reactions in the Novak-Tyson mechanism. Although these experiments were not done to test the model, their results were in surprisingly good quantitative agreement with the Novak-Tyson estimates of the kinetic constants, as shown by Marlovits et al. (1998). The latter authors did the sort of back-of-an-envelope calculations familiar to biophysical chemists, but did not try to fit the model rigorously to the data, to estimate optimal parameter values, or to characterize how deviations from the optimum affect the goodness of fit.

We addressed these issues in Zwolak et al. (2004), using a local optimization algorithm (Dennis and Schnabel, 1983) with the Marlovits parameter set ($\beta_{\text{Marlovits}}$) as a starting point. We found a locally optimal parameter set $\beta_{\text{LocalOnly}}$, close to $\beta_{\text{Marlovits}}$. Sampling the objective function $E(\beta)$, close to $\beta_{\text{LocalOnly}}$, we found the expected bowl-shape with some parameter combinations tightly constrained by the data and other combinations much less constrained.

In this chapter we address the question whether the parameter set $\beta_{\text{LocalOnly}}$ is a globally optimal solution of the Xenopus model. To this end, we used a deterministic, global optimization procedure ("dividing rectangles") to explore a large region of parameter space, encompassing what we consider to be all possible reasonable values of the model's kinetic constants. The global optimizer efficiently explores this domain and returns "promising regions" of parameter space (where $E(\beta)$ is small and/or β is sufficiently far away from other promising regions). The global optimizer is not efficient at homing in on optimal points, so each promising region is studied further by the local optimizer. This procedure identified three local minima of the objective function, at points we call β_{Global}, β_{Global2},

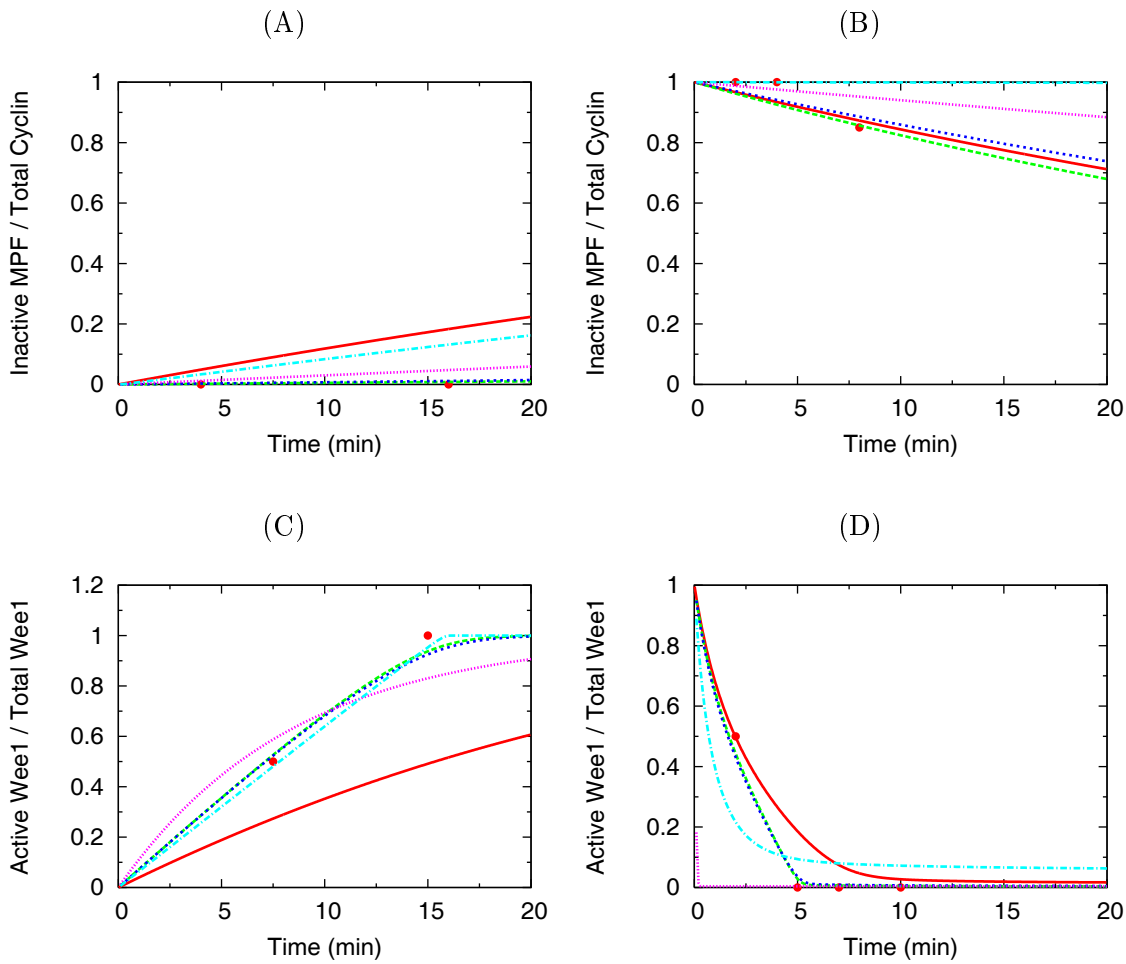

Fig. 3.7. Four plots showing simulations using β_{Global2} (------) and β_{Global4} (-·-·-) in comparison with $\beta_{\text{Marlovits}}$ (——), $\beta_{\text{LocalOnly}}$ (– – –), and β_{Global} (- - -) from Fig. 3.1. The experimental data is plotted with •. These plots represent the cases where β_{Global2} and β_{Global4} fail to accurately reproduce experimental data at an acceptable level for further analysis (as was performed on β_{Global}). The plots not shown give equally good fits for β_{Global2} and β_{Global4} as is seen with β_{Global} in Fig. 3.1.

and β_{Global4}. (β_{Global3} was deemed to be indistinguishable from β_{Global}.) These three local minimum points are all different from each other and from $\beta_{\text{LocalOnly}}$. At these points the objective function takes on the following values: $E(\beta_{\text{Global}}) = 0.032$, $E(\beta_{\text{LocalOnly}}) = 0.036$, $E(\beta_{\text{Global4}}) = 0.14$, $E(\beta_{\text{Global2}}) = 0.25$. The two best solutions, β_{Global}, and $\beta_{\text{LocalOnly}}$, give equally good fits to the data (Fig. 3.1), whereas the solutions β_{Global4} and β_{Global2} are less satisfactory (Fig. 3.7).

The major difference between β_{Global} and $\beta_{\text{LocalOnly}}$ is reflected in the kinetic rate law used to describe one step in the reaction mechanism. $\beta_{\text{LocalOnly}}$ treats this particular

MPF-catalyzed reaction as a Michaelis-Menten rate law that is noticeably saturated in the operational range of substrate concentrations, whereas β_{Global} treats the reaction as operating in the linear range of substrate concentrations. This observation suggested that the model be simplified, replacing the Michaelis-Menten rate law (two kinetic constants) by a mass-action rate law (one kinetic constant). The optimal parameter set for the simplified model we call β_{Simple}.

In Table 3.2 we have italicized those values of the kinetic constants shared by β_{Simple} and $\beta_{\text{LocalOnly}}$. These italicized rate constants are essentially identical, given the experimental uncertainty of the data used to estimate them. In this light, we do not have two different "optimal" parameter vectors, but only one parameter vector and two slightly different models. In one model, MPF-catalyzed phosphorylation of Cdc25 is depicted as a Michaelis-Menten-type reaction, and in the other model it is described by a simpler mass-action rate law. In all other aspects, the two models are in complete agreement about rate laws and kinetic constants.

CHAPTER 4: Finding Steady State Solutions

4.1. Introduction

The rate of change of concentration x_i of a chemical involved in a number of reactions in a well-stirred solution is described by an ordinary differential equation of the form $dx_i/dt = \sum_j v_{ij} r_j$, where v_{ij} is the stoichiometric coefficient of species i in reaction j ($v_{ij} > 0$ for products and $v_{ij} < 0$ for reactants) and r_j is the rate of the jth reaction. These rates are typically polynomial functions (mass-action kinetics) or rational functions (Michaelis-Menten or Hill functions) of the concentrations of the chemical species taking part in a reaction. A steady state solution of the rate equations is a constant vector $(x_1^*, x_2^*, \ldots, x_n^*)$ that satisfies the system of algebraic equations $\sum v_{ij} r_j = 0$, $i = 1, \ldots, n$. For a chemical (or biochemical) reaction system, the goal here is to find all feasible steady state solutions ($x_i^* \geq 0$ for all i). Identification of steady states is usually the first step in a qualitative analysis of the kinetic properties of a reaction network. After finding all steady states, one typically analyzes their stability properties and then tracks the steady states as parameters are varied, looking for changes in stability (bifurcations). To get a complete picture of the qualitative dynamics of a nonlinear reaction network, it is crucial to know, at the start, all feasible steady states. This chapter provides a computational method that is guaranteed to find all steady state solutions of a set of chemical reaction equations, provided every rate law is a polynomial function or rational function. In this case, the steady state equation can be transformed into a system of polynomial equations as proposed by Meintjes and Morgan (1985) in 1985 and many others since then. Probability-one homotopy theory provides a practical algorithm to find all roots of polynomial systems; the homotopy algorithm used here differs considerably from that in Meintjes and Morgan (1985).

Section 4.2 describes a simple example of a biochemical reaction network (for control of the cell cycle in frog eggs) that is used to illustrate the method. The structure of the ODE right hand sides (required for the methods described in this chapter) is described in Section 4.3. The algorithm for finding all the steady states is described in Section 4.4. The claim of finding all steady states in a model, guaranteed, relies on some technical assumptions about the ODE right hand side that are generically true, and a mathematical software package POLSYS_PLP (Wise et al., 2000) (Section 4.5). Parallel methods for this problem are discussed in Section 4.6. Some example partitions are given in Section 4.7 along with the number of curves that must be tracked for each partition. The results from the example problem and the conclusion of this chapter are in Section 4.8 and Section 4.9, respectively.

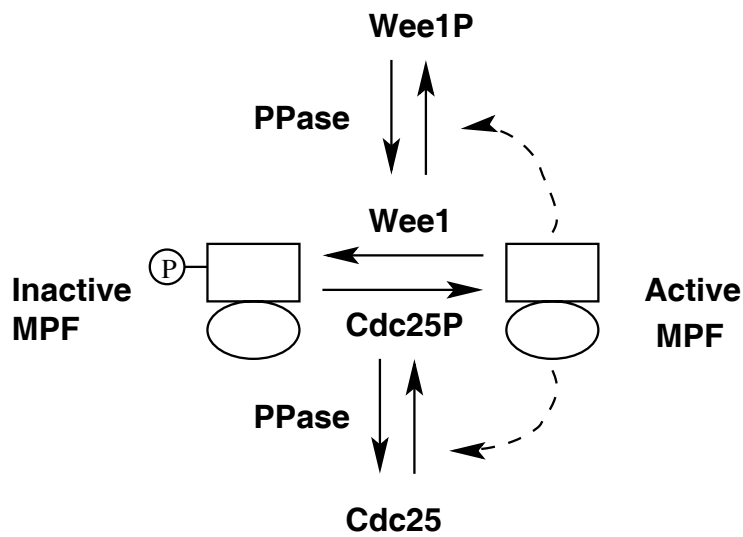

Fig. 4.1. The network of proteins for the example problem used in this chapter. The example problem is a simplified version of the model presented in Marlovits et al. (1998). In this network, active MPF phosphorylates Wee1 and Cdc25. There is an enzyme, PPase, that dephosphorylates Wee1 and Cdc25 (the name of the actual enzyme that catalyzes the dephosphorylation reactions is not important; the enzyme concentration is assumed to be constant). Wee1 phosphorylates MPF causing MPF to become inactive. Cdc25 dephosphorylates MPF, causing MPF to become active.

4.2. Example ODE model

The example problem is a simplified version of the cell cycle model presented in Marlovits et al. (1998). The network of proteins can be seen in Fig. 4.1. This is a simple model of the MPF activity in frog egg extracts. The system of ODEs for this model are

$$\frac{dM}{dt} = v_d'(1-D)(C_T - M) + v_d'' D(C_T - M) - v_w'(1-W)M + v_w'' WM, \tag{4.1}$$

$$\frac{dD}{dt} = \frac{v_d M(1-D)}{K_{md} + (1-D)} - \frac{v_{dr} D}{K_{mdr} + D}, \tag{4.2}$$

$$\frac{dW}{dt} = -\frac{v_w MW}{K_{mw} + W} + \frac{v_{wr}(1-W)}{K_{mwr} + (1-W)}, \tag{4.3}$$

where M is active MPF, C_T is total cyclin, $C_T - M$ is inactive MPF, D is active Cdc25 (phosphorylated), $1-D$ is inactive Cdc25, W is active Wee1 (not phosphorylated), and $1-W$ is inactive Wee1. The vs and Ks are rate constants and Michaelis constants, respectively. Note that the reverse enzyme in the D and W equations does not appear. This enzyme is assumed to have a constant value and is absorbed into the rate constants v_{dr} and v_{wr}.

4.3. Structure of ODE right hand side

The algorithm described in this chapter will only work with systems of ODEs

$$\frac{dx}{dt} = H(x)$$

of a certain form. Ultimately the steady state problem $H(x) = 0$ must be transformed into a system of polynomials. For each system $H(x) = 0$, there must be an algorithm to convert the right hand sides (RHS) to polynomial equations. One such algorithm is described in Section 4.4 and works on problems of the nature described below. The restriction imposed in Section 4.4 is that each component $H_i(x)$ of the right hand side of the system of ODEs be a linear combination of rational functions.

Most biological models of molecular networks have linear combinations of rational functions for the right hand side of their system of ODEs. In fact, the right hand sides are usually even more restricted to mass action and Michaelis-Menten kinetics. Mass action kinetics have the form

$$k \cdot S_1 \cdot S_2 \cdots S_n,$$

where k is a rate constant (parameter) and S_i represents the concentration of a protein (variable). Michaelis-Menten kinetics have the form

$$\frac{k \cdot S \cdot E}{K_m + S},$$

where k is a rate constant (parameter), K_m is a Michaelis constant (parameter), S is the substrate concentration (variable), and E is the enzyme concentration (variable). Clearly these kinetics (illustrated in (4.1)–(4.3)) are rational functions.

4.4. Algorithm

The algorithm presented here covers the type of kinetics described in the previous section. The steady state system must first be converted to a system of polynomials. The system of polynomials can then be given to a homotopy algorithm (POLSYS_PLP), which finds *all* zeros of the system. The roots must be checked in the steady state equations to verify that they are not extraneous roots introduced by the conversion to a system of polynomials. The verified steady states constitute *all* steady states of the ODE, assuming there are finitely many steady states. The case of infinitely many steady states is a degenerate situation, and is not considered further here.

These steps are now illustrated on the example model. First, set up the steady state equations by setting the right hand side of the ODE equal to zero. The example problem yields

$$0 = v_d'(1-D)(C_T - M) + v_d'' D(C_T - M) \tag{4.4}$$
$$- v_w'(1-W)M + v_w'' WM,$$
$$0 = \frac{v_d M(1-D)}{K_{md} + (1-D)} - \frac{v_{dr} D}{K_{mdr} + D}, \tag{4.5}$$
$$0 = -\frac{v_w MW}{K_{mw} + W} + \frac{v_{wr}(1-W)}{K_{mwr} + (1-W)}. \tag{4.6}$$

Second, multiply each equation by a constructed polynomial to eliminate the denominators. Note that the steady states are preserved by this transformation. Multiplying through by a polynomial may add extraneous zeros to the system, but they can be eliminated by substituting into the original steady state equations. The constructed polynomials for the example problem are

$$1, \tag{4.7}$$
$$(K_{md} + (1-D))(K_{mdr} + D), \tag{4.8}$$
$$(K_{mw} + W)(K_{mwr} + (1-W)). \tag{4.9}$$

Multiplying the right hand sides by these polynomials yields the polynomial system of equations

$$0 = v_d'(1-D)(C_T - M) + v_d'' D(C_T - M) \tag{4.10}$$
$$- v_w'(1-W)M + v_w'' WM,$$
$$0 = v_d M(1-D)(K_{mdr} + D) \tag{4.11}$$
$$- v_{dr} D(K_{md} + (1-D)),$$
$$0 = -v_w MW(K_{mwr} + (1-W)) \tag{4.12}$$
$$+ v_{wr}(1-W)(K_{mw} + W).$$

A globally convergent, probability-one homotopy algorithm (implemented in POLSYS_PLP) is then applied to find all the zeros of (4.10)–(4.12), among which the steady states of the ODE (4.1)–(4.3) lie. The theoretical justification for the claim that *all* the zeros are found is given in the next section.

Finally, it is necessary to verify which zeros of the polynomial system are steady states of the model. All the zeros of the polynomial system are checked by substituting them into the steady state equations.

Pseudocode for finding all the steady states follows:

begin

read ODE system right hand side $H = (H_1, \ldots, H_n)^t$;

for $i := 1$ to n **do**

$L_i := H_i *$LCM(denominators of terms comprising H_i);

done

apply POLSYS_PLP to the polynomial system $L(x) = 0$;

check whether each zero z returned by POLSYS_PLP satisfies $H(z) = 0$;

end

4.5. POLSYS_PLP

The mathematical software package POLSYS_PLP makes this work possible and deserves some explanation. POLSYS_PLP efficiently finds all zeros of a system of polynomials (assumed to have finitely many zeros, the nondegenerate case). The zeros are found by tracking zero curves of a homotopy map from a start system to a target system. The start system is another system of polynomials with easily found zeros and a structure similar to the target system. This section describes in some detail the method implemented by the code POLSYS_PLP as described in Wise et al. (2000) and cites theorems (also from Wise et al. (2000)) showing that all the zeros can be reliably found.

Polynomial systems of equations have the form

$$F_i(z) = \sum_{j=1}^{n_i} [c_{ij} \prod_{k=1}^{n} z_k^{d_{ijk}}] = 0, \quad i = 1, \ldots, n, \tag{4.13}$$

where the c_{ij} are complex, and the d_{ijk} are nonnegative integers. The degree of $F_i(z)$ is

$$d_i = \max_{1 \le j \le n_i} \sum_{k=1}^{n} d_{ijk},$$

and the total degree of the system (4.13) is

$$d = \prod_{i=1}^{n} d_i. \tag{4.14}$$

Following Wise et al. (2000), define a homotopy map $\rho : [0, 1) \times \mathbf{C}^n \to \mathbf{C}^n$ by

$$\rho(\lambda, z) = (1 - \lambda)G(z) + \lambda F(z). \tag{4.15}$$

$\lambda \in [0,1)$ is the homotopy parameter, $G(z) = 0$ is the start system, and $F(z) = 0$ is the target system. The start system is constructed with a partitioned linear product (PLP) structure, explained next.

PLP structure refers to a particular way of constructing the start system, and matching it to the target system. Following Wise et al. (2000), let $P = (P_1, P_2, \ldots, P_n)$ be an n-tuple of partitions P_i of the set $\{z_1, z_2, \ldots, z_n\}$. That is, for $i = 1, 2, \ldots, n$, $P_i = \{S_{i1}, S_{i2}, \ldots, S_{im_i}\}$, where S_{ij} has cardinality $n_{ij} \neq 0$, $\cup_{j=1}^{m_i} S_{ij} = \{z_1, z_2, \ldots, z_n\}$, and $S_{ij_1} \cap S_{ij_2} = \emptyset$ for $j_1 \neq j_2$. For $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m_i$ define d_{ij} to be the degree of the component F_i in only the variables of the set S_{ij}, that is, considering the variables of $\{z_1, z_2, \ldots, z_n\} \backslash S_{ij}$ as constants. For convenience of defining the start system, rename the variables z_k such that $S_{ij} = \{z_{ij1}, z_{ij2}, \ldots, z_{ijn_{ij}}\}$. The start system is represented mathematically by $G_i(z) = \prod_{j=1}^{m_i} G_{ij}$, where

$$G_{ij} = \begin{cases} \left(\sum_{k=1}^{n_{ij}} c_{ijk} z_{ijk} \right)^{d_{ij}} - 1, & \text{if } d_{ij} > 0; \\ 1, & \text{if } d_{ij} = 0, \end{cases} \tag{4.16}$$

for $i = 1, 2, \ldots, n$, where the numbers $c_{ijk} \in \mathbf{C}_0 = \mathbf{C} \backslash \{0\}$ are chosen at random. Note that all the start system's zeros can be easily found by solving systems of linear equations (Wise et al., 2000).

THEOREM 1 (MORGAN ET AL., 1995). Let $f : \mathbf{C}^n \to \mathbf{C}^n$ be a system of polynomials and $U \subset \mathbf{C}^n$ be (Zariski) open. Define $N(f, U)$ to be the number of nonsingular solutions to $f = 0$ that are in U. Assume that there are positive integers $r_1, \ldots, r_n$ and $m_1, \ldots, m_n$ and finitely generated complex vector spaces V_{ij} of polynomials for $i = 1, \ldots, n$ and $j = 1, \ldots, m_i$ such that

$$f_i = \sum_{k=1}^{r_i} \prod_{j=1}^{m_i} p_{ijk}, \tag{4.17}$$

where $p_{ijk} \in V_{ij}$ for $i = 1, \ldots, n$, $j = 1, \ldots, m_i$ and $k = 1, \ldots, r_i$. Let a system g be defined by $g_i = \prod_{j=1}^{m_i} g_{ij}$, with each g_{ij} a generic choice from V_{ij}. Then

$$N(f, U) \leq N(g, U), \tag{4.18}$$

and (4.18) is equality if, for each i with $1 \leq i \leq n$, there is a positive integer k_i such that the $p_{ijk_i} \in V_{ij}$ are generic for $j = 1, \ldots, m_i$. Also, $g(z) = 0$ is a suitable start system for the polynomial homotopy $\rho(\lambda, z) = \lambda f(z) + (1 - \lambda) g(z)$ to find all nonsingular solutions to $f(z) = 0$.

THEOREM 2 (WISE ET AL., 2000). For almost all choices of c_{ijk} in the start system defined by (4.16), $\rho^{-1}(0)$ consists of B_{PLP} smooth, disjoint, nonbifurcating curves emanating from $\{0\} \times \mathbf{C}^n$, which either diverge to infinity as λ approaches 1 or converge to solutions of $F(z) = 0$. Each nonsingular solution of $F(z) = 0$ will have a curve converging to it.

The number B_{PLP} of curves mentioned in Theorem 2 is called the partitioned linear product Bezout number, and by Theorem 1, is an upper bound on the number of (nonsingular, isolated) zeros of $F(z)$. Thus to find *all* the isolated roots of $F(z) = 0$, it suffices to track at most B_{PLP} homotopy zero curves, a finite process. The PLP Bezout number B_{PLP} can be computed (nontrivially) from (4.16) by enumerating all the zeros of $G(z)$ (see Wise et al. (2000) for full details).

If the homotopy zero curve tracking is done in complex projective space $\mathbf{P}^n$ (a compact set) rather than $\mathbf{C}^n$, then in fact none of the curves in Theorem 2 go to infinity — all curves lie in a compact set. This is what is done in POLSYS_PLP; all B_{PLP} curves, being finite in $\mathbf{P}^n$, are tracked in $\mathbf{P}^n$ completely to $\lambda = 1$. Solutions in $\mathbf{P}^n$ that do not exist in $\mathbf{C}^n$ (are at infinity, so to speak) are discarded as being nonphysical.

For the homotopy path tracking and the parallelized POLSYS_PLP (described in Section 4.6), it is important to note that the curves do not cross or bifurcate (Wise et al., 2000). Figure 4.2 shows an example of what the homotopy zero curves (the set $\rho^{-1}(0)$) for a polynomial system might look like. There are several algorithms for tracking these zero curves: ODE based, normal flow, and augmented Jacobian matrix [(Watson et al., 1987),(Watson et al., 1997)]; the POLSYS_PLP algorithm is a normal flow algorithm (Watson et al., 1997). All the numerical algorithms assume that the curves are smooth, nonintersecting, and do not bifurcate. From Theorem 2, these properties hold with probability one. Note that the curves in Fig. 4.2 do not have turning points; they are monotone with respect to λ. They do not intersect and they do not bifurcate. The curves can merge at $\lambda = 1$ if there are multiplicities in the target system where there were none in the start system (there is such an example in Fig. 4.2). All these properties of the zero curves just mentioned, and illustrated in Fig. 4.2, essentially always hold (precisely, they hold with probability one).

4.6. Parallelism

The previous section explained how POLSYS_PLP finds all the roots of polynomial systems. For even small systems, many curves must be tracked because B_{PLP} increases rapidly (possibly exponentially) with respect to the number of polynomials. For the example problem, 18 curves were tracked, but for a problem with just 3 more similar equations there could potentially be 324 curves to track. Some problems have as many as 20 equations; with potentially 3.48×10^9 curves. These numbers come from the Bezout numbers of the systems of polynomials. For the example problem there are two equations of degree 3 and one of

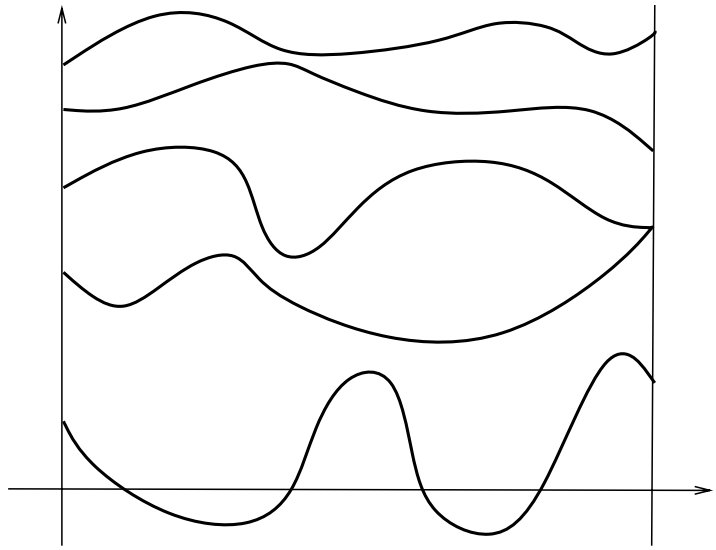

Fig. 4.2. An example of the zero set $\rho^{-1}(0)$ of a homotopy map $\rho(\lambda, x)$ for a polynomial system.

degree 2. Multiplying all the degrees gives 18 (the classical Bezout number) paths to track without exploiting structure. For the hypothetical problem with 20 equations each equation is assumed to have degree 3 (a reasonable number for kinetic equations). Note that for larger problems exploiting the structure of the system is important to greatly reduce the number of curves to be tracked.

The zero curves of the homotopy map are independent and therefore can be tracked independently. This is a perfectly parallelizable problem. However, the parallelism could be coarse grained (track the curves concurrently) or fine grained (parallelize the tracking of each individual curve).

Both methods for parallelizing POLSYS (a precursor of the code POLSYS_PLP) have been tried [(Allison et al., 1989),(Morgan and Watson, 1989)]. Tracking the curves independently on multiple processors decreases the run time significantly. The overhead for communication is low, but the efficiency may not always be high. Some curves have higher costs to track than others. There is no way to know (a priori) how long a curve will take to track. With this knowledge, one must explore the possibility of parallelizing the tracking of individual curves. This will keep the most CPUs busy the longest. However, results show that the overhead for the fine grained method is significant, and can dominate the calculation time. The parallelism occurs in the evaluation of the polynomial system and its Jacobian matrix. These calculations are quick, and do not overcome the costs of communication (Allison et al., 1989).

Allison et al. (1989) argues that the best method for parallelizing a code like POLSYS_PLP is the master-slave model where the master gives the slaves curves to track. The

slaves report the curve tracking results back to the master. Following Allison et al. (1989), pseudocode for a possible parallel POLSYS_PLP code follows:

FOR THE HOST (MASTER):

1. Initialize the data space and calculate a starting point for each path.
2. SEND problem data and a starting point to a node.
3. If another path needs to be assigned and a node is available, go to 2.
4. Now wait for a message from a node.
5. RECEIVE a "ready to transmit solution" message from a node (call it the "current" node).
6. SEND an acknowledgment ("ready to receive" message) to the current node.
7. If a "ready to transmit" message is received from another node, put the node identification into a queue until the current node completes transmitting a solution.
8. RECEIVE a "solution" message from the current node and print it.
9. If another path needs to be assigned, SEND problem data and a starting point to the current node.
10. If any nodes are in the queue (see 7), remove the first node from the queue, call it the current node and go to 6.
11. If awaiting messages from any other nodes, go to 4; otherwise stop.

FOR EACH NODE (SLAVE):

1. RECEIVE problem data and a starting point from the host.
2. Track the path associated with the starting point.
3. SEND a "ready to transmit solution" message to the host.
4. RECEIVE a "ready to receive" message from the host.
5. SEND the "solution" message to the host and go to 1.

4.7. PLP structure

Parallelizing the code alone will at best reduce the computation time by a factor proportional to the number of machines used. By exploiting the structure of the problem, the B_{PLP} number can be reduced exponentially. This section gives a new example problem and three partitions for the new example problem. The B_{PLP} number for each partition is given, and there is a significant difference between the B_{PLP} numbers for each partition. This difference shows that exploiting structure is important even if a parallel algorithm is used.

The simple cell cycle example problem used throughout this chapter cannot show the advantage of structure exploitation because the problem is small and leaves little room for reduction in B_{PLP}. A new larger example problem, not based on a real model, is

$$0 = x_1x_2 - x_1x_3, \tag{4.19}$$

$$0 = x_2x_3x_4 - x_5, \tag{4.20}$$

$$0 = x_5x_4 - x_5x_4^2 + x_1x_4 - x_1x_4^2, \tag{4.21}$$

$$0 = x_6x_5 - x_6x_5^2 + x_1x_5 - x_1x_5^2, \tag{4.22}$$

$$0 = x_2x_4x_6 + x_4x_5 - x_1, \tag{4.23}$$

$$0 = x_2x_6 - x_2x_6^2 + x_1x_6 - x_1x_6^2. \tag{4.24}$$

Conceivably, this could come from a real biological or chemical kinetic model.

Three partitions are shown here for the example problem: a 1-homogeneous partition (no structure), a m-homogeneous partition (identical structure for each equation), and a general PLP partition. The general PLP partition is based on the structure of the polynomial system and tries to put variables that do not occur together in the same set. The 1-homogeneous partition, ignoring all structure, is $\{\{x_1, x_2, x_3, x_4, x_5, \mathrm{x}_6\}\}$ for each equation in the system. This partition has a B_{PLP} number of 486. The m-homogeneous partition for $m = 6$ is $\{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}\}$ for each equation in the system. This partition has a B_{PLP} number of 88, significantly better than the 1-homogeneous partition. The general PLP partition is

$$\begin{aligned}
&\{\{x_1\}, \{x_2, x_3\}, \{x_4, x_5, x_6\}\},\\
&\{\{x_2\}, \{x_3\}, \{x_4, x_5\}, \{x_1, x_6\}\},\\
&\{\{x_1, x_5\}, \{x_4\}, \{x_2, x_3, x_6\}\},\\
&\{\{x_1, x_6\}, \{x_5\}, \{x_2, x_3, x_4\}\},\\
&\{\{x_1, x_5, x_6\}, \{x_2\}, \{x_4\}, \{x_3\}\},\\
&\{\{x_1, x_2\}, \{x_6\}, \{x_3, x_4, x_5\}\}.
\end{aligned}$$

This partition has a B_{PLP} number of 84. The general PLP partition is not significantly better than the m-homogeneous partition, but is significantly better than the 1-homogeneous partition.

For a given problem, a good PLP partition will usually be much better than the 1-homogeneous partition or a m-homogeneous partition. For the example problem in this section, the m-homogeneous partition (with $m = 6$) gives a good reduction of the B_{PLP} number. For some problems, a general PLP partition may be significantly better than any m-homogeneous partition and for some problems both will not give a good reduction in B_{PLP}. Exploiting the structure in large problems can reduce the number of paths significantly and on a real problem one can usually find a good partition.

4.8. Results

The results from POLSYS_PLP were checked by taking all the steady states from a few values of total cyclin and tracking the steady state curves. All the steady state values for active MPF were verified by tracking these curves using XPPAUT (Ermentrout, 2002). The graphs of the steady states found by the algorithms described in this chapter and the steady states found by XPPAUT's curve tracker are seen in Fig. 4.3. The steady states tracked by XPPAUT were placed over top of them and visually inspected for differences. The curves had no differences apparent to visual inspection. Note that XPPAUT cannot find a steady state. XPPAUT must start with a steady state, and it can then track the steady state over the range of a parameter (e.g., total cyclin).

Many roots were returned by POLSYS_PLP over a range of total cyclin values. Figure 4.3 only plots some of the roots for equation (4.10) for a range of total cyclin values. Appendix C shows all of the roots for equations (4.10)–(4.12) for a specific total cyclin value and Appendix D shows the parameter values for the roots in Appendix C. The solutions in Appendix C satisfy the polynomial system (4.10)–(4.11) in complex projective space (Wise et al., 2000). Some of the solutions are infinite when mapped to complex space and are not considered as physical solutions. Solutions 1, 2, 3, 6, 7, 8, 9, 15, 16, 17, and 18 are all considered infinite (nonphysical). Furthermore, solutions with nontrivial imaginary components (Solutions 5 and 14) or values outside the physical range of the protein concentration (e.g., negative values of protein concentrations as seen in Solutions 4 and 13) are discarded as biologically infeasible.

Parallel results for POLSYS_PLP are not reported here, because they are essentially the same as those for POLSYS, which has been thoroughly studied [(Allison et al., 1989), (Morgan and Watson, 1989), (Watson et al., 1987)]. POLSYS_PLP represents a major advance over POLSYS by using PLP structure (hence reducing significantly the number of curves that must be tracked), but any given curve is handled exactly the same by POLSYS and POLSYS_PLP. The parallel aspects of POLSYS and POLSYS_PLP are thus essentially the same.

4.9. Conclusion

An algorithm was developed for finding all steady state solutions to a restricted class of ODE based models that include common molecular network kinetics. The algorithm was tested and verified to work on a model of cell cycle control. The algorithm will next be tried on more complicated models and the steady states checked. Larger problems will require a parallel implementation POLSYS_PLP, and full exploitation of the PLP structure of the polynomial systems (no structure was exploited for the example problem here).

This is a significant new tool for theoreticians studying complex reaction networks in chemistry, biology, atmospheric science, etc.. For the first time, a feasible algorithm to determine all physically realistic steady state solutions of a model exists.

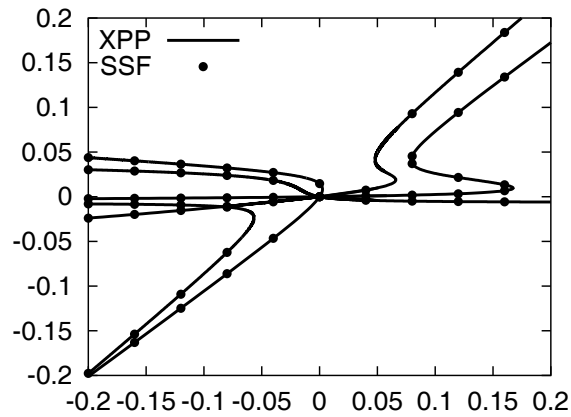

Fig. 4.3. The steady states of active MPF versus the parameter total cyclin from the example problem. Although negative values of active MPF and total cyclin are not physically feasible, they are plotted here for completeness since they are mathematical solutions. The steady states were first found with the algorithms described in this chapter (SSF, steady state finder). Some of the steady states were picked and entered into XPP. The steady state curves were tracked in XPP and are also plotted here.

CHAPTER 5: ODRPACK95

5.1. INTRODUCTION

Least squares is arguably the most common method for fitting data to a model when there are errors in the observations. For example, suppose we have data pairs $(x_i, y_i),\ i = 1, \ldots, n$ where x_i is the independent variable and y_i is the dependent variable and suppose that x_i and y_i are related by a smooth, possibly nonlinear function f, i.e.,

$$y_i = f(x_i; \beta), \tag{5.1.1}$$

where $\beta \in \mathcal{R}^p$ is a set of parameters to be determined. Equation (5.1.1) is meant to imply that if there are no errors in either x_i or y_i and if β is known exactly, then (5.1.1) holds exactly. If there are errors in the data, then the true value of β can only be approximately obtained, unless the number of observations goes to infinity.

In classical least squares, it is assumed that x_i is known exactly and y_i is observed with error. Although it is often the case that x_i have errors, these errors can be safely ignored if they are much smaller than the corresponding errors in y_i. Thus, if we take the error in y_i to be given by ϵ_i, we can write

$$y_i = f(x_i; \beta) + \epsilon_i \tag{5.1.2}$$

and approximate β by solving the classical, or ordinary least squares problem given by

$$\min_{\beta} \frac{1}{2} \sum_{i=1}^{n} [f(x_i; \beta) - y_i]^2. \tag{OLS}$$

This, of course, can be interpreted as minimizing the sum of the squares of the vertical distances from the data points to the curve $f(x; \beta)$.

If, however, the errors in x_i cannot be ignored and we designate the error in x_i by δ_i, then (5.1.2) becomes

$$y_i = f(x_i + \delta_i; \beta) + \epsilon_i$$

and it is reasonable to approximate the parameter β by minimizing the sum of the squares of the orthogonal distances from the data points to the curve $f(x, \beta)$. As shown in Boggs, Byrd, and Schnabel (1987)this gives rise to the *orthogonal distance regression* problem given by

$$\min_{\beta, \delta} \frac{1}{2} \sum_{i=1}^{n} \left[(f(x_i + \delta_i; \beta) - y_i)^2 + \delta^2 \right]. \tag{ODR}$$

Note that (ODR) is easily seen to be equivalent to

$$\min_{\beta, \delta, \epsilon} \frac{1}{2} \sum_{i=1}^{n} (\epsilon_i^2 + \delta_i^2)$$

subject to the constraints

$$y_i = f(x_i + \delta_i; \beta) + \epsilon_i \qquad i = 1, \ldots, n$$

from which it is easy to see that we are, indeed, minimizing the sum of the squares of the orthogonal distances.

A numerically stable and efficient algorithm for solving (ODR) is given in Boggs, Byrd, and Schnabel (1987) and a detailed implementation, called ODRPACK, that provides a number of practical options and statistical output is given in Algorithm 676 (Boggs et al., 1989). In Boggs, Byrd, and Schnabel (1987), the authors show that the work per iteration for their algorithm for solving (ODR) problem is exactly the same as the work per iteration for solving (OLS). An enhancement of ODRPACK is available from Netlib (Dongarra and Grosse, 1987). This is a Fortran 77 implementation that includes the ability to handle a general weighting scheme, allows x to be multidimensional, and contains a version to allow the data to be complex. This code has been downloaded and used many times by scientists, engineers, and practitioners around the world; it is described in several textbooks, including Björck (1996)and Dongarra and Grosse (1987). (ODR) has important statistical applications; in the statistical literature it often goes by the name "errors in variables" (see, e.g., Fuller (1987)).

Over the years, there have been occasional requests to implement a version of ODRPACK that allows explicit bounds on the values of β, but this was not done. The general form of the bound-constrained (ODR) problem can be expressed as

$$\min_{\beta,\delta} \frac{1}{2} \sum_{i=1}^{n} \left[(f(x_i + \delta_i; \beta))^2 + \delta_i^2 \right] \qquad (BC - ODR)$$

subject to the constraints

$$l \leq \beta \leq u, \qquad (BC - ODR)$$

where l and u are vectors of length p that provide the lower and upper bounds on β, respectively. ODRPACK has some features that could be used to solve a bound-constrained problem, but the resulting algorithm is not efficient.

This chapter has two goals. First, it addresses the issue of modifying the ODRPACK algorithm to handle bounds efficiently and, second, it updates ODRPACK by rewriting much of it in Fortran 95. The resulting code, called ODRPACK95 is thus much simpler to use because it takes advantage of Fortran 95 to do dynamic memory management and to allow easier passing of parameters.

The chapter is organized as follows. In Section 5.2 we review briefly ODRPACK and the algorithm given in Boggs, Byrd, and Schnabel (1987). We also review some of the features of ODRPACK that could be used to handle bounds and show why these are not efficient. Then, is Section 5.3, we give our modifications to the algorithm to be able to handle these bounds efficiently. We conclude with a discussion of the testing that we have done in Sections 5.5 and 5.6. Section 5.7 describes basic usage of ODRPACK95.

5.2. ODRPACK DESCRIPTION

A brief description of the ODRPACK algorithm is provided here. Weights are left out for simplicity and can be added at the expense of complexity and bookkeeping, but require no fundamental changes in the algorithm and pseudocode described here. ODRPACK is based on a trust region Levenberg-Marquardt algorithm with scaling and numeric or analytic derivatives and is described in detail in Boggs, Byrd, and Schnabel (1987), Boggs et al. (1989), and Boggs et al. (1992).

Building on the BC-ODR problem described earlier, the partials of the errors ϵ and δ are taken with respect to the parameters β and δ to give the Jacobian matrix

$$J = \begin{bmatrix} \frac{\partial \epsilon_1}{\partial \beta_1} & \cdots & \frac{\partial \epsilon_1}{\partial \beta_p} & \frac{\partial \epsilon_1}{\partial \delta_1} & \cdots & \frac{\partial \epsilon_1}{\partial \delta_n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \epsilon_n}{\partial \beta_1} & \cdots & \frac{\partial \epsilon_n}{\partial \beta_p} & \frac{\partial \epsilon_n}{\partial \delta_1} & \cdots & \frac{\partial \epsilon_n}{\partial \delta_n} \\ \frac{\partial \delta_1}{\partial \beta_1} & \cdots & \frac{\partial \delta_1}{\partial \beta_p} & \frac{\partial \delta_1}{\partial \delta_1} & \cdots & \frac{\partial \delta_1}{\partial \delta_n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \delta_n}{\partial \beta_1} & \cdots & \frac{\partial \delta_n}{\partial \beta_p} & \frac{\partial \delta_n}{\partial \delta_1} & \cdots & \frac{\partial \delta_n}{\partial \delta_n} \end{bmatrix}.$$

Note that the appearance of δ in both the errors and the parameters gives a special and exploitable structure to the Jacobian matrix. The Jacobian matrix can be divided into four quadrants based on the combinations of ϵ and δ with β and δ, where all but the upper-left quadrant contain special properties. For convenience the quadrants are labelled

$$J = \begin{bmatrix} G & V \\ Z & D \end{bmatrix}.$$

G is the Jacobian matrix of ϵ with respect to β and has no special properties. V is the Jacobian matrix of ϵ with respect to δ and has the special property of a being a diagonal matrix. This property exists because each ϵ_i depends only on δ_i, see Eq. (5.1.3). Z is the Jacobian matrix of δ with respect to β and is all zeros because δ is an independent variable. Finally, for a similar reason, D is a diagonal matrix of constants representing the Jacobian of δ with respect to δ. This realization and exploitation of the structure of the ODR problem makes ODRPACK efficient; the time-complexity with respect to n is reduced from quadratic to linear.

The following pseudocode gives an overview of the algorithm.

do until parameters converge, sum of squares converge, or iteration limit reached;

$G_{ij} = \partial f_i(x_i + \delta_i; \beta)/\partial \beta_j$, $i = 1, \ldots, n$, $j = 1, \ldots, p$;

$V = \mathrm{diag}\{\partial f_i(x_i + \delta_i; \beta)/\partial \delta_i,\ i = 1, \ldots, n\}$;

$D = \mathrm{diag}\{1\}$: Note here that in the weighted problem this diagonal matrix would be a constant function of the weights.

$P = V^T V + D^2$: P is defined here to make the following equations simpler.

Formulate the linear least squares problem (derived from the linearization of E)

$$\min_s \left\| \left((I - VP^{-1}V^T)^{\frac{1}{2}} Gs \right) - (I - VP^{-1}V^T)^{-\frac{1}{2}} \left(-\epsilon + VP^{-1}(V^T\epsilon + D\delta) \right) \right\|^2,$$

and solve for s with a QR factorization of the coefficient matrix of s. Note that Boggs, Byrd, and Schnabel (1987) realized the ODR problem can be solved efficiently this way instead of solving the normal equations with the full Jacobian matrix J.

$t = -P^{-1}(V^T\epsilon + D\delta + V^T Gs)$;

Use s and t to update β and δ, respectively. The Levenberg-Marquardt method starts with the steepest decent method and smoothly changes to the Gauss-Newton method, where s and t are simply added to β and δ, as the solution is approached. ODRPACK uses a trust region implementation of the Levenberg-Marquardt method which reduces the step size based on the confidence in a model of the objective function. See Moré and Wright (1993) for details on how parameters are updated in the Levenberg-Marquardt algorithm.

end do

ODRPACK has a primitive method for handling invalid parameters. The user-supplied subroutine that calculates f can indicate invalid parameters by returning a flag to ODRPACK. ODRPACK then reverts to the last successful point in parameter space and reduces the step size until f can be evaluated. In the case where a user wishes parameters to be bounded, this approach may cause ODRPACK to stall near a bound even though a valid direction still exists that reduces E. In this case, a user can continue optimization using an active set strategy, keeping parameters in the active set fixed at their respective boundaries. This requires that ODRPACK be restarted with a new active set every time a parameter hits a boundary.

ODRPACK's exploitation of the structure of the Jacobian matrix makes it an efficient algorithm for the unconstrained weighted orthogonal nonlinear least squares problem. However, for problems where β is physically constrained and interior barrier functions are physically inappropriate, ODRPACK's handling of invalid parameters can be debilitating (Zwolak et al., 2004). The next section describes the changes to ODRPACK, motivated by the need to directly support simple bound constraints.

5.3. DIFFERENCES BETWEEN ODRPACK95 AND ODRPACK

The ODRPACK95 code contains the original ODRPACK code wherever possible. Deviations, rewrites, and additions made to the original code are described in this section. All changes fall into two major categories: those required to support bounds, and those required by or made possible by the conversion to Fortran 95.

5.3.1 Bound Constraints

The most important addition made in ODRPACK95 is the support for bound constraints on the parameter vector β: $L_i \leq \beta_i \leq U_i$, $i = 1, \ldots, p$. The algorithm used for bound constraints is the same as that in LANCELOT (Conn, Gould, and Toint, 1992), which calculates the projected step so as to always maintain a feasible β. Furthermore, when a parameter value is at its bound, the corresponding numerical partial derivative will use a one-sided finite difference approximation to avoid calculations with parameters outside the bounds. Lastly, the initialization in ODRPACK95 requires function evaluations at feasible parameter values. Thus the ODRPACK initialization algorithm has been modified to only use feasible β values. ODRPACK95 never calls the user supplied function with parameters outside the user supplied bounds and the final solution is guaranteed to be feasible.

The most important change adds a restriction on β during each iteration before any function evaluations are made with β. The restriction ensures that the current function evaluation is made with feasible β. After β is updated with the step s, whose direction is a convex combination of the Gauss-Newton direction and the steepest descent direction, β will be projected into the hyperbox $[L, U]$, precisely:

$$\beta_i := \begin{cases} \beta_i, & \text{if } L_i \leq \beta_i \leq U_i, \\ L_i, & \text{if } \beta_i < L_i, \\ U_i, & \text{if } U_i < \beta_i, \end{cases} \qquad i = 1, \ldots, p.$$

Before β is updated and projected, numerical or analytic derivatives are calculated. If analytic derivatives are used, then no additional function evaluations are required; the derivatives are calculated at the current β. When forward or backward differences are used, then the step h_i must obey $L_i \leq \beta_i + h_i \leq U_i$, for $i = 1, \ldots, p$. The sign of h_i is changed if the bounds are violated, namely

$$h_i := \begin{cases} h_i, & \text{if } L_i \leq \beta_i + h_i \leq U_i; \\ -h_i, & \text{if } L_i > \beta_i + h_i \text{ or } \beta_i + h_i > U_i, \end{cases} \qquad i = 1, \ldots, p.$$

It is possible that $\beta_i + |h_i| > U_i$ and $\beta_i - |h_i| < L_i$ for some i. ODRPACK95 avoids this situation by ensuring that $U_i - L_i \geq 2|h_i|$, for all i, where $h_i > 0$ is chosen with respect to the initial β during initialization. When central differences are used, then the points where the function is evaluated are shifted together until they are both within the bounds, precisely

$$(\beta_i^-, \beta_i^+) := \begin{cases} (\beta_i - h_i, \beta_i + h_i), & \text{if } L_i \leq \beta_i - |h_i| < \beta_i + |h_i| \leq U_i, \\ (L_i, L_i + 2|h_i|), & \text{if } L_i > \beta_i - |h_i|, \\ (U_i - 2|h_i|, U_i), & \text{if } U_i < \beta_i + |h_i|. \end{cases}$$

This has the effect of shifting the points β_i^- and β_i^+ such that they are always $2h_i$ apart and within the bounds. If, for example, β_i is $0.2h_i$ from U_i (such that $\beta_i + 0.2h_i = U_i$),

then U_i and $U_i - 2h_i$ are used in the central difference formula instead of $\beta_i + h_i$ and $\beta_i - h_i$. Furthermore, if β_i is on the a bound (e.g., $\beta_i = L_i$ or $\beta_i = U_i$); then the modified central difference method used here becomes a forward or backwards differentiation formula (depending on which bound β_i lies on). Again, ODRPACK95 ensures during initialization that $U_i - L_i \geq 2|h_i|$, for $i = 1, \ldots, p$.

During initialization, function evaluations are required for derivative checking, the initial point, and prediction of the number of reliable digits. The derivative checking uses the same code as the numerical derivatives to request function evaluations, and therefore, does not evaluate the function outside the bounds. The initial point is verified to be within the bounds, and if it is not, then the code returns with an appropriate error flag. Finally, prediction of the number of reliable digits in the objective function must be made. These calculations are similar to derivative calculations (in fact, they are first and second order derivative approximations with some additional numerics that estimate how many digits are reliable in the objective function). These calculations occur centered at the initial β. To ensure that these calculations occur only at feasible points, the center point used in the calculations is minimally adjusted from the initial β to be far enough from the bounds for the calculations to succeed (in a manner similar to that of central differences described above).

With these changes, ODRPACK95 provides the same reliable and efficient optimization as ODRPACK, but with simple bound constraints. At no time will ODRPACK95 evaluate the function outside the bounds, and the final solution will be a local constrained minimum.

5.3.2 Fortran 95

ODRPACK95 conforms to the Fortran 95 specification ISO/IEC 1539-1. The code was compiled, run, and tested on a DEC Alpha, a Sun Sparc Station, and an Intel Xeon using the Digital Equipment Corporation, Sun, and Intel compilers, respectively. The code uses the Fortran 95 fixed format to minimize changes from (the FORTRAN 77 fixed format code of) ODRPACK and the likelihood of bugs introduced into the code. The conversion to Fortran 95 facilitates a number of other significant improvements in ODRPACK95.

Among those improvements are optional arguments, use of Fortran 95 modules, automatic array allocation, and use of Fortran 95 intrinsic functions for machine constants. All non-essential arguments to ODRPACK95 are optional; this makes the ODRPACK95 interface considerably simpler and allows defaults to be set when arguments are not present. The call to ODRPACK95 was simplified from

```
      CALL DODRC(
     +           FCN,
     +           N,M,NP,NQ,
     +           BETA,
     +           Y,LDY,X,LDX,
     +           WE,LDWE1,LD2WE1,WD,LDWD1,LD2WD1,
     +           IFIXB,IFIXX,LDIFX,
```

```
     +           JOB,NDIGIT,TAUFAC,
     +           SSTOL,PARTOL,MAXIT,
     +           IPRINT,LUNERR,LUNRPT,
     +           STPB,STPD,LDSTPD,
     +           SCLB,SCLD,LDSCLD,
     +           WORK,LWMIN,IWORK,LIWMIN,
     +           INFO
      )
```

to

```
      CALL ODR(
     +           FCN,
     +           N,M,NP,NQ,
     +           BETA,
     +           Y,X,
     +           LOWER=L,UPPER=U
      )
```

and there is only one interface to ODRPACK95 while ODRPACK has DODRC, DODR, SODRC, and SODR. These multiple interfaces to ODRPACK exist to allow short and long argument lists and single and double precision arithmetic. The user's calling program would contain the statement

```
USE ODRPACK95
```

giving their code access to the ODRPACK95 interface and aiding in compile time error checking. The array arguments in the interface will be automatically allocated if the user does not supply the (optional) argument or does not allocate the provided argument. The arrays will be deallocated only if the user did not provide a return argument for them. Lastly, all the calculations are done using the machine constants from the Fortran 95 intrinsics, eliminating the former need to supply a function to return machine constants. These changes all together make ODRPACK95 a substantial improvement over the original ODRPACK.

5.4. ORGANIZATION AND TESTING

The ODRPACK95 distribution contains several Fortran source files (`*.f`), a make file (`makefile`), a user's guide (`guide.ps`), a readme file (`readme`), a change log (`changes`), and some input data (`data?.dat`) for some example problems (`drive?.f`). The default make target builds ODRPACK95, compiles the example and test problems (`test.f`), and runs the example and test problems. The files containing the results of the examples and tests are named like the source files that generated them with a `.out` extension. Some BLAS/LAPACK routines are used by ODRPACK95 and are contained in `lpkbls.f` in case the user's system does not already have the BLAS/LAPACK routines installed. The user or installer must manually select usage of a local BLAS/LAPACK package or the routines in `lpkbls.f` by editing `makefile`. Lastly, the file `real_precision.f` contains a Fortran

KIND definition for the real precision to use (IEEE 64-bit arithmetic is the default; note that changing the REAL KIND will require compatible BLAS and LAPACK).

ODRPACK95 was tested thoroughly with dozens of test cases and test problems. Some of these are distributed with the code and are described here. The test cases for ODRPACK are also distributed with ODRPACK95 and are described in Boggs et al. (1992). The ODRPACK95 tests can be run with `make test.out`. The output file `test.out` contains the results of the tests and will end with "`ALL TESTS AGREE WITH EXPECTED RESULTS`" in the case that all tests passed. This is a way to ensure a properly installed and functioning ODRPACK95.

The model for the test problem new in ODRPACK95 is specially constructed to exercise many of the conditions that may arise in bound constrained optimization. The new test cases are listed below and numbered as they are seen in `test.f`.

13) Parameters start on a boundary, move away from the boundary, hit a boundary, move away from the boundary, and stop at a minimum.
14) Parameters start interior to the bounds, never hit a boundary, and stop at a minimum.
15) Parameters start interior to the bounds and stop on a boundary.
16) Parameters start outside the bounds, and ODRPACK95 returns an error flag.
17) Bounds are ill defined ($L_i > U_i$ for some i), and ODRPACK95 returns an error flag.
18) Central differences are used. Parameters start on a boundary, move away from the boundary, hit a boundary, move away from the boundary, and stop at a minimum.
19) Bounds are well defined but slightly too close for ODRPACK95 to do any calculations. An error flag is returned.
20) Bounds are well defined and slightly farther apart than the previous case. They are far enough apart to allow NDIGIT calculations but still too close for ODRPACK95 to proceed. An error flag is returned.
21) Bounds are well defined and as close as the machine allows, and therefore too close for finite differences and NDIGIT calculations. ODRPACK95 returns an error flag.

The model used for all the bound constraint test cases above is

$$f(x;\beta) = \beta_1 e^{\beta_2 * x},$$

where the global minimum point (used to generate experimental data) is $\beta = (1,1)^T$.

5.5. PERFORMANCE

The test cases documented in the previous section were also used to benchmark ODRPACK95. ODRPACK95 performs exactly as ODRPACK when $\beta+s$ remains feasible except that an additional IF-statement is executed to check that $\beta+s$ is feasible. When a boundary is crossed or reached, ODRPACK95 must execute many additional statements to ensure all function evaluations are performed with feasible parameters and that the resulting β is feasible. It is this case of active bound constraints that this section addresses.

The benchmarks were performed with a modified `test.f`. Lines were added before and after the call to `ODR` that read the time with the Fortran 95 `CPU_TIME` intrinsic function. In addition, the lower bound for Test Case 18 was modified. Without this modification, Test Case 18 is the same as Test Case 13 except that central differences are used. This modification adds to the diversity of the benchmarks. Table 5.1 shows the timing and setup for the benchmarks. After these modifications, the code was compiled, then run with the command

```
nice -n -20 ./a.out > stdout.txt
```

on a dual Xeon machine running Linux. The `nice` command ensures the benchmark gets highest priority of the running processes (no other active processes were running during the benchmark, but many system processes were sleeping or waiting for interrupts). No options for the Intel Fortran 95 compiler were used for the benchmarks.

The number of iterations for the runs (see Table 5.1) vary significantly. The variances can be attributed to ODRPACK/ODRPACK95's response to information about the objective function along the different paths through parameter space. (Since each test case took a different path through parameter space, the objective function will look different on those paths, causing ODRPACK/ODRPACK95 to behave differently.) Reaching a bound increases the number of iterations, but the work per iteration is roughly the same whether β is on a bound or not.

A notable exception is when central differences are used. The use of central differences about doubles the work per iteration compared with forward and backward differences. On a boundary, central differences are replaced by forward or backward differences because the function cannot be evaluated outside the bounds. This explains the almost double number of function evaluations per iteration seen in Test Case 18 in Table 5.1. It is always the case that use of central differences in ODRPACK95 will be cheaper per iteration when the parameters are on a boundary than when the parameters are away from all boundaries.

Table 5.1. The setup and timing results for benchmarks of ODRPACK95. Included is the test case number (Test #) as found in `test.f`; the lower and upper bounds of β (Lower and Upper); the initial β (β_0); the final β (β_{final}); the number of ODRPACK95 iterations while β was on a bound (#it$_B$) and the total number of iterations (#it$_T$); the run time measured from when `ODR` was called to when it finished (Time (ms)); number of function evaluations (#FEV); and the number of function evaluations per iteration (#FEV/#i). Case 18 was modified from `test.f` by changing its lower bound.

Test #	13	14	15	18
Lower	(0.10,0)	(0.00,0)	(1.10,0)	(0.01,0)
Upper	(200,5)	(400,6)	(400,6)	(200,5)
β_0	(200,5)	(200,5)	(200,3)	(200,5)
β_{final}	(1.0,1)	(1.0,1)	(1.1,1)	(1.0,1)
#it$_B$/#it$_T$	83/95	0/25	68/68	3/26
Time (ms)	9.765	3.907	5.859	3.907
#FEV	388	108	285	188
#FEV/#it$_T$	4.05	4.2	4.15	7.23

5.6. USAGE

The usage of ODRPACK95 is greatly simplified from that of ODRPACK. A simple example can be found in Appendix E and the output of the example in Appendix F. The most simple form of a call to ODRPACK95 with bounds is

```
      CALL ODR(
     +          FCN,
     +          N,M,NP,NQ,
     +          BETA,
     +          Y,X,
     +          LOWER=L,UPPER=U
      )
```

There are many more optional arguments detailed in the ODRPACK and ODRPACK95 users guide. The required arguments (in the call statement above) and optional bound arguments are explained here.

- `FCN` The user supplied function that evaluates the model and partial derivatives of the model.
- `N` The number of experimental data points (x_i, y_i). Experimental data can come in vector form.
- `M` The size of the vector for the independent experimental data.
- `NQ` The size of the vector for the dependent experimental data.
- `NP` The number of parameters for the model.
- `BETA` The parameters for the model used as an initial guess. The final solution (if one) is returned in this variable.

`Y` `N`×`NQ` array of the dependent experimental data.

`X` `N`×`M` array of the independent experimental data.

`LOWER` Optional array of lower bounds on `BETA`.

`UPPER` Optional array of upper bounds on `BETA`.

CHAPTER 6: Conclusion

Many scientists and engineers solve optimization problems every day. A typical problem is to fit a mathematical model to experimental data. Although in many cases such fits require only ordinary least squares regression (where all error in experimental data is attributed to a dependent variable, or errors in the measurement of data), sometimes orthogonal distance regression (where error in experimental data is attributed equally to independent and dependent variables) is more appropriate. ODRPACK is the only tool specifically designed to efficiently implement orthogonal distance regression. In many cases orthogonal distance regression must be carried out under constraints on the parameters, i.e. the requirement to keep rate constants positive. ODRPACK95 efficiently implements bound constrained orthogonal distance regression and is the only tool to do so. ODRPACK95 also provides a cleaner interface that takes advantage of the newer Fortran 95 language. Along with the other Fortran 95 features (array allocation, intrinsics for machine constants, and modules), ODRPACK95 will replace ODRPACK as the new standard for orthogonal distance regression.

ODRPACK95 is intended for users skilled in programming computers. To apply this tool and many others to problems in the life sciences, biologists must team with computer scientists to exploit the power of computers. The work described in this thesis is a result of that type of team. Theoretical biologists and computer scientists successfully came together to apply ODRPACK, VTDIRECT, and LSODAR to estimate parameters in a mathematical of a specific biological problem (cell division in frog eggs). This work has "paved the way" for more ambitious parameter estimation problems by the theoretical biologist's group at Virginia Polytechnic Institute and State University in the future.

The work already completed applies local and global optimization to fit models, by adjusting rate constants, to experimental data. This is an enormous task to do without a computer. Even with a computer, the task is still difficult unless the right tools are created and employed. The synthesis of ODRPACK, VTDIRECT, and LSODAR, along with model-specific transforms, have proven to be the right tool (although, perhaps not the only "right" tool). Parameter space was adequately explored by VTDIRECT, parameters were refined by ODRPACK, and models simulated quickly by LSODAR. The synthesis of these tools, unexpectedly, brought theoretical biologist's attention to a new model of the frog egg. These tools have great potential and will be employed in future work.

The more specific problem of finding all steady solutions in chemical kinetic models may have a lesser impact than ODRPACK95 and parameter estimation. Although finding steady state solutions in systems of ODEs is also a widespread problem for scientists and engineers, the steady state finder in this thesis only works on models limited to rational functions on the right hand sides of the ODEs. This structure can be found in biological models, and more generally, in chemical kinetic models. For these models, the steady state

finder can be used as a transform or part of a transform during parameter estimation or as a generator of steady states for bifurcation analysis. The important feature of this tool is that it finds *all* steady state solutions. This allows the theoretician to exhaustively explore the steady states solutions.

Although ODRPACK95 and the steady state finder may be useful in many areas of science and engineering, the main goal of this work has always been to help theoretical biologists model biological systems. The frog egg model used in this work was chosen for testing and developing these tools. The tools are now sufficiently mature to tackle larger problems and become more practical.

BIBLIOGRAPHY

Allen, N.A., Calzone, L., Chen, K.C., Ciliberto, A., Ramakrishnan, N., Shaffer, C.A., Sible, J.C., Tyson, J.J., Vass, M.T., Watson, L.T., and Zwolak, J.W. 2003. Modeling Regulatory Networks at Virginia Tech. *OMICS, A Journal of Integrative Biology*. 7(3), 285–299.

Allison, D.C.S., Chakraborty, A., and Watson, L.T. 1989. Granularity issues for solving polynomial systems via globally convergent algorithms on a hypercube. *J. Supercomputing*. 3, 5–20.

Arkin, A., Ross, J., and McAdams, H. H. 1998. Stochastic kinetic analysis of developmental pathway bifurcation in phage lambda-infected *Escherichia coli* cells. *Genetics*. 149, 1633–1648.

Asthagiri, A. R., and Lauffenburger, D. A. 2001. A computational study of feedback effects on signal dynamics in a mitogen-activated protein kinase (MAPK) pathway model. *Biotechnol. Prog.*. 17, 227–239.

Barkai, N., and Leibler, S. 1997. Robustness in simple biochemical networks. *Nature*. 387, 913–917.

Björck, Å. 1996. *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, PA.

Boggs, P.T., Byrd, R.H., and Schnabel, R.B. 1987. A stable and efficient algorithm for nonlinear orthogonal distance regression. *SIAM J. Sci. Stat. Comput.* 8(6), 1052–78.

Boggs, P.T., Byrd, R.H., Donaldson, J.R., and Schnabel, R.B. 1989. Algorithm 676 — ODRPACK: software for weighted orthogonal distance regression. *ACM Trans. Math. Software*. 15(4), 348–364.

Boggs, P.T., Byrd, R.H., Rogers, J.E., and Schnabel, R.B. 1992. *User's Reference Guide for ODRPACK Version 2.01: Software for Weighted Orthogonal Distance Regression*, Center for Computing and Applied Mathematics, U.S. Department of Commerce, Gaithersburg, MD.

Bray, D., Bourret, R. B., and Simon, M. I. 1993. Computer simulation of the phosphorylation cascade controlling bacterial chemotaxis. *Mol. Biol. Cell*. 4(5), 469–482.

Bray, D. 1995. Protein molecules as computational elements in living cells. *Nature*. 376, 307–312.

Brazhnik, P., de la Fuente, A., and Mendes, P. 2002. Gene networks: how to put the function in genomics. *Trends Biotechnol.*. (20), 467–72.

Brent, R. 2000. Genomic biology. *Cell*. 100, 169–183.

Carroll, R.J., Ruppert, D., and Stefanski, L.A. 1995. *Measurement Error in Nonlinear Models*, Chapman and Hall/CRC, New York, NY.

Chen, K., Csikasz-Nagy, A., Gyorffy, B., Val, J., Novak, B., and Tyson, J. J. 2000. Kinetic analysis of a molecular model of the budding yeast cell cycle. *Mol. Biol. Cell.* 11, 369–391.

Chen, K. C., Calzone, L., Csikasz-Nagy, A., Cross, F. R., Novak, B., and Tyson, J. J. 2004. Integrative Analysis of Cell Cycle Control in Budding Yeast. *Mol. Biol. Cell.* 15(8), 3841–3862.

Conn, A.R., Gould, N.I.M., and Toint, Ph.L. 1992. *LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization (Release A)*, Springer-Verlag, Berlin.

Cross, F. R., Archambault, V., Miller, M., and Klovstad, M. 2002. Testing a Mathematical Model of the Yeast Cell Cycle. *Mol. Biol. Cell.* 13(1), 52–70.

Dekker, T.J. 1969. Finding a zero by means of successive linear interpolation. In Dejon, B., and Henrici, P., eds., *Constructive Aspects of the Fundamental Theorem of Algebra*, Wiley-Interscience, London.

Dennis, J. E. Jr., and Schnabel, R. B. 1983. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Dongarra, J.J. and Grosse, E.H. 1987. Distribution of mathematical software via electronic mail. *Communications of the ACM.* 30, 403–407.

Eissing, T., Conzelmann, H., Gilles, E. D., Allgöwer, F., Bullinger, E., and Scheurich, P. 2004. Bistability analysis of a caspase activation model for receptor-induced apoptosis. *J. Biol. Chem.*. 279(35), 36892–36897.

B. Ermentrout 2002. *Simulating, Analyzing, and Animating Dynamical Systems: A Guide to XPPAUT for Researchers and Students*, SIAM, Philadelphia.

Fuller, W. A. 1987. *Measurement Error Models*, John Wiley, New York.

Gantmacher, F. R. 1977. *The Theory of Matrices*, Chelsea Publishing Company, New York, NY.

Gear, C.W. 1971. *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ.

Hanahan, D., and Weinberg, R. A. 2000. The hallmarks of cancer. *Cell.* 100, 57–70.

Hartwell, L. H., Hopfield, J. J., Leibler, S., and Murray, A. W. 1999. From molecular to modular cell biology. *Nature.* 402, C47–52.

Hasty, J., McMillen, D., Isaacs, F., and Collins, J. J. 2001. Computational studies of gene regulatory networks: in numero molecular biology. *Nat. Rev. Genet.*. 2, 268–279.

He, J., Watson, L.T., Ramakrishnan, N., Shaffer, C.A., Verstak, A., Jiang, J., Bae, K., and Tranter, W.H. 2002. Dynamic data structures for a direct search algorithm. *Comput. Optim. Appl.*. 23, 5–25.

He, J., Verstak, A., Watson, L.T., Stinson, C.A., Ramakrishnan, N., Shaffer, C.A., Rappaport, T.S., Anderson, C.R., Bae, K., Jiang, J., and Tranter, W.H. 2004. Globally optimal transmitter placement for indoor wireless communication systems. *IEEE Trans. Wireless Commun.*. in press.

Hindmarsh, A.C. 1980. LSODE and LSODI, two new initial value ordinary differential equation solvers. *ACM SIGNUM Newsletter*. 15(4), 10–11.

Hindmarsh, A.C. 1983. ODEPACK: a systematized collection of ODE solvers, 55–64. In Stepleman, R.S., et al., eds., *Scientific Computing*, North Holland Publishing Co., New York.

Hoffmann, A., Levchenko, A., Scott, M. L., and Baltimore, D. 2002. The IκB-NF-κB Signaling Module: Temporal Control and Selective Gene Activation. *Science*. 298(5596), 1241–1245.

Hynne, F., Dano, S., and Sorensen, P. G. 2001. Full-scale model of glycolysis in Saccharomyces cerevisiae. *Biophys. Chem.* 94, 121–163.

Jones, D.R., Perttunen, C.D., and Stuckman, B.E. 1993. Lipschitzian Optimization Without the Lipschitz Constant. *Journal of Optimization Theory and Applications*. 79(1), 157–181.

Kahaner, D., Moler, C., and Nash, S. 1989. *Numerical Methods and Software*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Kohn, K. W. 1999. Molecular interaction map of the mammalian cell cycle control and DNA repair systems. *Mol. Biol. Cell.* 10, 2703–2734.

Kumagai, A., and Dunphy, W. G. 1992. Regulation of the cdc25 protein during the cell cycle in Xenopus extracts. *Cell.* 70, 139–151.

Kumagai, A., and Dunphy, W. G. 1995. Control of the Cdc2/Cyclin B complex in Xenopus egg extracts arrested at a G2/M checkpoint with DNA synthesis inhibitors. *Mol. Bio. of the Cell.* 6, 199–213.

Kumar S. P, and Feidler, J. C. 2003. BioSPICE: a computational infrastructure for integrative biology. *OMICS.* (7), 225.

Lazebnik, Y. 2002. Can a biologist fix a radio?— Or, what I learned while studying apoptosis. *Cancer Cell.* (2), 179–82.

Marlovits, G., Tyson, C.J., Novak, B., and Tyson, J.J. 1998. Modeling M-phase control in Xenopus oocyte extracts: the surveillance mechanism for unreplicated DNA. *Biophys. Chem.* 72, 169–184.

Martiel, J. L., and Goldbeter, A. 1987. A model based on receptor desensitization for cyclic-AMP signaling in Dictyostelium cells. *Biophys. J.*. 52, 808–828.

McAdams, H. H., and Shapiro, L. 1995. Circuit simulation of genetic networks. *Science.* 269, 650–656.

Meinhardt, H., and de Boer, P. A. 2001. Pattern formation in *Escherichia coli*: a model for the pole-to-pole oscillation of MIN proteins and the localization of the division site. *Proc. Natl. Acad. Sci. USA.* 98, 14202-7.

Meintjes, K. and Morgan, A. P. 1985. A methodology for solving chemical equilibrium systems. *Tech Rep. GMR-4971, General Motors Research Lab.*. Warren, MI.

Mendes, P. 1993. GEPASI: A software package for modelling the dynamics, steady states and control of biochemical and other systems. *Comput. Applic. Biosci.*. 9, 563–571.

Moles, C. G., Mendes, P., and Banga, J. R. 2003. Parameter estimation in biochemical pathways: a comparison of global optimization methods. *Genome Res.* 13, in press.

Moore, J. 1997. Private Communication, Aug.

Moré, J.J., and Wright, S.J. 1993. *Optimization Software Guide*, SIAM, Philadelphia, PA.

Morgan, A. P. and Watson, L.T. 1989. A globally convergent parallel algorithm for zeros of polynomial systems. *Nonlinear Analysis, Theory, Methods Appl.*. 13(11), 1339–1350.

Morgan, A.P., Sommese, A.J., and Wampler, C.W. 1995. A product-decomposition bound for Bezout numbers. *SIAM J. Numer. Anal.*. 32(4), 1308–1325.

Murray, A. and Hunt, T. 1993. *The Cell Cycle*, W. H. Freeman and Company, New York, NY.

Nocedal, J., and Wright, S.J. 1999. *Numerical Optimization*, Springer-Verlag, New York.

Novak, B. and Tyson, J. 1993. Numerical analysis of a comprehensive model of M-phase control in Xenopus oocyte extracts and intact embryos. *Journal of Cell Science*. 106, 1153–68.

Petzold, L. 1983. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM J. Sci. Stat. Comput.* 4, 136–148.

Pomerening, J. R., Sontag, E. D., and Ferrell, J. E. 2003. Building a cell cycle oscillator: hysteresis and bistability in the activation of Cdc2. *Nature Cell Biology*. 5, 346–351.

Radhakrishnan, K., and Hindmarsh, A.C. 1993. *Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations*, NASA Reference Publication 1327. Lawrence Livermore National Laboratory, Livermore, CA, Dec.

Sha, W., Moore, J., Chen, K., Lassaletta, A. D., Yi, C-S., Tyson, J. J., and Sible, J. C. 2003. Hysteresis drives cell-cycle transitions in Xenopus laevis egg extracts. *Proc. Nat. Acad. Sci. USA*. 100, 975–980.

Shampine, L.F., and Allen, R.C. 1973. *Numerical Computing: An Introduction*, W. B. Saunders Company, Philadelphia, PA.

Shampine, L.F., and Gordon, M.K. 1975. *Computer Solution of Ordinary Differential Equations, The Initial Value Problem*, W. H. Freeman, San Francisco, CA.

Sharp, D. H., and Reinitz, J. 1998. Prediction of mutant expression patterns using gene circuits. *Biosystems*. 47, 79–90.

Solomon, M. J., Glotzer, M., Lee, T. H., Philippe, M., and Kirschner, M. W. 1990. Cyclin activation of P34cdc2. *Cell*. 63, 1013–1024.

Tang, Z., Coleman, T. R., and Dunphy, W. G. 1993. Two distinct mechanisms for negative regulation of the Wee1 protein kinase. *EMBO J.* 12, 9:3427–36.

Teusink, B., Passarge, J., Reijenga, C. A., Esgalhado, E., van der Weijden, C. C., Schepper, M., Walsh, M. C., Bakker, B. M., van Dam, K., Westerhoff, H. V., and Snoep, J. L.

2000. Can yeast glycolysis be understood in terms of in vitro kinetics of the constituent enzymes?. *Eur. J. Biochem.* 267, 5313–5329.

Tyson, J. J., Chen, K., and Novak, B. 2001. Network dynamics and cell physiology. *Nature Reviews Molecular Cell Biology.* 2, 908–16.

Tyson, J. J., Novak, B., Chen, K., and Val, J. 1995. Checkpoints in the cell cycle from a modeler's perspective. *Progress in Cell Cycle Research.* 1, 1–8.

von Dassow, G., Meir, E., Munro, E. M., and O'Dell, G. M. 2000. The segment polarity network is a robust developmental module. *Nature.* 406, 188–192.

Watson, L.T., Billups, S.C., and Morgan, A.P. 1987. Algorithm 652: HOMPACK: a suite of codes for globally convergent homotopy algorithms. *ACM Trans. Math. Software.* 13(3), 281–310.

Watson, L.T., Sosonkina, M., Melville, R.C., Morgan, A.P., and Walker, H.F. 1997. Algorithm 777: HOMPACK90: a suite of Fortran 90 codes for globally convergent homotopy algorithms. *ACM Trans. Math. Software.* 23(4), 514–549.

Watson, L.T., Baker, C.A. 2001. A fully-distributed parallel global search algorithm. *Engrg. Comput.*. 18, 155–169.

Wei, S., Moore, J., Chen, K., Lassaletta, A. D., Yi, C. S., Tyson, J. J., and Sible, J. C. 2002. Hysteresis drives cell-cycle transitions in *Xenopus Laevis* egg extracts. Submitted for publication.

Wise, S.M., Sommese, A.J., and Watson, L.T. 2000. Algorithm 801: POLSYS_PLP: a partitioned linear product homotopy code for solving polynomial systems of equations. *ACM Trans. Math. Software.* 26(1), 176–200.

Zwolak, J.W., Tyson, J.J., and Watson, L.T. 2004. Parameter Estimation for a Mathematical Model of the Cell Cycle in Frog Eggs. *Journal of Computational Biology.* in press.

APPENDIX A: Pseudocode for TIMELAG

TIMELAG takes as parameters C_T and a. C_T is the total cyclin concentration and a is 0 if MPF is initially inactive and 1 if MPF is initially active.

subroutine TIMELAG

if $C_T \leq 0$ **then**

  TIMELAG $= -1440 * C_T + 2880$;

  return;

endif

M_{inf} = asymptotic concentration of active MPF, retrieved from a call to LSODAR;

<u>Comment</u>: Check if MPF changes from active to inactive or vice versa. If it does not change its activity, then return.

if ($M_{inf} > C_T/2$ **and** $a = 1$) **or** ($M_{inf} < C_T/2$ **and** $a = 0$) **then**

  TIMELAG $= -1440 * C_T + 2880$;

  return

endif

<u>Comment</u>: Use the root finder in LSODAR to find the timelag.

if $a = 1$ **then**

  TIMELAG = time where MPF concentration is $(C_T - M_{inf})/2 + M_{inf}$

else

  TIMELAG = time where MPF concentration is $M_{inf}/2$

endif

return

end subroutine TIMELAG

APPENDIX B: Pseudocode for THRESHOLD

The THRESHOLD function calculates the threshold for MPF activation or inactivation for a given set of rate constants β. If the first parameter, a, to THRESHOLD is 1, then MPF is initially active and the inactivation threshold is calculated. If the first parameter to THRESHOLD is 0 then MPF is initially inactive and the activation threshold is calculated. The estimate for the threshold is returned in h.

```
subroutine THRESHOLD
  B := 0.001; Initialize the lower bound for the threshold. The initial value of B may not
    be a lower bound. A more realistic lower bound will be found later, if B is not a lower
    bound.
  C := 1; Initialize the upper bound for the threshold. The initial value of C may not be
    an upper bound. A more realistic upper bound will be found later, if C is not an upper
    bound.
  e_tol := 10^-10; Define the error tolerance for calculating the threshold.
  Find the lower bound on the threshold.
  while ((a = 0 and TIMELAG(a, β, b) < 1440) or (a = 1 and TIMELAG(a, β, b) ≥
1440)) and (b > e_tol) do
    b := b/2;
  enddo
  if b ≤ e_tol then
    h := b;
    return;
  endif
  Find the upper bound on the threshold.
  while ((a = 0 and TIMELAG(a, β, c) ≥ 1440) or (a = 1 and TIMELAG(a, β, c) <
1440)) and (c < 1/e_tol) do
    c := c * 2;
  enddo
  if c ≥ 1/e_tol then
    h := c;
    return;
  endif
  while (c − b)/b > e_tol do
    next = (c + b)/2; Bisect the interval.
    if (a = 0 and TIMELAG(a, β, next) ≥ 1440) or (a = 1 and TIMELAG(a, β, next) <
    1440) then
      b := next;
    else
      c := next;
    endif
  enddo
  h := b;
end subroutine THRESHOLD
```

APPENDIX C: Results from POLSYS_PLP

All solutions from POLSYS_PLP for the example problem with one parameter set.

$$
\begin{aligned}
z^{(1)} &= (\ -3.18\text{E}{+}08 - i3.69\text{E}{+}07, \\
&\quad\ \ 4.24\text{E}{-}01 + i8.88\text{E}{-}02, \\
&\quad -9.28\text{E}{-}02 - i1.37\text{E}{-}02) \\
z^{(2)} &= (\ \ \ 8.33\text{E}{-}03 - i6.37\text{E}{-}04, \\
&\quad\ \ 3.85\text{E}{+}09 + i5.81\text{E}{+}08, \\
&\quad -9.24\text{E}{+}08 - i1.59\text{E}{+}09) \\
z^{(3)} &= (\ \ \ 7.83\text{E}{+}07 + i1.50\text{E}{+}07, \\
&\quad\ \ 4.41\text{E}{-}01 + i3.09\text{E}{-}01, \\
&\quad -9.53\text{E}{-}02 - i4.79\text{E}{-}02) \\
z^{(4)} &= (\ \ \ 1.39\text{E}{-}01 - i1.33\text{E}{-}09, \\
&\quad -1.59\text{E}{+}00 + i3.94\text{E}{-}08, \\
&\quad\ \ 2.16\text{E}{-}02 + i1.29\text{E}{-}09) \\
z^{(5)} &= (\ \ \ 1.86\text{E}{-}02 + i1.43\text{E}{-}02, \\
&\quad\ \ 1.08\text{E}{+}00 + i1.60\text{E}{-}01, \\
&\quad\ \ 6.31\text{E}{-}01 - i5.00\text{E}{-}01) \\
z^{(6)} &= (\ \ \ 5.87\text{E}{-}02 - i1.65\text{E}{-}02, \\
&\quad -5.87\text{E}{+}06 + i9.79\text{E}{+}06, \\
&\quad -1.27\text{E}{+}06 + i2.12\text{E}{+}06) \\
z^{(7)} &= (\ -4.69\text{E}{+}08 - i3.02\text{E}{+}09, \\
&\quad -2.19\text{E}{-}01 - i1.00\text{E}{+}00, \\
&\quad\ \ 6.83\text{E}{-}03 + i1.55\text{E}{-}01) \\
z^{(8)} &= (\ \ \ 5.23\text{E}{-}02 + i1.06\text{E}{-}02, \\
&\quad -1.54\text{E}{+}06 - i2.46\text{E}{+}05, \\
&\quad -3.33\text{E}{+}05 - i5.31\text{E}{+}04) \\
z^{(9)} &= (\ -6.15\text{E}{-}04 - i2.62\text{E}{-}03, \\
&\quad -1.51\text{E}{+}07 - i1.40\text{E}{+}06, \\
&\quad -3.32\text{E}{+}06 - i5.24\text{E}{+}06) \\
z^{(10)} &= (\ \ \ 3.34\text{E}{-}03 - i3.05\text{E}{-}10, \\
&\quad\ \ 6.43\text{E}{-}02 - i1.29\text{E}{-}08, \\
&\quad\ \ 9.36\text{E}{-}01 + i6.02\text{E}{-}09) \\
z^{(11)} &= (\ \ \ 9.43\text{E}{-}02 + i2.65\text{E}{-}08, \\
&\quad\ \ 9.65\text{E}{-}01 + i1.80\text{E}{-}07, \\
&\quad\ \ 3.52\text{E}{-}02 + i2.04\text{E}{-}08) \\
z^{(12)} &= (\ \ \ 2.16\text{E}{-}02 + i5.22\text{E}{-}11, \\
&\quad\ \ 5.46\text{E}{-}01 + i2.96\text{E}{-}10,
\end{aligned}
$$

$$
\begin{aligned}
& \quad 4.54\text{E}{-01} - i1.72\text{E}{-09}) \\
z^{(13)} = (& -5.45\text{E}{-03} - i1.34\text{E}{-10}, \\
& -9.07\text{E}{-02} + i9.37\text{E}{-10}, \\
& -8.27\text{E}{-02} - i1.24\text{E}{-09}) \\
z^{(14)} = (& \quad 1.86\text{E}{-02} - i1.43\text{E}{-02}, \\
& \quad 1.08\text{E}{+00} - i1.60\text{E}{-01}, \\
& \quad 6.31\text{E}{-01} + i5.00\text{E}{-01}) \\
z^{(15)} = (& \quad 6.23\text{E}{-02} + i5.91\text{E}{-02}, \\
& \quad 6.60\text{E}{+09} + i2.92\text{E}{+10}, \\
& -2.03\text{E}{+09} - i7.28\text{E}{+09}) \\
z^{(16)} = (& \quad 1.08\text{E}{-02} + i1.63\text{E}{-02}, \\
& \quad 6.61\text{E}{+11} - i1.97\text{E}{+11}, \\
& \quad 1.05\text{E}{+11} - i6.57\text{E}{+11}) \\
z^{(17)} = (& -3.78\text{E}{+06} + i3.94\text{E}{+09}, \\
& -1.04\text{E}{-01} - i1.10\text{E}{+00}, \\
& -1.10\text{E}{-02} + i1.69\text{E}{-01}) \\
z^{(18)} = (& -6.81\text{E}{-03} + i4.74\text{E}{-04}, \\
& -1.41\text{E}{+08} + i6.82\text{E}{+08}, \\
& \quad 5.34\text{E}{+08} - i1.12\text{E}{+08})
\end{aligned}
$$

APPENDIX D: Model Parameters for Chapter 4

The parameter set used for the solutions in Appendix C.

$$
\begin{array}{lll}
v_d = 2 & K_{md} = 0.1 & v'_d = 0.017 \\
v_{dr} = 0.1 & K_{mdr} = 1.0 & v''_d = 0.17 \\
v_w = 2 & K_{mw} = 0.1 & v'_w = 0.01 \\
v_{wr} = 0.1 & K_{mwr} = 1.0 & v''_w = 1 \\
C_T = 0.12 & &
\end{array}
$$

APPENDIX E: Example ODRPACK95 Usage

A simple example of ODRPACK95 usage on a sample problem.

```
PROGRAM ODRPACK95_EXAMPLE
   USE ODRPACK95
   USE REAL_PRECISION
   REAL (KIND=R8), ALLOCATABLE :: BETA(:),L(:),U(:),X(:,:),Y(:,:)
   INTEGER :: NP,N,M,NQ
   INTERFACE
      SUBROUTINE FCN(N,M,NP,NQ,LDN,LDM,LDNP,BETA,XPLUSD,IFIXB,IFIXX,LDIFX,&
         IDEVAL,F,FJACB,FJACD,ISTOP)
         USE REAL_PRECISION
         INTEGER :: IDEVAL,ISTOP,LDIFX,LDM,LDN,LDNP,M,N,NP,NQ
         REAL (KIND=R8) :: BETA(NP),F(LDN,NQ),FJACB(LDN,LDNP,NQ), &
            FJACD(LDN,LDM,NQ),XPLUSD(LDN,M)
         INTEGER :: IFIXB(NP),IFIXX(LDIFX,M)
      END SUBROUTINE FCN
   END INTERFACE

   NP = 2
   N  = 4
   M  = 1
   NQ = 1
   ALLOCATE(BETA(NP),L(NP),U(NP),X(N,M),Y(N,NQ))
   BETA(1:2) = (/ 2.0_R8, 0.5_R8 /)
   L(1:2)    = (/ 0.0_R8, 0.0_R8 /)
   U(1:2)    = (/ 10.0_R8, 0.9_R8 /)
   X(1:4,1)  = (/ 0.982_R8, 1.998_R8, 4.978_R8, 6.01_R8 /)
   Y(1:4,1)  = (/ 2.7_R8, 7.4_R8, 148.0_R8, 403.0_R8 /)
   CALL ODR(FCN,N,M,NP,NQ,BETA,Y,X,LOWER=L,UPPER=U)
END PROGRAM ODRPACK95_EXAMPLE

SUBROUTINE FCN(N,M,NP,NQ,LDN,LDM,LDNP,BETA,XPLUSD,IFIXB,IFIXX,LDIFX,&
   IDEVAL,F,FJACB,FJACD,ISTOP)
   USE REAL_PRECISION
   INTEGER :: IDEVAL,ISTOP,LDIFX,LDM,LDN,LDNP,M,N,NP,NQ
   REAL (KIND=R8) :: BETA(NP),F(LDN,NQ),FJACB(LDN,LDNP,NQ), &
      FJACD(LDN,LDM,NQ),XPLUSD(LDN,M)
   INTEGER :: IFIXB(NP),IFIXX(LDIFX,M)
   ISTOP = 0
   ! Calculate model.
   IF (MOD(IDEVAL,10).NE.0) THEN
      DO I=1,N
         F(I,1) = BETA(1)*EXP(BETA(2)*XPLUSD(I,1))
      END DO
   END IF
   ! Calculate model partials with respect to BETA.
   IF (MOD(IDEVAL/10,10).NE.0) THEN
      DO I=1,N
         FJACB(I,1,1) = EXP(BETA(2)*XPLUSD(I,1))
         FJACB(I,2,1) = BETA(1)*XPLUSD(I,1)*EXP(BETA(2)*XPLUSD(I,1))
      END DO
   END IF
   ! Calculate model partials with respect to DELTA.
```

```
   IF (MOD(IDEVAL/100,10).NE.0) THEN
      DO I=1,N
         FJACD(I,1,1) = BETA(1)*BETA(2)*EXP(BETA(2)*XPLUSD(I,1))
      END DO
   END IF

END SUBROUTINE FCN
```

APPENDIX F: Output of ODRPACK95 for a Sample Problem

The output of the example program from Appendix E.

```
*******************************************************
* ODRPACK95 VERSION 1.00 OF 07-15-2004 (REAL (KIND=R8)) *
*******************************************************


*** INITIAL SUMMARY FOR FIT BY METHOD OF ODR ***

--- PROBLEM SIZE:
           N =     4          (NUMBER WITH NONZERO WEIGHT =     4)
          NQ =     1
           M =     1
          NP =     2          (NUMBER UNFIXED =     2)

--- CONTROL VALUES:
         JOB = 00000
             = ABCDE, WHERE
                     A=0 ==> FIT IS NOT A RESTART.
                     B=0 ==> DELTAS ARE INITIALIZED TO ZERO.
                     C=0 ==> COVARIANCE MATRIX WILL BE COMPUTED USING
                             DERIVATIVES RE-EVALUATED AT THE SOLUTION.
                     D=0 ==> DERIVATIVES ARE ESTIMATED BY FORWARD DIFFERENCES.
                     E=0 ==> METHOD IS EXPLICIT ODR.
      NDIGIT =    16          (ESTIMATED BY ODRPACK95)
      TAUFAC =     1.00E+00

--- STOPPING CRITERIA:
       SSTOL =     1.49E-08   (SUM OF SQUARES STOPPING TOLERANCE)
      PARTOL =     3.67E-11   (PARAMETER STOPPING TOLERANCE)
       MAXIT =    50          (MAXIMUM NUMBER OF ITERATIONS)

--- INITIAL WEIGHTED SUM OF SQUARES        =                    1.46854548E+05
        SUM OF SQUARED WEIGHTED DELTAS     =   0.00000000E+00
        SUM OF SQUARED WEIGHTED EPSILONS   =   1.46854548E+05

--- FUNCTION PARAMETER SUMMARY:

      INDEX   BETA(K)    FIXED     SCALE   LOWER(K)   UPPER(K)    DERIVATIVE
                                                                   STEP SIZE
        (K)            (IFIXB)    (SCLB)                             (STPB)

          1  2.00E+00       NO  5.00E-01  0.00E+000  1.00E+001   1.00000E-10
          2  5.00E-01       NO  5.00E-01  0.00E+000  9.00E-001   1.00000E-10

--- EXPLANATORY VARIABLE AND DELTA WEIGHT SUMMARY:

      INDEX      X(I,J)  DELTA(I,J)    FIXED     SCALE    WEIGHT    DERIVATIVE
                                                                     STEP SIZE
      (I,J)                          (IFIXX)    (SCLD)      (WD)        (STPD)

        1,1   9.820E-01   0.000E+00       NO  1.66E-01  1.00E+00   1.00000E-10
```

```
        N,1   6.010E+00   0.000E+00       NO  1.66E-01  1.00E+00   1.00000E-10

--- RESPONSE VARIABLE AND EPSILON ERROR WEIGHT SUMMARY:

      INDEX      Y(I,L)      WEIGHT
      (I,L)                    (WE)

        1,1   2.700E+00   1.000E+00
        N,1   4.030E+02   1.000E+00

*** FINAL SUMMARY FOR FIT BY METHOD OF ODR ***

--- STOPPING CONDITIONS:
        INFO =     1 ==> SUM OF SQUARES CONVERGENCE.
       NITER =    25          (NUMBER OF ITERATIONS)
        NFEV =   140          (NUMBER OF FUNCTION EVALUATIONS)
       IRANK =     0          (RANK DEFICIENCY)
       RCOND =     7.68E-02   (INVERSE CONDITION NUMBER)
       ISTOP =     0          (RETURNED BY USER FROM SUBROUTINE FCN)

--- FINAL WEIGHTED SUMS OF SQUARES       =                    2.67368608E-01
        SUM OF SQUARED WEIGHTED DELTAS   =    2.46882426E-01
        SUM OF SQUARED WEIGHTED EPSILONS =    2.04861824E-02

--- RESIDUAL STANDARD DEVIATION          =                    3.65628642E-01
        DEGREES OF FREEDOM               =    2

--- ESTIMATED BETA(J), J = 1, ..., NP:

                   BETA      LOWER     UPPER      S.D.  __ 95% CONFIDENCE __
                                                 BETA         INTERVAL

      1   1.63337057E+00   0.00E+00  1.00E+01  5.02E-01  -5.26E-01 TO  3.79E+00
      2   9.00000000E-01   0.00E+00  9.00E-01  7.44E-02   5.80E-01 TO  1.22E+00
```

APPENDIX G: Source Code

This appendix contains the source code used for local and global optimization of the parameters of the frog egg model. This is exactly the code used in Chapter 3, but is more recent than the code used in Chapter 2. The code developed in Chapter 5 was never used on this problem.

lsodar_mod.f90

```
MODULE LSODAR_MOD

!==============================================================================
!
!  MODULE LSODAR_MOD
!
!  This module provides an easy to use wrapper to the LSODAR package from
!  ODEPACK.  The wrapper also provides additional functionality.
!
!  This module is used by
!
!       setting the appropriate variables using the SET routine,
!       calling the LSODAR_RUN or LSODAR_SIMULATE routine,
!       getting the desired results from the GET routine.
!
!  There are parameters which tell LSODAR_MOD to output variable values at fixed
!  time steps or under specified conditions.  This can be used in addition to or
!  instead of the GET routine.
!
!  The user is expected to implement external routines by the names of
!  MODEL_RHS, MODEL_ROOTS, MODEL_FOUND_ROOT as defined in MODEL_MOD.
!
!  This module has the following flow of execution:
!    - The user calls SET.  The user may call SET many times.
!    - The user calls LSODAR_RUN or LSODAR_SIMULATE.  Those routines call CHECK.
!    - CHECK checks if all necessary variables are set and allocated and
!      attempts to allocate the necessary variables.
!    - The simulation is run.  By default the variables are printed to stdout at
!      a specified time step.  The user may change this.
!    - The user may optionally call GET to get some of the integrator's returned
!      results.
!
!  LSODAR_RUN will run lsodar and return when lsodar returns.
!  LSODAR_SIMULATE will call LSODAR_RUN and check for roots when LSODAR_RUN
!  returns.  If a root is found the user routine MODEL_FOUND_ROOT will be called
!  and LSODAR_RUN will be called again.
!
!  Typically the user will call LSODAR_SAVE after initial setting of the lsodar
!  parameters.  By doing so the user can make many calls to LSODAR_RUN or
!  LSODAR_SIMULATE by simply calling LSODAR_RECALL (and perhaps change a few
!  parameters) between runs.
!

USE REAL_PRECISION

PRIVATE
```

```
!-------------------------------------------------------------------------------
! LSODAR subroutine vars.  These vars are actually passed to LSODAR directly or
! indirectly.  They are the vars set when SET is called.
!
! The documentation provided here is intended for quick reference.  The
! documentation comes from the lsodar.f source file.  More complete
! documentation can be found in lsodar.f
!
! "_I"s are appended to the variable names for internal representation.  "_"s
! are appended for the backups.
!
! KEY
!
!   ->  IN
!  <-   OUT
!  <->  IN and OUT
!   S   Must be set before LSODAR can be called.
!   A   Must be allocated before LSODAR can be called.
!   O   Optional input.
!
!
!-------------------------------------------------------------------------------
!  -> O  DEBUG          default = .FALSE.
!                       If .TRUE. then debugging information is printed inside
!                       some subroutines.
!  -> S  NEQ            default = none, must be set by user.
!                       Number of ODEs.
!  -> S  ITOL           default = 1
!                       Indicator of the type of error control.  This may not be
!                       changed.  1 specifies scalar ATOL.
!  -> S  ITASK          Task to be performed.
!                       default 1  normal computations
!                               2  take one step
!                               3  stop at first internal mesh point
!                               4  normal computations with overshooting
!                               5  take one step without passing tcrit
! <-> S  ISTATE         Specifies the state of the integrator.
!                       On input
!                       default 1  this is the first call
!                               2  continue the integration from previous call
!                                  where only tout and itask have been modified
!                               3  continue integration where any param could
!                                  have been modified.
!                       On output
!                               1  nothing was done
!                               2  integration performed successfully, no roots
!                               3  integration seccussfull and roots found
!                               -  some error has occurred, see lsodar docs or
!                                  error messages in this code
!  -> S  IOPT           0 (default) no optional arguments, 1 optional arguments
!                       IOPT is automatically set when the user sets an optional
!                       variable.  Once IOPT is set, it is always
!                       set.  If the user subsequently doesn't want to use
!                       optional arguments, he/she simply sets all the optional
!                       arguments to 0.  IOPT will still be set, but each
!                       optional argument uses a default if set to 0.
!  -> S  LRW            default = 22+NEQ*max(16,NEQ+9)+3*NG
!                       Length of RWORK.
! <->  A IWORK          defaults = 0
!                       Integer work array.  There are optional inputs here when
!                       IOPT = 1
!  -> S  LIW            default = 20+NEQ
!                       Length of IWORK.
!  -> S  JT             Jacobian type indicator.  This variable cannot be
!                       modified.
!                               1  user supplied full Jacobian
!                       default 2  internally generate full Jacobian
!                               4  user supplied banded Jacobian
!                               5  internally generated banded Jacobian
!  -> S  NG             default = 0
!                       Number of root functions.
! <-   A JROOT          JROOT(i) = 1 if and only if GEX(i) has a root at T.
!  -> O  MAXWORK        default = 0, tells lsodar to use its default.
!                       Maximum steps the integrator may take (mxstep)
!  -> O  IXPR           default = 0, tells lsodar to use its default.
!                       Whether to print extra printing on method switches.
!  -> O  MXHNIL         default = 0, tells lsodar to use its default.
!  -> O  MXORDN         default = 0, tells lsodar to use its default.
```

```
!  -> O  MXORDS         default = 0, tells lsodar to use its default.
! <-> SA Y              defaults = 0
!                       On input: initial conditions.
!                       On output: variable values at T.
! <-> S  T              default = 0
!                       On input: Current time (usually 0).
!                       On output: time the integration stopped.
!  -> S  TOUT           default = 10, an arbitrary default.  User should
!                       overwrite.
!                       The next value of T a solution is desired.
!  -> S  RTOL           default = 1D-8
!                       Relative error tolerance.
!  -> S  ATOL           default = 1D-8
!                       Absolute error tolerance.
! <->  A RWORK          defaults = 0
!                       Real work array.  Optional inputs can only be set if
!                       IOPT = 1.
!  -> O  H0             default = 0, tells lsodar to use its default.
!  -> O  HMAX           default = 0, tells lsodar to use its default.
!  -> O  HMIN           default = 0, tells lsodar to use its default.
!  -> O  PRINT_EVERY    default = 0, tells this module to print variables every
!                       PRINT_EVERY time units.  A value of <=0 means do not
!                       print anything.  The printing goes to FILE_UNIT.
!  -> S  ROOT_TOL       default = 1D-12
!                       If GOUT is within ROOT_TOL of a root then GEX sets GOUT
!                       to zero forcing LSODAR to find a root.
!  -> S  ROOT_TTOL      default = 1D-3
!                       If a root is found then run the next ROOT_TTOL time
!                       steps with root finding off.
!  -> O  FIND_ROOTS     default = 1
!                       0       Turns root finding off.
!                       1       Turns root finding on.
!                       This variable can be used to turn root finding off and
!                       on.  It exists because after a root is found lsodar
!                       usually cannot resume integration without finding the
!                       same root again.  This allows the user (and
!                       LSODAR_SIMULATE) to turn root finding off briefly.
!  -> O  FILE_UNIT      default = 6
!                       The file unit to print results from PRINT_EVERY to.
!  -> O  NUM_AUX        default = 0
!                       Defines the number of auxiliary variables to print.  The
!                       auxiliary variables are defined in MODEL_AUXILIARY.  The
!                       auxiliary variables are printed after the Y variables.
!-------------------------------------------------------------------------------
!
!  The variables that are required to be set are set to -1 by default.  -1 is an
!  illegal input for all these variables.  This code checks if the inputs are
!  negative to determine when the input is ready for a call to lsodar.
!
!  Some of the variables can be set automatically and are.
!
!  There are two exceptions.  T and TOUT can have negative values.  They are set
!  to valid defaults (0 and 10 respectively).  These defaults may not be what
!  the user wants and may give undesired results.  However, they will guarantee
!  the integration works and no segmentation faults are given.
!
INTEGER          :: NEQ_I=-1, ITOL_I=1, ITASK_I=1, ISTATE_I=1, IOPT_I=0, &
                    LRW_I=-1,                                            &
                    IWORK_I(:), LIW_I=-1, JT_I=2, NG_I=0, JROOT_I(:),    &
                    MAXWORK_I=0,                                         &
                    IXPR_I=0, MXHNIL_I=0, MXORDN_I=0, MXORDS_I=0,        &
                    FIND_ROOTS_I=1, FILE_UNIT_I=6, NUM_AUX_I=0
REAL (KIND=R8)   :: Y_I(:), T_I=0, TOUT_I=10, RTOL_I=1D-8, ATOL_I=1D-8,  &
                    RWORK_I(:), H0_I=0,                                  &
                    HMAX_I=0, HMIN_I=0, PRINT_EVERY_I=0,                 &
                    ROOT_TOL_I=1D-12, ROOT_TTOL_I=1D-3
ALLOCATABLE      :: Y_I, IWORK_I, JROOT_I, RWORK_I
LOGICAL          :: DEBUG_I = .FALSE.



! LSODAR vars, backups, for saving and loading
INTEGER          :: NEQ_, ITOL_, ITASK_, ISTATE_, IOPT_, LRW_,      &
                    IWORK_(:), LIW_, JT_, NG_, JROOT_(:), MAXWORK_, &
                    IXPR_, MXHNIL_, MXORDN_, MXORDS_, FIND_ROOTS_,  &
                    FILE_UNIT_, NUM_AUX_
REAL (KIND=R8)                                                      &
                 :: Y_(:), T_, TOUT_, RTOL_, ATOL_, RWORK_(:), H0_, &
                    HMAX_, HMIN_, PRINT_EVERY_, ROOT_TOL_, ROOT_TTOL_
```

```
ALLOCATABLE      :: Y_, IWORK_, JROOT_, RWORK_
LOGICAL          :: DEBUG_




PUBLIC LSODAR_SET, LSODAR_RUN, LSODAR_SIMULATE, LSODAR_SAVE, LSODAR_RECALL, &
       LSODAR_GET




CONTAINS




!-----------------------------------------------------------------------------
!
!  LSODAR_SAVE
!
!  Saves the current variables (*_I) to the backup variables (*_).  This routine
!  saves every variable local to the LSODAR_MOD module, but not local variables
!  of subroutines and functions.
!

SUBROUTINE LSODAR_SAVE ()

DEBUG_           = DEBUG_I
NEQ_             = NEQ_I
ITOL_            = ITOL_I
ITASK_           = ITASK_I
ISTATE_          = ISTATE_I
IOPT_            = IOPT_I
LRW_             = LRW_I
IWORK_(1:LIW_I) = IWORK_I(1:LIW_I)
LIW_             = LIW_I
JT_              = JT_I
NG_              = NG_I
JROOT_(1:NG_I)  = JROOT_I(1:NG_I)
MAXWORK_         = MAXWORK_I
IXPR_            = IXPR_I
MXHNIL_          = MXHNIL_I
MXORDN_          = MXORDN_I
Y_(1:NEQ_I)      = Y_I(1:NEQ_I)
T_               = T_I
TOUT_            = TOUT_I
RTOL_            = RTOL_I
ATOL_            = ATOL_I
RWORK_(1:LRW_I) = RWORK_I(1:LRW_I)
HO_              = HO_I
HMAX_            = HMAX_I
HMIN_            = HMIN_I
PRINT_EVERY_     = PRINT_EVERY_I
ROOT_TOL_        = ROOT_TOL_I
ROOT_TTOL_       = ROOT_TTOL_I
FIND_ROOTS_      = FIND_ROOTS_I
FILE_UNIT_       = FILE_UNIT_I
NUM_AUX_         = NUM_AUX_I

END SUBROUTINE LSODAR_SAVE



!-----------------------------------------------------------------------------
!
!  LSODAR_GET
!
!  Returns the user requested variables.
!  More variables need to be added to this routine.
!
SUBROUTINE LSODAR_GET( &
```

```
    DEBUG,            &
    NEQ,              &
    ITASK,            &
    ISTATE,           &
    NG,               &
    MAXWORK,          &
    IXPR,             &
    MXHNIL,           &
    MXORDN,           &
    MXORDS,           &
    Y,                &
    T,                &
    TOUT,             &
    RTOL,             &
    ATOL,             &
    HO,               &
    HMAX,             &
    HMIN,             &
    PRINT_EVERY,      &
    ROOT_TOL,         &
    ROOT_TTOL,        &
    JROOT,            &
    FIND_ROOTS,       &
    FILE_UNIT,        &
    NUM_AUX           &
)
    INTEGER, OPTIONAL, INTENT(OUT)        :: NEQ,              &
                                             ITASK,            &
                                             ISTATE,           &
                                             NG,               &
                                             MAXWORK,          &
                                             IXPR,             &
                                             MXHNIL,           &
                                             MXORDN,           &
                                             MXORDS,           &
                                             JROOT(:),         &
                                             FIND_ROOTS,       &
                                             FILE_UNIT,        &
                                             NUM_AUX
```

```
REAL (KIND=R8), OPTIONAL, INTENT(OUT) :: Y(:),             &
                                         T,                &
                                         TOUT,             &
                                         RTOL,             &
                                         ATOL,             &
                                         HO,               &
                                         HMAX,             &
                                         HMIN,             &
                                         PRINT_EVERY,      &
                                         ROOT_TOL,         &
                                         ROOT_TTOL
LOGICAL, OPTIONAL, INTENT(OUT)        :: DEBUG

IF ( PRESENT(DEBUG)       ) THEN ; DEBUG       = DEBUG_I       ; END IF
IF ( PRESENT(NEQ)         ) THEN ; NEQ         = NEQ_I         ; END IF
IF ( PRESENT(ITASK)       ) THEN ; ITASK       = ITASK_I       ; END IF
IF ( PRESENT(ISTATE)      ) THEN ; ISTATE      = ISTATE_I      ; END IF
IF ( PRESENT(NG)          ) THEN ; NG          = NG_I          ; END IF
IF ( PRESENT(MAXWORK)     ) THEN ; MAXWORK     = MAXWORK_I     ; END IF
IF ( PRESENT(IXPR)        ) THEN ; IXPR        = IXPR_I        ; END IF
IF ( PRESENT(MXHNIL)      ) THEN ; MXHNIL      = MXHNIL_I      ; END IF
IF ( PRESENT(MXORDN)      ) THEN ; MXORDN      = MXORDN_I      ; END IF
IF ( PRESENT(MXORDS)      ) THEN ; MXORDS      = MXORDS_I      ; END IF
IF ( PRESENT(T)           ) THEN ; T           = T_I           ; END IF
IF ( PRESENT(TOUT)        ) THEN ; TOUT        = TOUT_I        ; END IF
IF ( PRESENT(RTOL)        ) THEN ; RTOL        = RTOL_I        ; END IF
IF ( PRESENT(ATOL)        ) THEN ; ATOL        = ATOL_I        ; END IF
IF ( PRESENT(HO)          ) THEN ; HO          = HO_I          ; END IF
IF ( PRESENT(HMAX)        ) THEN ; HMAX        = HMAX_I        ; END IF
IF ( PRESENT(HMIN)        ) THEN ; HMIN        = HMIN_I        ; END IF
IF ( PRESENT(PRINT_EVERY) ) THEN ; PRINT_EVERY = PRINT_EVERY_I ; END IF
IF ( PRESENT(ROOT_TOL)    ) THEN ; ROOT_TOL    = ROOT_TOL_I    ; END IF
IF ( PRESENT(ROOT_TTOL)   ) THEN ; ROOT_TTOL   = ROOT_TTOL_I   ; END IF
IF ( PRESENT(FIND_ROOTS)  ) THEN ; FIND_ROOTS  = FIND_ROOTS_I  ; END IF
IF ( PRESENT(FILE_UNIT )  ) THEN ; FILE_UNIT   = FILE_UNIT_I   ; END IF
IF ( PRESENT(NUM_AUX)     ) THEN ; NUM_AUX     = NUM_AUX_I     ; END IF
IF ( PRESENT(Y) .AND. ALLOCATED(Y_I) ) THEN ; Y(1:NEQ_I) = Y_I(1:NEQ_I) ; END IF
IF ( PRESENT(JROOT) .AND. ALLOCATED(JROOT_I) ) THEN
    JROOT(1:NG_I) = JROOT_I(1:NG_I)
```

```
    END IF


END SUBROUTINE LSODAR_GET




!-----------------------------------------------------------------------------
!
!  LSODAR_RECALL
!
!  Recalls the saved variables (*_) to the current variables (*_I).  This is
!  mirror function of LSODAR_SAVE.
!

SUBROUTINE LSODAR_RECALL ()

DEBUG_I           = DEBUG_
NEQ_I             = NEQ_
ITOL_I            = ITOL_
ITASK_I           = ITASK_
ISTATE_I          = ISTATE_
IOPT_I            = IOPT_
LRW_I             = LRW_
IWORK_I(1:LIW_)   = IWORK_(1:LIW_)
LIW_I             = LIW_
JT_I              = JT_
NG_I              = NG_
JROOT_I(1:NG_)    = JROOT_(1:NG_)
MAXWORK_I         = MAXWORK_
IXPR_I            = IXPR_
MXHNIL_I          = MXHNIL_
MXORDN_I          = MXORDN_
Y_I(1:NEQ_)       = Y_(1:NEQ_)
T_I               = T_
TOUT_I            = TOUT_
RTOL_I            = RTOL_
ATOL_I            = ATOL_
RWORK_I(1:LRW_)   = RWORK_(1:LRW_)
HO_I              = HO_
HMAX_I            = HMAX_
HMIN_I            = HMIN_
PRINT_EVERY_I     = PRINT_EVERY_
ROOT_TOL_I        = ROOT_TOL_
ROOT_TTOL_I       = ROOT_TTOL_
FIND_ROOTS_I      = FIND_ROOTS_
FILE_UNIT_I       = FILE_UNIT_
NUM_AUX_I         = NUM_AUX_

END SUBROUTINE LSODAR_RECALL



!-----------------------------------------------------------------------------
!
!  LSODAR_SET
!
!  SIDE EFFECTS: Saved variables could be invalidated.  This happens if and only
!  if NEQ is set.
!
!  STATUS: This subroutine is complete and partially tested
!
!  Allows the user to set some lsodar variables.
!
!  NEQ must be set.
!  ITOL may not be set.  This wrapper only allows scalar atol.
!  ITASK may be set.  The default is 1.
!  ISTATE may be set.  The default is 1.
!  IOPT is automatically set when one of the optional lsodar arguments is used.
!  LRW is automatically set.
!  IWORK is automatically set and allocated.
!  LIW is automatically set.
!  JT may not be set.  Its value is 2.
```

```
!  NG may be set.  defaults to 0.
!  MAXWORK may be set. Its default value is 0 telling lsodar to use its default.
!  IXPR may be set. Its default value is 0 telling lsodar to use its default.
!  MXHNIL may be set. Its default value is 0 telling lsodar to use its default.
!  MXORDN may be set. Its default value is 0 telling lsodar to use its default.
!  MXORDS may be set. Its default value is 0 telling lsodar to use its default.
!  Y may be set.  It defaults to all zeros.  It is automatically allocated.
!  T may be set.  It defaults to zero.
!  TOUT ought to be set.  It defaults to 10.
!  RTOL may be set.  It defaults to 1D-8.
!  ATOL may be set.  It defaults to 1D-8.
!  RWORK is automatically set and allocated.
!  H0 may be set. Its default value is 0 telling lsodar to use its default.
!  HMAX may be set. Its default value is 0 telling lsodar to use its default.
!  HMIN may be set. Its default value is 0 telling lsodar to use its default.
!  JROOT is automatically allocated (it cannot be set, it is an output).
!  PRINT_EVERY may be set.  The default is 0.
!  FIND_ROOTS may be set.  The default is 0.
!  FILE_UNIT may be set.  The default is 6.
!

SUBROUTINE LSODAR_SET( &
    DEBUG,             &
    NEQ,               &
    ITASK,             &
    ISTATE,            &
    NG,                &
    MAXWORK,           &
    IXPR,              &
    MXHNIL,            &
    MXORDN,            &
    MXORDS,            &
    Y,                 &
    T,                 &
    TOUT,              &
    RTOL,              &
    ATOL,              &
    H0,                &
    HMAX,              &
    HMIN,              &
    PRINT_EVERY,       &
    ROOT_TOL,          &
    ROOT_TTOL,         &
    FIND_ROOTS,        &
    FILE_UNIT,         &
    NUM_AUX            &
)
    INTEGER, OPTIONAL, INTENT(IN)        :: NEQ,              &
                                            ITASK,            &
                                            ISTATE,           &
                                            NG,               &
                                            MAXWORK,          &
                                            IXPR,             &
                                            MXHNIL,           &
                                            MXORDN,           &
                                            MXORDS,           &
                                            FIND_ROOTS,       &
                                            FILE_UNIT,        &
                                            NUM_AUX
    REAL (KIND=R8), OPTIONAL, INTENT(IN) :: Y(:),             &
                                            T,                &
                                            TOUT,             &
                                            RTOL,             &
                                            ATOL,             &
                                            H0,               &
                                            HMAX,             &
                                            HMIN,             &
                                            PRINT_EVERY,      &
                                            ROOT_TOL,         &
                                            ROOT_TTOL
    LOGICAL, OPTIONAL, INTENT(IN)        :: DEBUG

    IF ( PRESENT(DEBUG)       ) THEN ; DEBUG_I       = DEBUG       ; END IF
    IF ( PRESENT(NEQ)         ) THEN ; NEQ_I         = NEQ         ; END IF
    IF ( PRESENT(ITASK)       ) THEN ; ITASK_I       = ITASK       ; END IF
    IF ( PRESENT(ISTATE)      ) THEN ; ISTATE_I      = ISTATE      ; END IF
    IF ( PRESENT(NG)          ) THEN ; NG_I          = NG          ; END IF
    IF ( PRESENT(MAXWORK)     ) THEN ; MAXWORK_I     = MAXWORK     ; END IF
```

```
IF ( PRESENT(IXPR)        ) THEN ; IXPR_I        = IXPR        ; END IF
IF ( PRESENT(MXHNIL)      ) THEN ; MXHNIL_I      = MXHNIL      ; END IF
IF ( PRESENT(MXORDN)      ) THEN ; MXORDN_I      = MXORDN      ; END IF
IF ( PRESENT(MXORDS)      ) THEN ; MXORDS_I      = MXORDS      ; END IF
IF ( PRESENT(T)           ) THEN ; T_I           = T           ; END IF
IF ( PRESENT(TOUT)        ) THEN ; TOUT_I        = TOUT        ; END IF
IF ( PRESENT(RTOL)        ) THEN ; RTOL_I        = RTOL        ; END IF
IF ( PRESENT(ATOL)        ) THEN ; ATOL_I        = ATOL        ; END IF
IF ( PRESENT(H0)          ) THEN ; H0_I          = H0          ; END IF
IF ( PRESENT(HMAX)        ) THEN ; HMAX_I        = HMAX        ; END IF
IF ( PRESENT(HMIN)        ) THEN ; HMIN_I        = HMIN        ; END IF
IF ( PRESENT(PRINT_EVERY) ) THEN ; PRINT_EVERY_I = PRINT_EVERY ; END IF
IF ( PRESENT(ROOT_TOL)    ) THEN ; ROOT_TOL_I    = ROOT_TOL    ; END IF
IF ( PRESENT(ROOT_TTOL)   ) THEN ; ROOT_TTOL_I   = ROOT_TTOL   ; END IF
IF ( PRESENT(FIND_ROOTS)  ) THEN ; FIND_ROOTS_I  = FIND_ROOTS  ; END IF
IF ( PRESENT(FILE_UNIT)   ) THEN ; FILE_UNIT_I   = FILE_UNIT   ; END IF
IF ( PRESENT(NUM_AUX)     ) THEN ; NUM_AUX_I     = NUM_AUX     ; END IF


! Set IOPT if optional arguments are given.  Once IOPT is set, it is always
! set.  If the user subsequently doesn't want to use optional arguments, he
! she simply sets all the optional arguments to 0.
IF (                      &
    PRESENT(MAXWORK) .OR. &
    PRESENT(IXPR   ) .OR. &
    PRESENT(MXHNIL ) .OR. &
    PRESENT(MXORDN ) .OR. &
    PRESENT(MXORDS ) .OR. &
    PRESENT(H0     ) .OR. &
    PRESENT(HMAX   ) .OR. &
    PRESENT(HMIN   )      &
) THEN
    IOPT_I = 1
END IF


! When NEQ or NG is set do the following:
!   Update LRW.
!   (Re)Allocate RWORK.
IF ( PRESENT(NEQ) .OR. PRESENT(NG) ) THEN
   IF ( ALLOCATED(RWORK_I) ) THEN ; DEALLOCATE(RWORK_I) ; END IF
   IF ( ALLOCATED(RWORK_ ) ) THEN ; DEALLOCATE(RWORK_ ) ; END IF
   LRW_I = 22 + NEQ_I * MAX(16, NEQ_I + 9) + 3 * NG_I
   ALLOCATE(RWORK_I(LRW_I), RWORK_(LRW_I))
   RWORK_I(1:LRW_I) = 0.0
END IF


! When NEQ is set do the following:
!   Update LIW.
!   (Re)Allocate IWORK and Y.
IF ( PRESENT(NEQ) ) THEN
   IF ( ALLOCATED(IWORK_I) ) THEN ; DEALLOCATE(IWORK_I) ; END IF
   IF ( ALLOCATED(IWORK_ ) ) THEN ; DEALLOCATE(IWORK_ ) ; END IF
   IF ( ALLOCATED(Y_I    ) ) THEN ; DEALLOCATE(Y_I    ) ; END IF
   IF ( ALLOCATED(Y_     ) ) THEN ; DEALLOCATE(Y_     ) ; END IF
   LIW_I = 20 + NEQ_I
   ALLOCATE(IWORK_I(LIW_I), IWORK_(LIW_I), Y_I(NEQ_I), Y_(NEQ_I))
   IWORK_I(1:LIW_I) = 0
   Y_I(    1:NEQ_I) = 0.0
END IF


! When NG is set do the following:
!   (Re)Allocate JROOT.
IF ( PRESENT(NG) ) THEN
   IF ( ALLOCATED(JROOT_I) ) THEN ; DEALLOCATE(JROOT_I) ; END IF
   IF ( ALLOCATED(JROOT_ ) ) THEN ; DEALLOCATE(JROOT_ ) ; END IF
   ALLOCATE(JROOT_I(NG_I), JROOT_(NG_I))
   JROOT_I(:) = 0.0_R8
   JROOT_ (:) = 0.0_R8
ELSE
   IF ( NG_I .EQ. 0 ) THEN
       IF ( ALLOCATED(JROOT_I) ) THEN ; DEALLOCATE(JROOT_I) ; END IF
       IF ( ALLOCATED(JROOT_ ) ) THEN ; DEALLOCATE(JROOT_ ) ; END IF
       ALLOCATE(JROOT_I(1), JROOT_(1))
```

```
            JROOT_I(:) = 0.0_R8
            JROOT_ (:) = 0.0_R8
        END IF
    END IF


    IF ( PRESENT(Y) .AND. ALLOCATED(Y_I) ) THEN ; Y_I(1:NEQ_I) = Y(1:NEQ_I) ; END IF



END SUBROUTINE LSODAR_SET




!-------------------------------------------------------------------------------
!
!  LSODAR_RUN
!
!  Runs LSODAR.  First this routine ensures the user has called LSODAR_SET and
!  set NEQ (the only variable required to be set by the user).  Then the routine
!  calls LSODAR and runs until TOUT.  The routine returns.
!
!  If PRINT_EVERY is set to a positive value then LSODAR is called with
!  TOUT=PRINT_EVERY then TOUT=2*PRINT_EVERY and so forth until the user set TOUT
!  is reached.  Between each call the time is printed and the variables are
!  printed in the order they appear in Y.  The output goes to FILE_UNIT
!  (defaults to stdout, unit 6).
!
!  In the event that LSODAR returns an error an appropriate message is printed
!  and the routine is returned.  The printed message corresponds to the error
!  flag returned and is taken from the LSODAR documentation.
!
!  In the event that LSODAR returns a root then this routine returns.  The
!  LSODAR_SIMULATE routine is more appropriate if a specific action (like cell
!  division) will be taken at the root and integration is to proceed.
!
```

```
SUBROUTINE LSODAR_RUN()

    ! Local variables
    REAL (KIND=R8) :: TFINAL
    REAL (KIND=R8) :: AUX_VARS(:)
    ALLOCATABLE :: AUX_VARS

    IF ( DEBUG_I ) THEN
        WRITE(*,*) "==========================================&
                   &================================"
        WRITE(*,*) "                        DEBUG REPORT FROM LSODAR_RUN"
        WRITE(*,*) "DEBUG = ", DEBUG_I
        WRITE(*,*) "NEQ = ", NEQ_I
        WRITE(*,*) "ITASK = ", ITASK_I
        WRITE(*,*) "ISTATE = ", ISTATE_I
        WRITE(*,*) "NG = ", NG_I
        WRITE(*,*) "MAXWORK = ", MAXWORK_I
        WRITE(*,*) "IXPR = ", IXPR_I
        WRITE(*,*) "MXHNIL = ", MXHNIL_I
        WRITE(*,*) "MXORDN = ", MXORDN_I
        WRITE(*,*) "MXORDS = ", MXORDS_I
        WRITE(*,*) "Y = ", Y_I
        WRITE(*,*) "T = ", T_I
        WRITE(*,*) "TOUT = ", TOUT_I
        WRITE(*,*) "RTOL = ", RTOL_I
        WRITE(*,*) "ATOL = ", ATOL_I
        WRITE(*,*) "H0 = ", H0_I
        WRITE(*,*) "HMAX = ", HMAX_I
        WRITE(*,*) "HMIN = ", HMIN_I
        WRITE(*,*) "PRINT_EVERY = ", PRINT_EVERY_I
        WRITE(*,*) "ROOT_TOL = ", ROOT_TOL_I
        WRITE(*,*) "ROOT_TTOL = ", ROOT_TTOL_I
        WRITE(*,*) "JROOT = ", JROOT_I
        WRITE(*,*) "FIND_ROOTS = ", FIND_ROOTS_I
        WRITE(*,*) "FILE_UNIT = ", FILE_UNIT_I
        WRITE(*,*) "NUM_AUX = ", NUM_AUX_I
        WRITE(*,*) "==========================================&
                   &================================"
    END IF
```

```
ALLOCATE( AUX_VARS(NUM_AUX_I) )

! Ensure the user has set NEQ.  This routine cannot call LSODAR if NEQ is
! not positive.  Setting NEQ allocates necessary variables for LSODAR.
IF ( NEQ_I .LE. 0 ) THEN
    WRITE(*,*) "ERROR: NEQ MUST BE SET BEFORE LSODAR_RUN IS CALLED"
    RETURN
END IF

! Set optional variables if IOPT
IF ( IOPT_I .EQ. 1 .AND. ( ISTATE_I .EQ. 1 ) ) THEN
    IWORK_I(6) = MAXWORK_I
    IWORK_I(5) = IXPR_I
    IWORK_I(7) = MXHNIL_I
    IWORK_I(8) = MXORDN_I
    IWORK_I(9) = MXORDS_I
    RWORK_I(5) = H0_I
    RWORK_I(6) = HMAX_I
    RWORK_I(7) = HMIN_I
END IF

IF ( PRINT_EVERY_I .GT. 0_R8 ) THEN
    ! Call LSODAR and print variables every PRINT_EVERY
    TFINAL = TOUT_I
    TOUT_I = PRINT_EVERY_I + T_I
    IF ( NUM_AUX_I .GT. 0 ) THEN
        CALL MODEL_AUXILIARY( NUM_AUX_I, AUX_VARS, NEQ_I, Y_I, T_I )
        WRITE(FILE_UNIT_I,*) T_I, Y_I, AUX_VARS
    ELSE
        WRITE(FILE_UNIT_I,*) T_I, Y_I
    END IF
    DO WHILE ( TOUT_I .LT. TFINAL .AND.                 &
               ( ISTATE_I .EQ. 1 .OR. ISTATE_I .EQ. 2 ) &
             )
        CALL RUN_I()
        IF ( NUM_AUX_I .GT. 0 ) THEN
            CALL MODEL_AUXILIARY( NUM_AUX_I, AUX_VARS, NEQ_I, Y_I, T_I )
            WRITE(FILE_UNIT_I,*) T_I, Y_I, AUX_VARS
        ELSE
            WRITE(FILE_UNIT_I,*) T_I, Y_I
        END IF
        TOUT_I = TOUT_I + PRINT_EVERY_I
    END DO
    IF ( ISTATE_I .EQ. 1 .OR. ISTATE_I .EQ. 2 ) THEN
        TOUT_I = TFINAL
        CALL RUN_I()
        IF ( NUM_AUX_I .GT. 0 ) THEN
            CALL MODEL_AUXILIARY( NUM_AUX_I, AUX_VARS, NEQ_I, Y_I, T_I )
            WRITE(FILE_UNIT_I,*) T_I, Y_I, AUX_VARS
        ELSE
            WRITE(FILE_UNIT_I,*) T_I, Y_I
        END IF
    END IF
ELSE
    ! Call LSODAR and return
    CALL RUN_I()
END IF

END SUBROUTINE LSODAR_RUN


!-----------------------------------------------------------------------------
!
!  LSODAR_SIMULATE
!
!  This routine is very similar to LSODAR_RUN.  It calls LSODAR_RUN to perform
!  most of its functions.  The only difference in behavior is when a root is
!  found.  This routine tells the MODEL_FOUND_ROOT subroutine that a root is
!  found, sets ISTATE=1 (restart integration), and continues the simulation.
!  These actions are repeated until TOUT is reached.
!
!  LSODAR_SIMULATE cannot tell if LSODAR_RUN returned because the user did not
!  set NEQ.  Therefore, NEQ is checked first, followed by an initial call to
!  LSODAR_RUN.  Then, ISTATE is checked.  If ISTATE == 3 then a root was found.
```

```
!  The arguments for MODEL_FOUND_ROOT are set up and MODEL_FOUND_ROOT is called.
!  If TOUT > T then ISTATE is set to 1 and the LSODAR_RUN is called again.  When
!  TOUT == T this routine returns.
!

SUBROUTINE LSODAR_SIMULATE()
    USE MODEL_MOD
    REAL (KIND=R8) :: TOUT_BACKUP
    INTEGER        :: FIND_ROOTS_BACKUP

    ! Ensure the user has set NEQ.  This routine cannot call LSODAR if NEQ is
    ! not positive.  Setting NEQ allocates necessary variables for LSODAR.
    IF ( NEQ_I .LE. 0 ) THEN
        WRITE(*,*) "ERROR: NEQ MUST BE SET BEFORE LSODAR_RUN IS CALLED"
        RETURN
    END IF

    TOUT_BACKUP = TOUT_I
    FIND_ROOTS_BACKUP = FIND_ROOTS_I

    ! Make initial call to simulator
    CALL LSODAR_RUN()

    ! Handle any found roots, and continue integration.  If ISTATE != 3 then no
    ! roots were found and this routine exits.
    ! If a root is found then the integrator is called with root finding off for
    ! ROOT_TTOL time steps.  Then the integrator proceeds normally.
    DO WHILE ( ISTATE_I .EQ. 3 .AND. T_I .LT. TOUT_I)
        CALL MODEL_FOUND_ROOT( NEQ_I, T_I, Y_I, NG_I, JROOT_I )
        ISTATE_I = 1
        ! Turn off root finding.
        FIND_ROOTS_I = 0
        TOUT_I = T_I + ROOT_TTOL_I
        IF ( TOUT_I .GT. TOUT_BACKUP ) THEN
            TOUT_I = TOUT_BACKUP
        END IF
        CALL LSODAR_RUN()
        ! Turn on root finding.
        FIND_ROOTS_I = FIND_ROOTS_BACKUP
        TOUT_I = TOUT_BACKUP
        IF ( ISTATE_I .EQ. 2 .AND. T_I .LT. TOUT_I ) THEN
            ISTATE_I = 1
            CALL LSODAR_RUN()
        END IF
    END DO

END SUBROUTINE LSODAR_SIMULATE



!-----------------------------------------------------------------------------
!
!  RUN
!
!  Runs LSODAR and prints an error if one exists.  Nothing more.  This routine
!  is for internal use only.
!

SUBROUTINE RUN_I()

    USE MODEL_MOD

    IF ( FIND_ROOTS_I .EQ. 0 ) THEN
        CALL lsodar(                                                    &
            MODEL_RHS, NEQ_I, Y_I, T_I, TOUT_I, ITOL_I, RTOL_I, ATOL_I,   &
            ITASK_I, ISTATE_I, IOPT_I, RWORK_I, LRW_I, IWORK_I, LIW_I,    &
            JEX_I, JT_I, GEX, 0, JROOT_I                                  &
        )
    ELSE
        CALL lsodar(                                                    &
            MODEL_RHS, NEQ_I, Y_I, T_I, TOUT_I, ITOL_I, RTOL_I, ATOL_I,   &
            ITASK_I, ISTATE_I, IOPT_I, RWORK_I, LRW_I, IWORK_I, LIW_I,    &
            JEX_I, JT_I, GEX, NG_I, JROOT_I                               &
        )
    END IF
```

```
  IF (ISTATE_I .EQ. -1) THEN
    WRITE(6,*) 'LSODAR: excess work done on this call (perhaps wrong jt)'
  ELSE IF (ISTATE_I .EQ. -2) THEN
    WRITE(6,*) 'LSODAR: excess accuracy requested (tolerances too small)'
  ELSE IF (ISTATE_I .EQ. -3) THEN
    WRITE(6,*) 'LSODAR: illegal input detected (see printed message).'
  ELSE IF (ISTATE_I .EQ. -4) THEN
    WRITE(6,*) 'LSODAR: repeated error test failures (check all inputs).'
  ELSE IF (ISTATE_I .EQ. -5) THEN
    WRITE(6,*) "LSODAR: repeated convergence failures (perhaps bad jacobian &
               &supplied or wrong choice of jt or tolerances)."
  ELSE IF (ISTATE_I .EQ. -6) THEN
    WRITE(6,*) 'LSODAR: error weight became zero during problem. (solution &
               &component i vanished, and atol or atol(i) = 0.)'
  ELSE IF (ISTATE_I .EQ. -7) THEN
    WRITE(6,*) 'LSODAR: work space insufficient to finish (see messages).'
  ELSE IF (ISTATE_I .LT. 0) THEN
    WRITE(6,*) 'LSODAR: some unknown error'
  END IF

END SUBROUTINE RUN_I
```

```
! Dummy Jacobian calculator for LSODAR.
SUBROUTINE JEX (NEQ, T, Y, ML, MU, PD, NRPD)

INTEGER          :: NRPD, NEQ, ML, MU
REAL (KIND=R8)   :: PD(NRPD,1), T, Y(NEQ)



RETURN
END SUBROUTINE JEX
```

```
! This subroutine is a wrapper for the MODEL_ROOTS routine.  It simply sets GOUT
! equal to zero in all cases where elements of GOUT are less than ROOT_TOL.
SUBROUTINE GEX( NEQ, T, Y, NG, GOUT )
    USE REAL_PRECISION
    USE MODEL_MOD
    INTEGER, INTENT(IN)         :: NEQ, NG
    REAL (KIND=R8), INTENT(IN)  :: T, Y(NEQ)
    REAL (KIND=R8), INTENT(OUT) :: GOUT(NG)

    ! local variables
    INTEGER :: I

    CALL MODEL_ROOTS( NEQ, T, Y, NG, GOUT )

    DO I = 1, NG
        IF ( ABS(GOUT(I)) .LE. ROOT_TOL_I ) THEN
            GOUT(I) = 0.0_R8
        END IF
    END DO

END SUBROUTINE GEX




END MODULE LSODAR_MOD
```

model_mod.f90

```
MODULE MODEL_MOD

!==============================================================================
!
!  MODULE MODEL_MOD
!
!  This module provides the interfaces to the user supplied routines used by
!  LSODAR_MOD, ODRPACK_MOD, and OBJ_FUNC_MOD.
!
!  MODEL_RHS    Provides the right hand sides of the ODES.  Stores the values of
!               the RHSs in YDOT in the same order as Y.
!
!               NEQ     Number of equations (size of Y and YDOT)
!               T       Time
!               Y       The variable values at the current time (T).
!               YDOT    The value of the RHSs.  The output of this routine.
!
!  MODEL_ROOTS  Provides evaluations of the root functions (GOUT).  When any of
!               these functions evaluates to zero a root has been found.  LSODAR
!               uses this subroutine to determine when such a case arises.
!
!               NEQ     The number of ODEs.
!               T       The current time.
!               Y       The variables of the ODEs.
!               NG      The number of root functions.
!               GOUT    The root functions (the only output variable).
!
!  MODEL_FOUND_ROOT  Is executed after a root has been found.  The roots found
!               are specified in ROOTS_AT.  The values of Y may be changed, but
!               contain the current values at the current time.  If the values
!               of Y are changed then the integrator is restarted with the new Y
!               as the initial condition.
!
!               NEQ     The number of ODEs.
!               T       The current time.
!               Y       The variables of the ODEs.  This variable can be
!                       modified.  The integrator would then restart with this
!                       variable as the initial condition.
!               NG      The number of root functions.
!               JROOT   An integer array.  0 means no root at this function.
!                       One mean root at this function.
!
!  MODEL_EXPERIMENT  Is executed to request simulation of an experiment.  The
!               XDATA for the experiment are passed in and the YDATA are
!               returned.  The parameters for the model are passed also.  The
!               YDATA may not be defined for the parameters (PARAM) given.  If
!               the experiment is not defined for PARAM then DEFINED is set to 0
!               where appropriate.
!
!               NP              The number of parameters.
!               PARAMS          The parameter values to use in this simulation.
!               EXP_NAME        The name of the experiment to simulate.
!               NX              The size of XDATA.
!               NY              The size of YDATA.
!               XDATA           The x data provided as input to the simulation.
!                               In the case of a time course simulation this
!                               would be each of the time points desired.
!               YDATA           The y data returned from the simulation.  In the
!                               case of time course simulations this would be
!                               the concentration of a protien.
!               DEFINED         Tells the caller whether y is defined for the
!                               given parameters (and x).  Each y in YDATA is
!                               defined or undefined independently.  If
!                               DEFINED(I) .EQ. 0 then YDATA(I) is not defined.
!                               If DEFINED(I) .EQ. 1 then YDATA(I) is defined.
!

INTERFACE

    SUBROUTINE MODEL_RHS ( NEQ, T, Y, YDOT )
        USE REAL_PRECISION
        INTEGER, INTENT(IN)         :: NEQ
        REAL (KIND=R8), INTENT(IN)  :: Y(NEQ), T
        REAL (KIND=R8), INTENT(OUT) :: YDOT(NEQ)
    END SUBROUTINE MODEL_RHS
```

```
    SUBROUTINE MODEL_ROOTS ( NEQ, T, Y, NG, GOUT )
        USE REAL_PRECISION
        INTEGER, INTENT(IN)         :: NEQ, NG
        REAL (KIND=R8), INTENT(IN)  :: T, Y(NEQ)
        REAL (KIND=R8), INTENT(OUT) :: GOUT(NG)
    END SUBROUTINE MODEL_ROOTS

    SUBROUTINE MODEL_FOUND_ROOT ( NEQ, T, Y, NG, JROOT )
        USE REAL_PRECISION
        INTEGER, INTENT(IN)           :: NEQ, NG, JROOT(NG)
        REAL (KIND=R8), INTENT(IN)    :: T
        REAL (KIND=R8), INTENT(INOUT) :: Y(NEQ)
    END SUBROUTINE MODEL_FOUND_ROOT

    SUBROUTINE MODEL_EXPERIMENT ( NP, PARAMS, EXP_NAME, &
                                  NX, NY, XDATA, YDATA, &
                                  DEFINED               &
    )
        USE REAL_PRECISION
        USE STRINGS
        INTEGER, INTENT(IN)            :: NP, NX, NY
        CHARACTER (MAXSTR), INTENT(IN) :: EXP_NAME
        REAL (KIND=R8), INTENT(IN)     :: PARAMS(NP), XDATA(NX)
        REAL (KIND=R8), INTENT(OUT)    :: YDATA(NY)
        INTEGER, INTENT(OUT)           :: DEFINED(NY)
    END SUBROUTINE MODEL_EXPERIMENT

    SUBROUTINE MODEL_AUXILIARY ( NA, AUX, NV, VARS, T )
        USE REAL_PRECISION
        INTEGER, INTENT(IN) :: NA, NV
        REAL (KIND=R8), INTENT(OUT) :: AUX(NA)
        REAL (KIND=R8), INTENT(IN)  :: VARS(NV), T
    END SUBROUTINE MODEL_AUXILIARY

END INTERFACE

END MODULE MODEL_MOD
```

obj_func_mod.f90

```
MODULE OBJ_FUNC_MOD
!==============================================================================
!
!  OBJ_FUNC_MOD
!
!  This module provides some general objective functions for use with VTDirect.
!  The module also provides a FCN implementation for ODRPACK.  (FCN is not an
!  objective function, but has similar functionality to the routines here.)
!  The user calls VTDirect passing it one of these objective functions, or a
!  user writen objective function (in the latter case this module is not used).
!
!  To use this module the user must call the OBJ_FUNC_SET routine.  The
!  OBJ_FUNC_SET routine must receive at least the experimental data and
!  experiment names as correspond to MODEL_EXPERIMENTS.  See variable
!  documentation below for a list of required (S) and optional (O) parameters.
!
!  Optionally the user may specify relative evaluation cost of each experiment.
!  If a parallel objective function is used then these costs will be used to
!  prioritize and distrubute work to the processors.
!
!  Optionally the user may specify parameters to be on a log scale or linear
!  scale with respect to VTDirect.  The scales may be mixed (some parameters on
!  linear while others on log).
!

USE REAL_PRECISION
USE STRINGS

PRIVATE

!------------------------------------------------------------------------------
!
!  VARIABLES
!
!  Variables used by the objective functions in this module.  The variables can
!  be set with the OBJ_FUNC_SET routine.
!
```

```
!  Internal variable names have "_I"s appended to the names.
!
! KEY
!
!   ->  IN
!  <-   OUT
!  <->  IN and OUT
!   S   Must be set before LSODAR can be called.
!   A   Must be allocated before LSODAR can be called.
!   O   Optional input.
!   R   Required for a specific objective function.
!
!-------------------------------------------------------------------------------
!  -> S  NEXP           default = none, must be set by user.
!                       Number of experiments.
!  -> SA NXDATA         default = none, must be set by the user.
!                       An integer array.  The number of X
!                       experimental data for each experiment.
!                       size: NEXP
!  -> SA NYDATA         default = none, must be set by the user.
!                       An integer array.  The number of Y
!                       experimental data for each experiment.
!                       size: NEXP
!  -> SA EXP_NAMES      default = none, must be set by the user.
!                       The names of each experiment.  Used when calling
!                       MODEL_EXPERIMENTS.  The name of the experiment to be
!                       simulated is passed.
!                       size: NEXP
!  -> SA EXP_XDATA      default = none, must be set by the user.
!                       A packed array of the X data.  The X data is
!                       independent.  The Y data is calculated from the X data.
!                       For the ith experiment the X data are in this array at
!                       locations sum(NXDATA(1:ith-1)) to
!                       sum(NXDATA(1:ith))-1.  The first and second experiments
!                       are special cases (no sum is involved).
!                       size: SUM(NXDATA)
!  -> OA EXP_YDATA      default = none, must be set by the user.
!                       A packed array of the Y data.  The data is dependent on
!                       the X data.  The data can be retrieved in the same manar
!                       as EXP_XDATA.
!                       size: SUM(NYDATA)
!  -> OA EXP_COSTS      default = 1s.
!                       Defines the relative cost to calculate a single data
!                       point for each experiment.  This array is only used when
!                       a parallel objective function is used.  The default (all
!                       1s) assumes each experiment takes the same ammount of
!                       time to calculate.  The costs correspond to the cost to
!                       calculate a single point (one x-y pair, or one y).
!                       size: SUM(NYDATA)
!  -> O  NP             default = none,
!                       The number of parameters.  NP must be present if log
!                       scales are used for any parameters.  If no log scales
!                       are used NP is ignored.
!  -> OA LOG_SCALE      default = all 0s.
!                       If present then the C(ith) parameter will be on a log
!                       scale if LOG_SCALE(ith) is 1 and linear otherwise.  NP,
!                       L, and U must be present if LOG_SCALE is present.
!                       size: NP
!  -> OA L              default = none
!                       The lower bound of the parameters.  L must be present
!                       if LOG_SCALE is present.  This is the same L passed to
!                       VTDirect.  Normally the objective function need not know
!                       L because the parameters are just passed on to
!                       MODEL_EXPERIMENTS.  With LOG_SCALE the parameters must
!                       be converted making knowledge of L necessary.  Also,
!                       ODRPACK_FCN uses this as bound on the parameters if
!                       L_DEF contains any values greater than 0.
!                       size: NP
!  -> OA U              default = none.
!                       The upper bound of the parameters.  Same requirements as
!                       L.
!                       size: NP
!  -> OA L_DEF          defaults = 0.
!                       L_DEF(I) > 0 means L(I) defines a lower bound on the
!                       parameters.  L_DEF(I) <= 0 means the Ith parameter has
!                       no lower bound.  This is only meaningful to ODRPACK_FCN.
!                       size: NP
!  -> OA U_DEF          defaults = 0.
```

```
!                    U_DEF(I) > 0 means U(I) defines an upper bound on the
!                    parameters.  U_DEF(I) <= 0 means the Ith parameter has
!                    no upper bound.  This is only meaningful to ODRPACK_FCN.
!                    size: NP
!  -> OA PARAMS      defaults = 0
!                    Initial parameter guesses and fixed parameter values.
!                    size: NP
!  -> OA EST_PARAMS  defaults = 1
!                    Values of 1 indicate parameters at EST_PARAMS(I) are to
!                    be estimated.  The parameters are fixed if the value is
!                    0 and behavior is undefined otherwise.
!                    size: NP
!  -> OA WX          defaults = 1.
!                    Weights of the X data.
!                    size: SUM(NXDATA)
!  -> OA WY          defaults = 1.
!                    Weights of the Y data.
!                    size: SUM(NYDATA)
!  -> O  PARALLEL    default = 0.
!                    0  Run in serial.
!                    1  Run in parallel if the objective function supports.
!                       An error is reported if an objective function is
!                       asked to run in parallel and cannot.
!  -> O  OD_DELTA    defaults = 0.
!                    OD_DELTA tells objective functions supporting orthogonal
!                    distances to start at X + OD_DELTA to
!                    find values of YDATA.  Sometimes YDATA
!                    is not defined for a given EXP_XDATA, this is the case
!                    where OD_DELTA is important.
!                    size: SUM(NXDATA)
!  -> O  OD_ON       default = .false.
!                    A value of .true. tells objective functions that support
!                    orthogonal distances to use orthogonal distances.  A
!                    value of .false. will make all objective functions use
!                    error in the y (dependent) data only.
!  -> OA LY          default = none
!                    LY is required by the descrete objective function.  It
!                    defines the lower bound for an acceptable YDATA.  LY(I)
!                    does not have to be set or defined if LY_DEF(I) .EQ. 0.
!                    size: SUM(NYDATA)
!  -> OA UY          default = none
!                    UY is required by the discrete objective function.  It
!                    defines the upper bound for an acceptable YDATA.  UY(I)
!                    does not have to be set or defined if UY_DEF(I) .EQ. 0.
!                    size: SUM(NYDATA)
!  -> OA LY_DEF      defaults = 0.
!                    If LY_DEF(I) .EQ. 1 then LY(I) specifies the lower bound
!                    for an acceptable YDATA(I).  If LY_DEF(I) .EQ. 0 then
!                    LY(I) is ignored and there is no lower bound for an
!                    acceptable YDATA(I).  Note if there is no lower or upper
!                    bound then YDATA(I) will be ignored in the objective
!                    function.  Also note that if LY_DEF and UY_DEF contain
!                    all 0s then the objective function will exit with an
!                    error.
!                    size: SUM(NYDATA)
!  -> OA UY_DEF      defaults = 0.
!                    If UY_DEF(I) .EQ. 1 then UY(I) specifies the upper bound
!                    for an acceptable YDATA(I).  If UY_DEF(I) .EQ. 0 then
!                    UY(I) is ignored and there is no upper bound for an
!                    acceptable YDATA(I).  Note if there is no lower or upper
!                    bound then YDATA(I) will be ignored in the objective
!                    function.  Also note that if LY_DEF and UY_DEF contain
!                    all 0s then the objective function will exit with an
!                    error.
!                    size: SUM(NYDATA)
!  -> OA ACCEPT_UNDEF defaults = 0.
!                    If ACCEPT_UNDEF(I) .EQ. 0 and YDATA(I) is undefined then
!                    the objective function is undefined.
!                    If ACCEPT_UNDEF(I) .EQ. 1 and YDATA(I) is undefined
!                    then the other YDATA in the same experiment are given
!                    the weight of YDATA(I).  If there are no other YDATA in
!                    the experiment then the behavior is the same as
!                    ACCEPT_UNDEF(I) .EQ. 2.
!                    If ACCEPT_UNDEF(I) .EQ. 2 and YDATA(I) is undefined then
!                    the weight for YDATA(I) is distributed to the YDATA for
!                    the rest of the model.
!                    If ACCEPT_UNDEF(I) .EQ. 3 and YDATA(I) is undefined then
!                    YDATA(I) is used anyway.  This assumes that the
```

```
!                       MODEL_EXPERIMENT function plugged in some acceptable
!                       value for YDATA(I) even though it could not calculate
!                       the value from the model.
!                       If ACCEPT_UNDEF(I) .EQ. 4 and YDATA(I) is undefined then
!                       YDATA(I) is assumed unacceptable and contributes to the
!                       objective function as if it were defined and
!                       unacceptable.  This value for ACCEPT_UNDEF is only valid
!                       for the discrete objective function.
!                       size: SUM(NYDATA)
!  -> O  OD_AFCTOL      default = -1, let DN2FB set the default
!                       The tolerance used when calculating the OD for the
!                       absolute function value change.
!  -> O  OD_RFCTOL      default = -1, let DN2FB set the default
!                       The tolerance used when calculating the OD for the
!                       relative function value change.
!  -> O  OD_XCTOL       default = -1, let DN2FB set the default
!                       The tolerance used when calculating the OD for the
!                       change in experimental data.
!
INTEGER        :: NEXP_I=-1, NP_I=-1, NXDATA_I(:), NYDATA_I(:), LOG_SCALE_I(:),&
                  PARALLEL_I=0, LY_DEF_I(:), UY_DEF_I(:), ACCEPT_UNDEF_I(:),   &
                  L_DEF_I(:), U_DEF_I(:), EST_PARAMS_I(:)
REAL (KIND=R8) :: EXP_XDATA_I(:), EXP_YDATA_I(:), EXP_COSTS_I(:), L_I(:),      &
                  U_I(:), WX_I(:), WY_I(:), OD_DELTA_I(:),                     &
                  LY_I(:), UY_I(:), PARAMS_I(:),                               &
                  OD_AFCTOL_I=-1, OD_RFCTOL_I=-1, OD_XCTOL_I=-1
CHARACTER (MAXSTR) :: EXP_NAMES_I(:)
LOGICAL        :: OD_ON_I=.FALSE.
ALLOCATABLE    :: NXDATA_I, NYDATA_I, LOG_SCALE_I, EXP_XDATA_I, EXP_YDATA_I,   &
                  EXP_COSTS_I, L_I, U_I, L_DEF_I, U_DEF_I, EST_PARAMS_I, WX_I, &
                  WY_I, LY_I, OD_DELTA_I,  &
                  UY_I, LY_DEF_I, UY_DEF_I, ACCEPT_UNDEF_I, EXP_NAMES_I,       &
                  PARAMS_I
!-----------------------------------------------------------------------------
```

```
!
!  Internal variables (user may never set, they are strictly for internal use )
!
!  NY   = SUM(NYDATA)
!  NX   = SUM(NXDATA)
!
INTEGER :: NY=0, NX=0
INTEGER :: GLOBAL_EXP_NUM, GLOBAL_DEFINED, GLOBAL_EXP_DATA_NUM
REAL (KIND=R8) :: GLOBAL_PARAMS(:)
ALLOCATABLE    :: GLOBAL_PARAMS
!  Variables to correct DN2FB which has an apparent bug.  It doesn't always
!  return the minimum, sometimes it just returns the last evaluated point.
REAL (KIND=R8) :: DN_BEST_X, DN_MIN_R
LOGICAL :: DN_FIRST
```

```
PUBLIC DISCRETE_OBJ_FUNC, CONTINUOUS_OBJ_FUNC, OBJ_FUNC_SET, ODRPACK_FCN, &
       CALC_DELTA
```

```
CONTAINS
```

```
!-----------------------------------------------------------------------------
!
!  OBJ_FUNC_SET
!
!  OBJ_FUNC_SET sets the variables used by the objective functions in this
!  module.  Variables are automatically allocated when the necessary information
```

```
!  is available.  Variables are automatically reallocated when information about
!  their size changes.
!
SUBROUTINE OBJ_FUNC_SET ( &
    NEXP,                 &
    NP,                   &
    NXDATA,               &
    NYDATA,               &
    LOG_SCALE,            &
    PARALLEL,             &
    LY_DEF,               &
    UY_DEF,               &
    ACCEPT_UNDEF,         &
    EXP_XDATA,            &
    EXP_YDATA,            &
    EXP_COSTS,            &
    L,                    &
    U,                    &
    L_DEF,                &
    U_DEF,                &
    EST_PARAMS,           &
    PARAMS,               &
    WX,                   &
    WY,                   &
    OD_DELTA,             &
    OD_ON,                &
    OD_AFCTOL,            &
    OD_RFCTOL,            &
    OD_XCTOL,             &
    LY,                   &
    UY,                   &
    EXP_NAMES             &
)
    INTEGER, OPTIONAL, INTENT(IN)        :: NEXP,                &
                                            NP,                  &
                                            NXDATA(:),           &
                                            NYDATA(:),           &
                                            LOG_SCALE(:),        &
                                            PARALLEL,            &
                                            LY_DEF(:),           &
                                            UY_DEF(:),           &
                                            ACCEPT_UNDEF(:),     &
                                            L_DEF(:),            &
                                            U_DEF(:),            &
                                            EST_PARAMS(:)

    REAL (KIND=R8), OPTIONAL, INTENT(IN) :: EXP_XDATA(:),        &
                                            EXP_YDATA(:),        &
                                            EXP_COSTS(:),        &
                                            L(:),                &
                                            U(:),                &
                                            WX(:),               &
                                            WY(:),               &
                                            OD_DELTA(:),         &
                                            OD_AFCTOL,           &
                                            OD_RFCTOL,           &
                                            OD_XCTOL,            &
                                            LY(:),               &
                                            UY(:),               &
                                            PARAMS(:)
    CHARACTER (MAXSTR), OPTIONAL         :: EXP_NAMES(:)
    LOGICAL, OPTIONAL                    :: OD_ON

    IF ( PRESENT(NEXP)      ) THEN ; NEXP_I      = NEXP      ; END IF
    IF ( PRESENT(NP)        ) THEN ; NP_I        = NP        ; END IF
    IF ( PRESENT(PARALLEL)  ) THEN ; PARALLEL_I  = PARALLEL  ; END IF
    IF ( PRESENT(OD_ON)     ) THEN ; OD_ON_I     = OD_ON     ; END IF
    IF ( PRESENT(OD_AFCTOL) ) THEN ; OD_AFCTOL_I = OD_AFCTOL ; END IF
    IF ( PRESENT(OD_RFCTOL) ) THEN ; OD_RFCTOL_I = OD_RFCTOL ; END IF
    IF ( PRESENT(OD_XCTOL)  ) THEN ; OD_XCTOL_I  = OD_XCTOL  ; END IF


    IF ( PRESENT(NEXP) ) THEN
        IF ( ALLOCATED(NXDATA_I)    ) THEN ; DEALLOCATE(NXDATA_I)    ; END IF
        IF ( ALLOCATED(NYDATA_I)    ) THEN ; DEALLOCATE(NYDATA_I)    ; END IF
```

```
    IF ( ALLOCATED(EXP_NAMES_I) ) THEN ; DEALLOCATE(EXP_NAMES_I) ; END IF
    ALLOCATE( NXDATA_I(NEXP_I), NYDATA_I(NEXP_I), EXP_NAMES_I(NEXP_I) )
    NXDATA_I(1:NEXP_I) = -1
    NYDATA_I(1:NEXP_I) = -1
END IF

IF ( PRESENT(NXDATA) ) THEN
    IF ( .NOT. ALLOCATED(NXDATA_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: NXDATA PRESENT BUT NOT ALLOCATED."
        RETURN
    END IF
    NXDATA_I(1:NEXP_I) = NXDATA(1:NEXP_I) ;
END IF

IF ( PRESENT(NYDATA) ) THEN
    IF ( .NOT. ALLOCATED(NYDATA_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: NYDATA PRESENT BUT NOT ALLOCATED."
        RETURN
    END IF
    NYDATA_I(1:NEXP_I) = NYDATA(1:NEXP_I) ;
END IF

IF ( PRESENT(EXP_NAMES) ) THEN
    IF ( .NOT. ALLOCATED(EXP_NAMES_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: EXP_NAMES PRESENT BUT NOT &
                   &ALLOCATED."
        RETURN
    END IF
    EXP_NAMES_I(1:NEXP_I) = EXP_NAMES(1:NEXP_I) ;
END IF




IF ( PRESENT(NP) ) THEN
    IF ( ALLOCATED(LOG_SCALE_I)  ) THEN ; DEALLOCATE(LOG_SCALE_I)  ; END IF
    IF ( ALLOCATED(L_I)          ) THEN ; DEALLOCATE(L_I)          ; END IF
    IF ( ALLOCATED(U_I)          ) THEN ; DEALLOCATE(U_I)          ; END IF
    IF ( ALLOCATED(L_DEF_I)      ) THEN ; DEALLOCATE(L_DEF_I)      ; END IF
    IF ( ALLOCATED(U_DEF_I)      ) THEN ; DEALLOCATE(U_DEF_I)      ; END IF
    IF ( ALLOCATED(PARAMS_I)     ) THEN ; DEALLOCATE(PARAMS_I)     ; END IF
    IF ( ALLOCATED(EST_PARAMS_I) ) THEN ; DEALLOCATE(EST_PARAMS_I) ; END IF
    IF ( ALLOCATED(GLOBAL_PARAMS)) THEN ; DEALLOCATE(GLOBAL_PARAMS); END IF
    ALLOCATE(LOG_SCALE_I(NP_I), L_I(NP_I), U_I(NP_I))
    ALLOCATE(L_DEF_I(NP_I), U_DEF_I(NP_I), PARAMS_I(NP_I), &
             EST_PARAMS_I(NP_I), GLOBAL_PARAMS(NP_I) )
    LOG_SCALE_I(1:NP_I) = 0
    L_I(1:NP_I) = -1.0_R8
    U_I(1:NP_I) = -1.0_R8
    L_DEF_I(1:NP_I) = 0
    U_DEF_I(1:NP_I) = 0
    EST_PARAMS_I(1:NP_I) = 1
    PARAMS_I(1:NP_I) = 0.0_R8
    GLOBAL_PARAMS(1:NP_I) = 0.0_R8
END IF

IF ( PRESENT(LOG_SCALE) ) THEN
    IF ( .NOT. ALLOCATED(LOG_SCALE_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: LOG_SCALE PRESENT BUT NOT &
                   &ALLOCATED."
        RETURN
    END IF
    LOG_SCALE_I(1:NP_I) = LOG_SCALE(1:NP_I) ;
END IF

IF ( PRESENT(L) ) THEN
    IF ( .NOT. ALLOCATED(L_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: L) PRESENT BUT NOT ALLOCATED."
        RETURN
    END IF
    L_I(1:NP_I) = L(1:NP_I);
END IF
IF ( PRESENT(U) ) THEN
    IF ( .NOT. ALLOCATED(U_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: U) PRESENT BUT NOT ALLOCATED."
        RETURN
    END IF
```

```
    U_I(1:NP_I) = U(1:NP_I);
END IF
IF ( PRESENT(L_DEF) ) THEN
    IF ( .NOT. ALLOCATED(L_DEF_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: L_DEF PRESENT BUT NOT ALLOCATED."
        RETURN
    END IF
    L_DEF_I(1:NP_I) = L_DEF(1:NP_I)
END IF
IF ( PRESENT(U_DEF) ) THEN
    IF ( .NOT. ALLOCATED(U_DEF_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: U_DEF PRESENT BUT NOT ALLOCATED."
        RETURN
    END IF
    U_DEF_I(1:NP_I) = U_DEF(1:NP_I)
END IF
IF ( PRESENT(PARAMS) ) THEN
    IF ( .NOT. ALLOCATED(PARAMS_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: PARAMS PRESENT BUT NOT ALLOCATED."
        RETURN
    END IF
    PARAMS_I(1:NP_I) = PARAMS(1:NP_I)
END IF
IF ( PRESENT(EST_PARAMS) ) THEN
    IF ( .NOT. ALLOCATED(EST_PARAMS_I) ) THEN
        WRITE(0,*) "ERROR: OBJ_FUNC_SET: EST_PARAMS PRESENT BUT NOT &
                   &ALLOCATED."
        RETURN
    END IF
    EST_PARAMS_I(1:NP_I) = EST_PARAMS(1:NP_I)
END IF



IF ( PRESENT(NYDATA) .AND. ALLOCATED(NYDATA_I) ) THEN
    NY = SUM(NYDATA_I(1:NEXP_I))
    IF ( ALLOCATED(EXP_YDATA_I)  ) THEN ; DEALLOCATE(EXP_YDATA_I)  ; END IF
    IF ( ALLOCATED(EXP_COSTS_I)  ) THEN ; DEALLOCATE(EXP_COSTS_I)  ; END IF
    IF ( ALLOCATED(WY_I)         ) THEN ; DEALLOCATE(WY_I)         ; END IF
    IF ( ALLOCATED(LY_I)         ) THEN ; DEALLOCATE(LY_I)         ; END IF
    IF ( ALLOCATED(UY_I)         ) THEN ; DEALLOCATE(UY_I)         ; END IF
    IF ( ALLOCATED(LY_DEF_I)     ) THEN ; DEALLOCATE(LY_DEF_I)     ; END IF
    IF ( ALLOCATED(UY_DEF_I)     ) THEN ; DEALLOCATE(UY_DEF_I)     ; END IF
    IF ( ALLOCATED(ACCEPT_UNDEF_I)) THEN; DEALLOCATE(ACCEPT_UNDEF_I);END IF
    ALLOCATE(EXP_YDATA_I(NY), EXP_COSTS_I(NY), WY_I(NY), LY_I(NY), &
             UY_I(NY), LY_DEF_I(NY), UY_DEF_I(NY), ACCEPT_UNDEF_I(NY))
    EXP_COSTS_I(1:NY)    = 1
    WY_I(1:NY)           = 1
    LY_DEF_I(1:NY)       = 0
    UY_DEF_I(1:NY)       = 0
    ACCEPT_UNDEF_I(1:NY) = 0
    !  These are not defaults, but instead make calculations work more
    !  efficiently if user does not use them.
    LY_I(1:NY)           = 0.0_R8
    UY_I(1:NY)           = 0.0_R8
END IF

IF ( PRESENT(LY_DEF) .AND. ALLOCATED(LY_DEF_I) ) THEN
    LY_DEF_I(1:NY) = LY_DEF(1:NY) ; END IF

IF ( PRESENT(UY_DEF) .AND. ALLOCATED(UY_DEF_I) ) THEN
    UY_DEF_I(1:NY) = UY_DEF(1:NY) ; END IF

IF ( PRESENT(ACCEPT_UNDEF) .AND. ALLOCATED(ACCEPT_UNDEF_I) ) THEN
    ACCEPT_UNDEF_I(1:NY) = ACCEPT_UNDEF(1:NY) ; END IF

IF ( PRESENT(EXP_YDATA) .AND. ALLOCATED(EXP_YDATA_I) ) THEN
    EXP_YDATA_I(1:NY) = EXP_YDATA(1:NY) ; END IF

IF ( PRESENT(EXP_COSTS) .AND. ALLOCATED(EXP_COSTS_I) ) THEN
    EXP_COSTS_I(1:NY) = EXP_COSTS(1:NY) ; END IF

IF ( PRESENT(WY) .AND. ALLOCATED(WY_I) ) THEN
    WY_I(1:NY) = WY(1:NY)
END IF
```

```
    IF ( PRESENT(LY) .AND. ALLOCATED(LY_I) ) THEN
        LY_I(1:NY) = LY(1:NY)
    END IF
    IF ( PRESENT(UY) .AND. ALLOCATED(UY_I) ) THEN
        UY_I(1:NY) = UY(1:NY)
    END IF




    IF ( PRESENT(NXDATA) .AND. ALLOCATED(NXDATA_I) ) THEN
        NX = SUM(NXDATA_I(1:NEXP_I))
        IF ( ALLOCATED(EXP_XDATA_I) ) THEN ; DEALLOCATE(EXP_XDATA_I) ; END IF
        IF ( ALLOCATED(WX_I)        ) THEN ; DEALLOCATE(WX_I)        ; END IF
        IF ( ALLOCATED(OD_DELTA_I)  ) THEN ; DEALLOCATE(OD_DELTA_I)  ; END IF
        ALLOCATE(EXP_XDATA_I(NX), WX_I(NX), OD_DELTA_I(NX) )
        WX_I(1:NX) = 1
        OD_DELTA_I(1:NX) = 0.0_R8
    END IF

    IF ( PRESENT(EXP_XDATA) .AND. ALLOCATED(EXP_XDATA_I) ) THEN
        EXP_XDATA_I(1:NX) = EXP_XDATA(1:NX) ; END IF

    IF ( PRESENT(WX) .AND. ALLOCATED(WX_I) ) THEN
        WX_I(1:NX) = WX(1:NX) ; END IF

    IF ( PRESENT(OD_DELTA) .AND. ALLOCATED(OD_DELTA_I) ) THEN
        OD_DELTA_I(1:NX) = OD_DELTA(1:NX) ; END IF

END SUBROUTINE OBJ_FUNC_SET




!-----------------------------------------------------------------------------
!
```

```
!  OBJ_FUNC_GET
!
!  Provides external access to a few internal variables.
!

SUBROUTINE OBJ_FUNC_GET( PARAMS, EST_PARAMS )
    INTEGER, INTENT(OUT), OPTIONAL :: EST_PARAMS(:)
    REAL (KIND=R8), INTENT(OUT), OPTIONAL :: PARAMS(:)

    IF ( PRESENT(EST_PARAMS) ) THEN
        EST_PARAMS(1:NP_I) = EST_PARAMS_I(1:NP_I)
    END IF
    IF ( PRESENT(PARAMS) ) THEN
        PARAMS(1:NP_I) = PARAMS_I(1:NP_I)
    END IF

END SUBROUTINE OBJ_FUNC_GET




!-----------------------------------------------------------------------------
!
!  DISCRETE_OBJ_FUNC
!
!  Calls MODEL_EXPERIMENT for all experiments in EXP_NAMES.  All the results
!  from MODEL_EXPERIMENT are mapped to 0 or 1 (0 for acceptable and 1 for
!  unacceptable).  The weights WX and WY are then applied (multiplied) to the 0s
!  and 1s and the results are summed.  The sum is returned in F.
!
!  The mapping to 0 and 1 is performed according to LY and UY.  LY provides a
!  lower bound for acceptibility of experiments and UY provides an upper bound
!  for acceptibility of experiments.  Some experiments may have only an LY or an
!  UY.  Whether an experiment has a lower or upper bound is defined in LY_DEF
!  and UY_DEF, respectively.
!
!  Variables that must be defined for this function:
!      NEXP
```

```
!       NXDATA
!       NYDATA
!       EXP_NAMES
!       EXP_XDATA
!       LY, UY, LY_DEF, UY_DEF
!
!  Variables that are used by this function if user sets them:
!       EXP_COSTS
!       NP
!       LOG_SCALE
!       L, U
!       WX, WY
!       PARALLEL
!       ACCEPT_UNDEF
!
!  NOTE: Not all of LY, UY, LY_DEF, and UY_DEF must be defined.  The are
!        dependent on each other and could leave simulated experiments as always
!        acceptable.  The code does not check for this case.  The user therefore
!        has the ability to ignore an experiment by setting UY_DEF(I) and
!        LY_DEF(I) equal to 0 affectively saying the experiment is always
!        acceptable.
!

FUNCTION DISCRETE_OBJ_FUNC(C, IFLAG) RESULT(F)
    USE REAL_PRECISION, ONLY : R8
    REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: C
    INTEGER, INTENT(OUT) :: IFLAG
    REAL(KIND = R8) :: F

    ! Local variables
    ! EXP_WEIGHT is the left over weight from undefined data that will be
    ! applied to the current experiment.
    ! MODEL_WEIGHT is the left over weight from undefined data that will be
    ! applied to the whole "model" (all the experimental data).
    INTEGER :: EXP_NUM, EXP_BEGIN, EXP_END, EXP_DATA_NUM
    REAL (KIND=R8) :: YDATA(NY), EXP_WEIGHT, MODEL_WEIGHT, EXP_F, Y
    INTEGER :: DEFINED(NY)

    IFLAG = 0

    IF ( .NOT. NEXP_I .GT. 0 ) THEN
        WRITE(0,*) "NEXP IS NOT GREATER THAN 0"
        STOP
    END IF

    IF ( ANY(NXDATA_I .LT. 0 ) ) THEN
        WRITE(0,*) "ONE OR MORE ELEMENTS IN NXDATA IS LESS THAN 0"
        STOP
    END IF

    IF ( ANY(NYDATA_I .LE. 0 ) ) THEN
        WRITE(0,*) "ONE OR MORE ELEMENTS IN NYDATA IS LESS THAN OR EQUAL TO 0"
        STOP
    END IF

    MODEL_WEIGHT = 0.0_R8
    F = 0.0_R8

    DO EXP_NUM = 1, NEXP_I

        EXP_BEGIN = SUM(NYDATA_I(1:EXP_NUM-1))+1
        EXP_END   = SUM(NYDATA_I(1:EXP_NUM))

        CALL MODEL_EXPERIMENT(                                          &
            SIZE(C),                                                    &
            C,                                                          &
            EXP_NAMES_I(EXP_NUM),                                       &
            NXDATA_I(EXP_NUM),                                          &
            NYDATA_I(EXP_NUM),                                          &
            EXP_XDATA_I(                                                &
                SUM(NXDATA_I(1:EXP_NUM-1))+1                            &
                : SUM(NXDATA_I(1:EXP_NUM))                              &
            ),                                                          &
            YDATA(EXP_BEGIN:EXP_END),                                   &
            DEFINED(EXP_BEGIN:EXP_END)                                  &
        )
```

```
!       WRITE(*,*) "    ", LY_I(EXP_BEGIN), " .LE. ",    &
!                          YDATA(EXP_BEGIN), " .LE. ", &
!                          UY_I(EXP_BEGIN)

        EXP_WEIGHT = 0.0_R8
        EXP_F      = 0.0_R8

        DO EXP_DATA_NUM = EXP_BEGIN, EXP_END
            Y = YDATA(EXP_DATA_NUM)
            IF ( DEFINED(EXP_DATA_NUM) .EQ. 1        &
                 .OR.                                &
                 ACCEPT_UNDEF_I(EXP_DATA_NUM) .EQ. 3 &
            ) THEN
                IF ( .NOT. (                                  &
                     ( Y-LY_I(EXP_DATA_NUM) .GE. 0.0_R8       &
                       .OR. LY_DEF_I(EXP_DATA_NUM) .EQ. 0     &
                     ) .AND.                                  &
                     ( Y-UY_I(EXP_DATA_NUM) .LE. 0.0_R8       &
                       .OR. UY_DEF_I(EXP_DATA_NUM) .EQ. 0     &
                     )                                        &
                   ) ) THEN
                    EXP_F = EXP_F + WY_I(EXP_DATA_NUM)
                END IF
            ELSE
                IF ( ACCEPT_UNDEF_I(EXP_DATA_NUM) .EQ. 1) THEN
                    IF ( EXP_END-EXP_BEGIN .EQ. 0 ) THEN
                        MODEL_WEIGHT = MODEL_WEIGHT + WY_I(EXP_DATA_NUM)
                    ELSE
                        EXP_WEIGHT = EXP_WEIGHT + WY_I(EXP_DATA_NUM)
                    END IF
                ELSEIF ( ACCEPT_UNDEF_I(EXP_DATA_NUM) .EQ. 2) THEN
                    MODEL_WEIGHT = MODEL_WEIGHT + WY_I(EXP_DATA_NUM)
                ELSEIF ( ACCEPT_UNDEF_I(EXP_DATA_NUM) .EQ. 4) THEN
                    EXP_F = EXP_F + WY_I(EXP_DATA_NUM)
                ELSE
                    IFLAG = 1
                    RETURN
                END IF
            END IF
        END DO

!       IF ( EXP_F .GT. 0.0_R8 ) THEN
!           WRITE(*,*) "WRONG     : ", TRIM(EXP_NAMES_I(EXP_NUM))
!       END IF
!       IF ( EXP_WEIGHT .GT. 0.0_R8 ) THEN
!           WRITE(*,*) "UNDEFINED : ", TRIM(EXP_NAMES_I(EXP_NUM))
!       END IF

        F = F + EXP_F * SUM(WY_I(EXP_BEGIN:EXP_END))                  &
                        / (SUM(WY_I(EXP_BEGIN:EXP_END)) - EXP_WEIGHT)

!       WRITE(*,*) "    F = ", F

    END DO

!   WRITE(*,*) "UNDEFINED : ", MODEL_WEIGHT
!   WRITE(*,*) "WRONG     : ", F

    F = F * SUM(WY_I(1:NY)) / (SUM(WY_I(1:NY)) - MODEL_WEIGHT)

END FUNCTION DISCRETE_OBJ_FUNC



!-------------------------------------------------------------------------------
!
!  CONTINUOUS_OBJ_FUNC
!
!  This routine needs some work.  It doesn't handle undefined experiments
!  appropriately.
!
!  Calls MODEL_EXPERIMENT for all experiments in EXP_NAMES.  All the results
!  from MODEL_EXPERIMENT are squared.  The weights WX and WY are then applied
!  (multiplied) to the squares and the results are summed.  The sum is returned
!  in F.
```

```
!
!  This routine supports orthogonal distances.  If OD_ON is true it will
!  calculate the orthogonal distance for every dependent/independent data pair
!  using LMDIF1.  The routine wants to start with defined data and will try
!  three points in this order: XDATA, XDATA+OD_DELTA, XDATA-OD_DELTA.  If the
!  data is undefined at all three points then the data is set as undefined with
!  the values returned at XDATA.
!
!  Variables that must be defined for this function:
!       NEXP
!       NXDATA
!       NYDATA
!       EXP_NAMES
!       EXP_XDATA
!
!  Variables that are used by this function if user sets them:
!       EXP_COSTS
!       NP
!       LOG_SCALE
!       L, U
!       WX, WY
!       PARALLEL
!       ACCEPT_UNDEF
!       OD_ON
!       OD_DELTA
!

FUNCTION CONTINUOUS_OBJ_FUNC(C, IFLAG) RESULT(F)
    USE REAL_PRECISION, ONLY : R8
    REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: C
    INTEGER, INTENT(OUT) :: IFLAG
    REAL(KIND = R8) :: F

    ! Local variables
    ! EXP_WEIGHT is the left over weight from undefined data that will be
    ! applied to the current experiment.
    ! MODEL_WEIGHT is the left over weight from undefined data that will be
    ! applied to the whole "model" (all the experimental data).
    ! WERR is a temporary array containing the weighted error for the
    ! experiments.  It is the size of the total number of YDATA
    INTEGER :: EXP_NUM, EXP_BEGIN, EXP_END, EXP_DATA_NUM
    REAL (KIND=R8) :: YDATA(NY), EXP_WEIGHT, MODEL_WEIGHT, EXP_F, Y
    INTEGER :: DEFINED(NY)
    REAL (KIND=R8) :: WERR(NY)

    IFLAG = 0

    IF ( .NOT. NEXP_I .GT. 0 ) THEN
        WRITE(0,*) "NEXP IS NOT GREATER THAN 0"
        STOP
    END IF

    IF ( ANY(NXDATA_I .LT. 0 ) ) THEN
        WRITE(0,*) "ONE OR MORE ELEMENTS IN NXDATA IS LESS THAN 0"
        STOP
    END IF

    IF ( ANY(NYDATA_I .LE. 0 ) ) THEN
        WRITE(0,*) "ONE OR MORE ELEMENTS IN NYDATA IS LESS THAN OR EQUAL TO 0"
        STOP
    END IF

    MODEL_WEIGHT = 0.0_R8
    F = 0.0_R8

!   WRITE(*,*) "INFO: PARAMS: ", C(:)

    DO EXP_NUM = 1, NEXP_I

        EXP_BEGIN = SUM(NYDATA_I(1:EXP_NUM-1))+1
        EXP_END   = SUM(NYDATA_I(1:EXP_NUM))

        IF ( OD_ON_I ) THEN

            CALL ORTHOGONAL_DISTANCE(                                       &
                C(1:NP_I), EXP_NUM, DEFINED(1:NY), WERR(1:NY)               &
            )
```

```
ELSE

    CALL NORMAL_DISTANCE(                                          &
         C(1:NP_I), EXP_NUM, DEFINED(1:NY), WERR(1:NY)             &
    )

END IF

EXP_WEIGHT = 0.0_R8
EXP_F      = 0.0_R8

DO EXP_DATA_NUM = EXP_BEGIN, EXP_END
    IF ( DEFINED(EXP_DATA_NUM) .EQ. 1        &
         .OR.                                &
         ACCEPT_UNDEF_I(EXP_DATA_NUM) .EQ. 3 &
    ) THEN
        EXP_F = EXP_F + WERR(EXP_DATA_NUM)
    ELSE
        WRITE(0,*) "EXP ", TRIM(EXP_NAMES_I(EXP_NUM)), " NOT DEFINED."
        WRITE(0,*) "WARNING: UNDEFINED EXPERIMENTS NOT CORRECTLY ", &
                   "HANDLED."
        IF ( ACCEPT_UNDEF_I(EXP_DATA_NUM) .EQ. 1) THEN
            IF ( EXP_END-EXP_BEGIN .EQ. 0 ) THEN
                MODEL_WEIGHT = MODEL_WEIGHT + WERR(EXP_DATA_NUM)
            ELSE
                EXP_WEIGHT = EXP_WEIGHT + WERR(EXP_DATA_NUM)
            END IF
        ELSEIF ( ACCEPT_UNDEF_I(EXP_DATA_NUM) .EQ. 2) THEN
            MODEL_WEIGHT = MODEL_WEIGHT + WERR(EXP_DATA_NUM)
        ELSE
            IFLAG = 1
            WRITE(0,*) "WARNING: UNDEFINED EXPERIMENT MADE OBJ UNDEFINED."
            RETURN
        END IF
    END IF
END DO

IF ( EXP_WEIGHT .GT. 0.0_R8 ) THEN
            WRITE(0,*) "INFO: UNDEFINED : ", TRIM(EXP_NAMES_I(EXP_NUM))
        END IF

        F = F + EXP_F

    END DO

    WRITE(*,*) "F C : ", F, C(:)

END FUNCTION CONTINUOUS_OBJ_FUNC




SUBROUTINE NORMAL_DISTANCE( C, EXP_NUM, DEFINED, WERR )
    REAL (KIND=R8) :: C(NP_I), WERR(NY)
    INTEGER :: EXP_NUM, DEFINED(NY)

    !  Local variables
    INTEGER :: EXP_BEGIN, EXP_END, EXP_DATA_NUM
    REAL (KIND=R8) :: YDATA(NY)

    EXP_BEGIN = SUM(NYDATA_I(1:EXP_NUM-1))+1
    EXP_END   = SUM(NYDATA_I(1:EXP_NUM))
    WERR(EXP_BEGIN:EXP_END) = 0.0_R8
    DEFINED(EXP_BEGIN:EXP_END) = 0

    CALL MODEL_EXPERIMENT(                                         &
        SIZE(C),                                                   &
        C,                                                         &
        EXP_NAMES_I(EXP_NUM),                                      &
        NXDATA_I(EXP_NUM),                                         &
        NYDATA_I(EXP_NUM),                                         &
        EXP_XDATA_I(                                               &
            SUM(NXDATA_I(1:EXP_NUM-1))+1                           &
            : SUM(NXDATA_I(1:EXP_NUM))                             &
        ),                                                         &
```

```
        YDATA(EXP_BEGIN:EXP_END),                                   &
        DEFINED(EXP_BEGIN:EXP_END)                                  &
    )
    WERR(EXP_BEGIN:EXP_END) = (                                     &
            YDATA(EXP_BEGIN:EXP_END) - EXP_YDATA_I(EXP_BEGIN:EXP_END) &
        ) **2 * WY_I(EXP_BEGIN:EXP_END)

END SUBROUTINE NORMAL_DISTANCE




!-----------------------------------------------------------------------------
!
!  CALC_DELTA
!
!  Calculates the DELTA for X+DELTA that yields the orthogonal distance.  Sets
!  DEFINED for each experiment according to whether the experiment is defined
!  for the set of parameters.
!
SUBROUTINE CALC_DELTA( C, DELTA, DEFINED, WERR )
    REAL (KIND=R8), INTENT(IN)  :: C(NP_I)
    REAL (KIND=R8), INTENT(OUT) :: DELTA(NY)
    REAL (KIND=R8), INTENT(OUT), OPTIONAL :: WERR(NY)
    INTEGER :: DEFINED(NY)

    ! Local variables
    INTEGER :: EXP_NUM

    ! Variables to remove after debuging
    REAL (KIND=R8) :: YOUT(NY), RWORK(NY), RTEMP(NY)
    INTEGER :: I

    DELTA(1:NY)   = 0.0_R8
    DEFINED(1:NY) = 0
    WERR(1:NY) = 0.0_R8

    DO EXP_NUM = 1, NEXP_I
        CALL ORTHOGONAL_DISTANCE( C, EXP_NUM, DEFINED, WERR, DELTA )
    END DO

!               YOUT(:) = 0
!               RWORK(1:NY) = DELTA(1:NY)
!               !WRITE(*,*) "PARAMS I: ", C(1:NP_I)
!               DO I = 1, 1 !NEXP_I
!                   CALL MODEL_EXPERIMENT( &
!                       NP_I, C, EXP_NAMES_I(I),                    &
!                       1,1, EXP_XDATA_I(1)+RWORK(1), YOUT(1), DEFINED(1)&
!!                      NXDATA_I(I), NYDATA_I(I),                       &
!!                      EXP_XDATA_I(                                        &
!!                          SUM(NXDATA_I(1:I))-NXDATA_I(I)+1:             &
!!                          SUM(NXDATA_I(1:I))                              &
!!                      ) +                                                 &
!!                      RWORK(                                              &
!!                          SUM(NXDATA_I(1:I))-NXDATA_I(I)+1:             &
!!                          SUM(NXDATA_I(1:I))                              &
!!                      ),                                                  &
!!                      YOUT(                                               &
!!                          SUM(NYDATA_I(1:I))-NYDATA_I(I)+1:             &
!!                          SUM(NYDATA_I(1:I))                              &
!!                      ),                                                  &
!!                      DEFINED(                                            &
!!                          SUM(NYDATA_I(1:I))-NYDATA_I(I)+1:             &
!!                          SUM(NYDATA_I(1:I))                              &
!!                      )                                                   &
!                   )
!               END DO
!               RTEMP(1) = RWORK(1)**2 * WX_I(1) + &
!                                   (EXP_YDATA_I(1) - YOUT(1))**2 * WY_I(1)
!!               RTEMP(1:NY) = RWORK(1:NX)**2 * WX_I(1:NX) + &
!!                                    YOUT(1:NY)**2 * WY_I(1:NY)
!               WRITE(*,*) "-------------------------------------------"
!               WRITE(*,*) "X + DELTA : ", EXP_XDATA_I(1) + RWORK(1)
!               WRITE(*,*) "YOUT : ", YOUT(1)
!!               WRITE(*,*) "WSOS I: ", SUM(RTEMP(1:NY))
!               WRITE(*,*) "RESIDUALS: ", RTEMP(1)
```

```
!                WRITE(*,*) "RESIDUALS - WERR : ", RTEMP(1) - WERR(1)

!                WRITE(*,*) "X + DELTA : ", EXP_XDATA_I(1:NX) + RWORK(1:NX)
!                WRITE(*,*) "YOUT : ", YOUT(1:NY)
!                WRITE(*,*) "WSOS I: ", SUM(RTEMP(1:NY))
!                WRITE(*,*) "RESIDUALS: ", RTEMP(1:NY)
!                WRITE(*,*) "RESIDUALS - WERR : ", RTEMP(1:NY) - WERR(1:NY)

END SUBROUTINE




!-------------------------------------------------------------------------------
!
!  ORTHOGONAL_DISTANCE
!
!  Calculates the weighted orthogonal distance between experiment and simulation
!  for experiment EXP_NUM with parameters C.  Returns the weighted orthogonal
!  distance in WERR and sets DEFINED according to the results from LMDIF.
!
SUBROUTINE ORTHOGONAL_DISTANCE( C, EXP_NUM, DEFINED, WERR, DELTA )
    REAL (KIND=R8), INTENT(IN)  :: C(NP_I)
    REAL (KIND=R8), INTENT(INOUT) :: WERR(NY)
    REAL (KIND=R8), INTENT(INOUT), OPTIONAL :: DELTA(NY)
    INTEGER, INTENT(IN)  :: EXP_NUM
    INTEGER, INTENT(INOUT) :: DEFINED(NY)

    !  Parameters to DN2FB
    INTEGER, PARAMETER :: DN_N_P = 1, DN_P_P = 1, DN_LIV_P = 4*DN_P_P + 82, &
        DN_LV_P = 105 + DN_P_P*(DN_N_P + 2*DN_P_P + 21) + 2*DN_N_P
    INTEGER :: DN_N=DN_N_P, DN_P=DN_P_P, DN_LIV=DN_LIV_P, DN_LV=DN_LV_P
    INTEGER :: DN_IV(DN_LIV_P), DN_UI(1)
    REAL (KIND=R8) :: DN_X(DN_P_P), DN_B(2,DN_P_P), DN_V(DN_LV_P), DN_UR(1)

    !  Local variables
    INTEGER :: EXP_BEGIN, EXP_END, EXP_DATA_NUM
    REAL (KIND=R8) :: DELTA_I(NY)

    EXP_BEGIN = SUM(NYDATA_I(1:EXP_NUM-1))+1
    EXP_END   = SUM(NYDATA_I(1:EXP_NUM))
    WERR(EXP_BEGIN:EXP_END) = 0.0_R8
    DEFINED(EXP_BEGIN:EXP_END) = 0

    GLOBAL_EXP_NUM = EXP_NUM

    DO EXP_DATA_NUM = EXP_BEGIN, EXP_END
!       WRITE(*,*) "CALCULATING OD FOR ", EXP_NUM, EXP_DATA_NUM
        GLOBAL_DEFINED = 0
        GLOBAL_PARAMS(1:NP_I) = C(1:NP_I)
        GLOBAL_EXP_DATA_NUM = EXP_DATA_NUM
        IF ( WX_I(EXP_DATA_NUM) .EQ. 0.0_R8 ) THEN
            CALL NORMAL_DISTANCE( C, EXP_NUM, DEFINED, WERR )
        ELSE
            !  Set DN2FB params
            DN_X(1) = EXP_XDATA_I(EXP_DATA_NUM) + OD_DELTA_I(EXP_DATA_NUM)
            DN_B(1,:) = 0.0_R8
            DN_B(2,:) = HUGE( 0.0_R8 )
            CALL DIVSET( 1, DN_IV, DN_LIV, DN_LV, DN_V )
            DN_IV(21) = 0 ! Don't print anything.
            IF ( OD_AFCTOL_I .GT. 0.0_R8 ) THEN
                DN_V(31) = OD_AFCTOL_I
            END IF
            IF ( OD_RFCTOL_I .GT. 0.0_R8 ) THEN
                DN_V(32) = OD_RFCTOL_I
            END IF
            IF ( OD_XCTOL_I .GT. 0.0_R8 ) THEN
                DN_V(33) = OD_XCTOL_I
            END IF
            DN_FIRST = .TRUE.
            CALL DN2FB( &
                DN_N, DN_P, DN_X, DN_B, DN_CALCR, DN_IV, DN_LIV, DN_LV, &
                DN_V, DN_UI, DN_UR, DN_UF &
            )
!            IF ( GLOBAL_EXP_DATA_NUM .EQ. 1 ) THEN
!           WRITE(*,*) "FIN: X = ", DN_BEST_X, " WSOS = ", DN_MIN_R
```

```
!            END IF
            WERR(EXP_DATA_NUM) = DN_MIN_R
            DELTA_I(EXP_DATA_NUM) = DN_BEST_X - EXP_XDATA_I(EXP_DATA_NUM)
            !WRITE(*,*) "DELTA, X_PLUS_D : ", DELTA_I(EXP_DATA_NUM), &
            !    DN_X(1)
            IF ( DN_IV(1) .EQ. 8 ) THEN
                WRITE(0,*) "DN2FB RETURNED FALSE CONVERGENCE."
                WRITE(0,*) "EXPERIMENT: ", TRIM(EXP_NAMES_I(EXP_NUM))
                WRITE(0,*) "EXPERIMENT DATA NUM: ", EXP_DATA_NUM
                WRITE(0,*) "EXP DATA: ", DN_X(:)
            END IF
            IF ( DN_IV(1) .NE. 3 .AND.  & ! DN_X convergence.
                 DN_IV(1) .NE. 4 .AND.  & ! Relative function convergence.
                 DN_IV(1) .NE. 5 .AND.  & ! Both converge.
                 DN_IV(1) .NE. 6 .AND.  & ! Absolute function convergence.
                 DN_IV(1) .NE. 7 .AND.  & ! Singular convergence.
                 DN_IV(1) .NE. 8 )      & ! False convergence. should be error.
            THEN
                WRITE(0,*) "DN2FB FAILED WITH CODE ", DN_IV(1)
                WRITE(0,*) "EXP DATA: ", DN_X(:)
                DEFINED(EXP_DATA_NUM) = 0
            ELSE
                DEFINED(EXP_DATA_NUM) = GLOBAL_DEFINED
            END IF
        END IF
    END DO

    IF ( PRESENT(DELTA) ) THEN
        DELTA(EXP_BEGIN:EXP_END) = DELTA_I(EXP_BEGIN:EXP_END)
    END IF

END SUBROUTINE ORTHOGONAL_DISTANCE



!-----------------------------------------------------------------------------
!
```

```
!  DN_CALCR
!
!  Calculates the residuals for DN_X.  Calls the MODEL_EXPERIMENT with DN_X as
!  the X data and evaluates the ODRPACK like objective function.
!
SUBROUTINE DN_CALCR( DN_N, DN_P, DN_X, DN_NF, DN_R, DN_UI, DN_UR, DN_UF )
    USE REAL_PRECISION

    ! Arguments to DN_CALCR
    INTEGER, INTENT(IN)    :: DN_N, DN_P
    INTEGER, INTENT(INOUT) :: DN_NF, DN_UI(1)
    REAL (KIND=R8), INTENT(IN)    :: DN_X(DN_P)
    REAL (KIND=R8), INTENT(INOUT) :: DN_R(DN_N), DN_UR(1)



    ! Local variables
    REAL (KIND=R8) :: YOUT

!   WRITE(*,*) "DN_NF: ", DN_NF, "  DN_X: ", DN_X(1)

    CALL MODEL_EXPERIMENT(                                          &
        NP_I,                                                       &
        GLOBAL_PARAMS(1:NP_I),                                      &
        EXP_NAMES_I(GLOBAL_EXP_NUM),                                &
        1, 1,                                                       &
        DN_X,                                                       &
        YOUT,                                                       &
        GLOBAL_DEFINED                                              &
    )

    DN_R(1) =                                                     &
        ( EXP_XDATA_I(GLOBAL_EXP_DATA_NUM) - DN_X(1) ) **2        &
         * WX_I(GLOBAL_EXP_DATA_NUM) +                            &
        ( EXP_YDATA_I(GLOBAL_EXP_DATA_NUM) - YOUT ) **2           &
         * WY_I(GLOBAL_EXP_DATA_NUM)

    IF ( DN_FIRST ) THEN
        DN_FIRST  = .FALSE.
```

```
        DN_BEST_X = DN_X(1)
        DN_MIN_R  = DN_R(1)
    ELSE
        IF ( DN_R(1) .LE. DN_MIN_R ) THEN
            DN_BEST_X = DN_X(1)
            DN_MIN_R  = DN_R(1)
        END IF
    END IF

!   WRITE(*,*) "PARAMS FOR FOLLOWING : ", GLOBAL_PARAMS(1:NP_I)
!    IF ( GLOBAL_EXP_DATA_NUM .EQ. 1 ) THEN
!    WRITE(*,*) GLOBAL_EXP_DATA_NUM, " : X = ", DN_X(1), " WSOS = ", DN_R(1), &
!       " Y = ", YOUT
!    END IF

END SUBROUTINE




!-----------------------------------------------------------------------------
!
!  DN_UF
!
!  A dummy routine for DN2FB.  This is a user function to do whatever the user
!  wants and is passed to DN_CALCR for use.  It is not needed for this code an
!  therefore does nothing.
!
SUBROUTINE DN_UF()
END SUBROUTINE DN_UF




SUBROUTINE ODRPACK_FCN( &
    N, M, NP, NQ, LDN, LDM, LDNP, BETA, XPLUSD, IFIXB, IFIXX, LDIFX, &
    IDEVAL, F, FJACB, FJACD, ISTOP                                   &
)

    INTEGER,        INTENT(IN)  :: N, M, NP, NQ, LDN, LDM, LDNP, IFIXB(NP), &
                                   IFIXX(LDIFX,M), LDIFX, IDEVAL
    REAL (KIND=R8), INTENT(IN)  :: BETA(NP), XPLUSD(LDN,M)
    INTEGER,        INTENT(OUT) :: ISTOP
    REAL (KIND=R8), INTENT(OUT) :: F(LDN,NQ), FJACB(LDN,LDNP,NQ), &
        FJACD(LDN,LDM,NQ)

    ! Local variables
    INTEGER, ALLOCATABLE :: DEFINED(:)
    INTEGER :: EXP_NUM, I

    ALLOCATE( DEFINED(SUM(NYDATA_I)) )

    ISTOP = 0

    IF ( ALLOCATED(L_DEF_I) .AND. ALLOCATED(L_I) ) THEN
        DO I = 1, NP
            IF ( L_DEF_I(I) .GT. 0 ) THEN
                IF ( BETA(I) .LT. L_I(I) ) THEN
                    ISTOP = 1
                    WRITE(*,*) "WARNING: THE USER DEFINED LOWER BOUND OF",&
                        L_I(I), "FOR PARAMETER", I, "WAS REACHED."
                    RETURN
                END IF
            END IF
        END DO
    END IF
    IF ( ALLOCATED(U_DEF_I) .AND. ALLOCATED(U_I) ) THEN
        DO I = 1, NP
            IF ( U_DEF_I(I) .GT. 0 ) THEN
                IF ( BETA(I) .GT. U_I(I) ) THEN
                    ISTOP = 1
                    WRITE(*,*) "WARNING: THE USER DEFINED UPPER BOUND OF",&
                        U_I(I), "FOR PARAMETER", I, "WAS REACHED."
                    RETURN
                END IF
```

```
        END IF
    END DO
END IF

IF ( MOD( IDEVAL, 10 ) .GE. 1 ) THEN
    DEFINED(:) = 0
    DO EXP_NUM = 1, NEXP_I
        CALL MODEL_EXPERIMENT(                          &
            NP, BETA, EXP_NAMES_I( EXP_NUM ),           &
            NXDATA_I( EXP_NUM ), NYDATA_I( EXP_NUM ), &
            XPLUSD(                                     &
                SUM( NXDATA_I(1:EXP_NUM-1) )+1          &
                : SUM( NXDATA_I(1:EXP_NUM) ),1          &
            ),                                          &
            F(                                          &
                SUM( NYDATA_I(1:EXP_NUM-1) )+1          &
                : SUM( NYDATA_I(1:EXP_NUM) ),1          &
            ),                                          &
            DEFINED(                                    &
                SUM( NYDATA_I(1:EXP_NUM-1) )+1          &
                : SUM( NYDATA_I(1:EXP_NUM) )            &
            )                                           &
        )
    END DO
    IF ( ANY( DEFINED .EQ. 0 .AND. ACCEPT_UNDEF_I .EQ. 0 ) ) THEN
        ISTOP = 1
        WRITE(*,*) "WARNING: THE MODEL IS NOT DEFINED FOR THE ",&
            "GIVEN PARAMETERS: ", BETA(1:NP)
    END IF
END IF

IF ( MOD( IDEVAL/10, 10 ) .GE. 1 ) THEN
    RETURN
END IF

IF ( MOD( IDEVAL/100, 10 ) .GE. 1 ) THEN
    RETURN
END IF
```

```
END SUBROUTINE ODRPACK_FCN


END MODULE OBJ_FUNC_MOD
```

odrpack_mod.f90

```
MODULE ODRPACK_MOD

USE REAL_PRECISION

PRIVATE

LOGICAL, PARAMETER :: DEBUG = .FALSE.




!-----------------------------------------------------------------------------
!
! <-> SA BETA           On input, the initial guess of the parameters.  On
!                       output, the final estimate of the parameters.
!                       Has length NP.
!  -> OA IFIXB          defaults = 1
!                       Values of 0 for IFIXB(I) indicate that parameter BETA(I)
!                       is not to be changed.
!                       Length of NP
!  -> OA IFIXX          defaults = 1
!                       Value of 0 for IFIXX(I) indicates that D(I) will be
!                       fixed; use when there is no error in X(I) or there is no
!                       X(I).
!                       Size and shape IFIXX(LDIFX,M)
!  -> O  IPRINT         default = -1
!                       Specifies how much report generation is done and where
!                       the reports are written to.
!                       -1 Tells ODRPACK to print its default reports.
! <-     INFO           The return status of ODRPACK
! <-   A IWORK          Various computer values are returned in this array.
!                       Length LIWORK
!  -> O  JOB            default = -1
!                       -1 specifies to use the ODRPACK default for JOB which is
!                       - use explicit ODR
!                       - use forward finite differences
!                       - the covariance matrix will be computed using Jacobian
!                         matrices recalculated at the solution
!                       - D is initialized to 0
!                       - the fit is not a restart
!  -> O  LD2WD          default = 1
!                       Second dimension of WD.
!  -> O  LD2WE          default = 1
!                       Second dimension of WE.
!  -> O  LDIFX          default = N
!                       Leading dimension of IFIXX
!  -> O  LDSCLD         default = N
!                       The leading dimension of SCLD.
!  -> O  LDSTPD         default = N
!                       The leading dimension of STPD
!  -> O  LDWD           default = N
!                       The leading dimension of WD
!  -> O  LDWE           default = N
!                       The leading dimension of WE.
!  -> O  LDX            default = N
!                       The number of observations.  Can exceed N.
!  -> O  LDY            default = N
!                       The number of observations.  Could exceed N if there are
!                       some implicit observations.
!  -     LIWORK         default = 20+p+q(p+m)
!                       The length of IWORK.
!  -> O  LUNERR         default = -1
!                       The LUN to print error messages to.
!                       -1 Tells ODRPACK to print to LUN 6.
!  -> O  LUNRPT         default = -1
!                       The LUN to print the report to.
!                       -1 Tells ODRPACK to print the report to LUN 6 only.
!  -     LWORK          defaults = 18+11p+p^2+m+m^2+4nq+6nm+2nqp+2nqm+q^2+5q+
!                                  q(p+m)+(LDWE*LD2WE)q
!                       The size of WORK.  The default is the largest WORK can
!                       ever be.
!  -> O  M              default = 1
!                       The number of elements in each explanatory variable.
!  -> O  MAXIT          default = -1
```

```
!                       The maximum number of iterations ODRPACK will make.
!                       -1 Tells ODRPACK to use 50 as the default for a new run
!                       and 10 for a restart.
!  -> S  N              The number of explicit observations.
!  -> O  NDIGIT         default = -1
!                       Specifies the number of reliable digits from FCN.
!                       -1 Tells ODRPACK to experimentally determine the number
!                       of reliable digits from FCN.  Experimental determination
!                       requires 4 function evaluations.
!  -> S  NP             The number of parameters.
!  -> O  NQ             default = 1
!                       The number of responses per observation.
!  -> O  PARTOL         default = -1
!                       The parameter tolerence used to determine parameter
!                       convergence.
!                       -1 Tells ODRPACK to use e^1/3 or e^2/3 depending on
!                       whether the problem is implicit or explicit,
!                       respectively.
!  -> OA SCLB           default = -1
!                       The values to scale BETA by to make the algorithm more
!                       stable.
!                       -1 tells ODRPACK to compute the values on its own.
!                       Length NP
!  -> OA SCLD           default = -1
!                       The values to scale D by to make the algorithm more
!                       stable.
!                       -1 tells ODRPACK to comput the values on its own.
!                       Size and shape SCLD(LDSCLD,M)
!  -> O  SSTOL          default = -1
!                       The sum of squares tolerence used to determine sum of
!                       squares convergence.
!                       -1 Tells ODRPACK to use e^0.5 (e = machine precision).
!  -> OA STPB           defaults = -1
!                       The relative step size to use in computing the
!                       derivatives.  The default is automatically determine the
!                       step sizes.
!                       Lenght of NP
!  -> OA STPD           defaults = -1
!                       The relative step size to use in computing the
!                       derivatives with respect to D (delta).
!                       The default is automatically determined.
!                       Size and shape STPD(LDSTPD,M)
!  -> O  TAUFAC         default = -1
!                       The fraction of the full Gauss-Newton step size to take
!                       on the first iteration.  Following iterations adapt
!                       according to the trust region levenburg-marquardt
!                       algorithm.
!                       -1 Tells ODRPACK to use its default (1).
!  -> OA WD             defaults = 1
!                       Weights for error in X (D).
!                       Size and shape WD(LDWD,LD2WD,M)
!  -> OA WE             defaults = 1
!                       Weights for error in Y (E).
!                       Size and shape WE(LDWE,LD2WE,NQ)
! <-> OA WORK           defaults = 0
!                       The work array for ODRPACK.  Optional inputs can be here
!                       if JOB has certain values.
!                       Length of LWORK
!  -> SA X              Observed values of the explanatory variable.
!                       Size and shape X(LDX,M)
!  -> SA Y              Observed values of the response variable.
!                       Size and shape Y(LDY,NQ)
!
REAL (KIND=R8) ::                 &
                  BETA_I(:), PARTOL_I=-1, SCLB_I(:), SCLD_I(:,:), SSTOL_I=-1,  &
                  STPB_I(:), STPD_I(:,:), TAUFAC_I=-1, WD_I(:,:,:),            &
                  WE_I(:,:,:), WORK_I(:), X_I(:,:), Y_I(:,:)
INTEGER        ::                 &
                  IFIXB_I(:), IFIXX_I(:,:), IPRINT_I=-1, INFO_I, IWORK_I(:),   &
                  JOB_I=-1, LD2WD_I=1, LD2WE_I=1, LDIFX_I=-1, LDSCLD_I=-1,     &
                  LDSTPD_I=-1, LDWD_I=-1, LDWE_I=-1, LDX_I=-1, LDY_I=-1,       &
                  LIWORK_I=-1, LUNERR_I=-1, LUNRPT_I=-1, LWORK_I=-1, M_I=1,    &
                  MAXIT_I=-1, N_I=-1, NDIGIT_I=-1, NP_I=-1, NQ_I=1
ALLOCATABLE BETA_I, SCLB_I, SCLD_I, STPB_I, STPD_I, WD_I, WE_I, WORK_I, X_I,   &
            Y_I, IFIXB_I, IFIXX_I, IWORK_I
```

```
!-----------------------------------------------------------------------------
!
!  Internal variables.
!
!  ARRAYS_INITIALIZED   Contains 0 if all the arrays have been initialized.
!                       Each digit represents the initialization state of one of
!                       the mandatory arrays.  They contain a value of 1 when
!                       the array is uninitialized and 0 when it is initialized.
!                         1 - BETA
!                        10 - X
!                       100 - Y
!
INTEGER, PARAMETER :: &
    B111 = 7, &
    B110 = 6, &
    B101 = 5, &
    B100 = 4, &
    B011 = 3, &
    B010 = 2, &
    B001 = 1, &
    B000 = 0
INTEGER :: ARRAYS_INITIALIZED = B111;




PUBLIC :: ODRPACK_SET, ODRPACK_GET, ODRPACK_RUN, ODRPACK_OBJFCN, ODRPACK_OBJFCN_PARTS




CONTAINS
```

```
! If you change the size of the problem (e.g. NP, N, M, etc.) then the
! allocated variables are reallocated and their contents lost.
SUBROUTINE ODRPACK_SET( &
    BETA, PARTOL, SCLB, SCLD, SSTOL,      &
    STPB, STPD, TAUFAC, WD, WE,           &
    WORK, X, Y,                           &
    IFIXB, IFIXX, IPRINT, INFO, IWORK,    &
    JOB, LD2WD, LD2WE, LDIFX, LDSCLD,     &
    LDSTPD, LDWD, LDWE, LDX, LDY,         &
    LUNERR, LUNRPT, M,                    &
    MAXIT, N, NDIGIT, NP, NQ              &
)
    REAL (KIND=R8), OPTIONAL, INTENT(IN) ::               &
        BETA(:), PARTOL, SCLB(:), SCLD(:,:), SSTOL,       &
        STPB(:), STPD(:,:), TAUFAC, WD(:,:,:), WE(:,:,:), &
        WORK(:), X(:,:), Y(:,:)
    INTEGER, OPTIONAL, INTENT(IN)        ::               &
        IFIXB(:), IFIXX(:,:), IPRINT, INFO, IWORK(:),     &
        JOB, LD2WD, LD2WE, LDIFX, LDSCLD,                 &
        LDSTPD, LDWD, LDWE, LDX, LDY,                     &
        LUNERR, LUNRPT, M,                                &
        MAXIT, N, NDIGIT, NP, NQ

    ! set scalars

    IF ( PRESENT(PARTOL) ) THEN; PARTOL_I = PARTOL ; END IF
    IF ( PRESENT(SSTOL)  ) THEN; SSTOL_I  = SSTOL  ; END IF
    IF ( PRESENT(TAUFAC) ) THEN; TAUFAC_I = TAUFAC ; END IF
    IF ( PRESENT(IPRINT) ) THEN; IPRINT_I = IPRINT ; END IF
    IF ( PRESENT(INFO)   ) THEN; INFO_I   = INFO   ; END IF
    IF ( PRESENT(JOB)    ) THEN; JOB_I    = JOB    ; END IF
    IF ( PRESENT(LD2WD)  ) THEN; LD2WD_I  = LD2WD  ; END IF
    IF ( PRESENT(LD2WE)  ) THEN; LD2WE_I  = LD2WE  ; END IF
    IF ( PRESENT(LDIFX)  ) THEN; LDIFX_I  = LDIFX  ; END IF
    IF ( PRESENT(LDSCLD) ) THEN; LDSCLD_I = LDSCLD ; END IF
    IF ( PRESENT(LDSTPD) ) THEN; LDSTPD_I = LDSTPD ; END IF
    IF ( PRESENT(LDWD)   ) THEN; LDWD_I   = LDWD   ; END IF
```

```
IF ( PRESENT(LDWE)   ) THEN; LDWE_I   = LDWE   ; END IF
IF ( PRESENT(LDX)    ) THEN; LDX_I    = LDX    ; END IF
IF ( PRESENT(LDY)    ) THEN; LDY_I    = LDY    ; END IF
IF ( PRESENT(LUNERR) ) THEN; LUNERR_I = LUNERR ; END IF
IF ( PRESENT(LUNRPT) ) THEN; LUNRPT_I = LUNRPT ; END IF
IF ( PRESENT(M)      ) THEN; M_I      = M      ; END IF
IF ( PRESENT(MAXIT)  ) THEN; MAXIT_I  = MAXIT  ; END IF
IF ( PRESENT(N)      ) THEN; N_I      = N      ; END IF
IF ( PRESENT(NDIGIT) ) THEN; NDIGIT_I = NDIGIT ; END IF
IF ( PRESENT(NP)     ) THEN; NP_I     = NP     ; END IF
IF ( PRESENT(NQ)     ) THEN; NQ_I     = NQ     ; END IF

! allocate arrays

IF ( PRESENT(NP) ) THEN
    IF ( ALLOCATED(BETA_I)  ) THEN; DEALLOCATE(BETA_I) ; END IF
    IF ( ALLOCATED(SCLB_I)  ) THEN; DEALLOCATE(SCLB_I) ; END IF
    IF ( ALLOCATED(IFIXB_I) ) THEN; DEALLOCATE(IFIXB_I); END IF
    IF ( ALLOCATED(STPB_I)  ) THEN; DEALLOCATE(STPB_I) ; END IF
    ALLOCATE( BETA_I(NP_I), SCLB_I(NP_I), IFIXB_I(NP_I), STPB_I(NP_I) )
    BETA_I (:) = 0.0_R8
    SCLB_I (:) = -1.0_R8
    IFIXB_I(:) = 1
    STPB_I (:) = -1.0_R8
    ARRAYS_INITIALIZED = IOR( ARRAYS_INITIALIZED, B001 )
END IF

IF ( PRESENT(LDSCLD) .OR. PRESENT(N) .OR. PRESENT(M) ) THEN
    IF ( ALLOCATED(SCLD_I) ) THEN; DEALLOCATE(SCLD_I); END IF
    ALLOCATE( SCLD_I( MAX( N_I, LDSCLD_I ), M_I ) )
    SCLD_I(:,:) = -1.0_R8
END IF

IF ( PRESENT(LDSTPD) .OR. PRESENT(N) .OR. PRESENT(M) ) THEN
    IF ( ALLOCATED(STPD_I) ) THEN; DEALLOCATE(STPD_I); END IF
    ALLOCATE( STPD_I( MAX( N_I, LDSTPD_I ), M_I ) )
    STPD_I(:,:) = -1.0_R8
END IF

IF ( &
    PRESENT(LDWD) .OR. PRESENT(LD2WD) .OR. PRESENT(M) .OR. PRESENT(N) &
) THEN
    IF ( ALLOCATED(WD_I) ) THEN; DEALLOCATE(WD_I); END IF
    ALLOCATE( WD_I( MAX( LDWD_I, N_I ), MAX( LD2WD_I, 1 ), M_I ) )
    WD_I(:,:,:) = 1.0_R8
END IF

IF ( &
    PRESENT(LDWE) .OR. PRESENT(LD2WE) .OR. PRESENT(NQ) .OR. PRESENT(N) &
) THEN
    IF ( ALLOCATED(WE_I) ) THEN; DEALLOCATE(WE_I); END IF
    ALLOCATE( WE_I( MAX( LDWE_I, N_I ), MAX( LD2WE_I, 1 ), NQ_I ) )
    WE_I(:,:,:) = 1.0_R8
END IF

IF ( PRESENT(NP) .OR. PRESENT(M) .OR. PRESENT(N) .OR. PRESENT(NQ) .OR. &
     PRESENT(LDWE) .OR. PRESENT(LD2WE) &
) THEN
    LWORK_I = 18 + 11*NP_I + NP_I**2 + M_I + M_I**2 + 4*N_I*NQ_I &
              + 6*N_I*M_I + 2*N_I*NQ_I*NP_I + 2*N_I*NQ_I*M_I     &
              + NQ_I**2 + 5*NQ_I + NQ_I*( NP_I + M_I )           &
              + MAX(LDWE_I,N_I)*MAX(LD2WE_I,1)*NQ_I
    IF ( ALLOCATED(WORK_I) ) THEN; DEALLOCATE(WORK_I); END IF
    ALLOCATE( WORK_I(LWORK_I) )
    WORK_I(:) = 0
END IF

IF ( PRESENT(LDX) .OR. PRESENT(M) .OR. PRESENT(N) ) THEN
    IF ( ALLOCATED(X_I) ) THEN; DEALLOCATE(X_I); END IF
    ALLOCATE( X_I( MAX( LDX_I, N_I ), M_I ) )
    ARRAYS_INITIALIZED = IOR( ARRAYS_INITIALIZED, B010 )
END IF

IF ( PRESENT(LDY) .OR. PRESENT(NQ) .OR. PRESENT(N) ) THEN
    IF ( ALLOCATED(Y_I) ) THEN; DEALLOCATE(Y_I); END IF
    ALLOCATE( Y_I( MAX( LDY_I, N_I ), NQ_I ) )
    ARRAYS_INITIALIZED = IOR( ARRAYS_INITIALIZED, B100 )
END IF
```

```
IF ( PRESENT(LDIFX) .OR. PRESENT(M) .OR. PRESENT(N) ) THEN
    IF ( ALLOCATED(IFIXX_I) ) THEN; DEALLOCATE(IFIXX_I); END IF
    ALLOCATE( IFIXX_I( MAX( LDIFX_I, N_I ), M_I ) )
    IFIXX_I(:,:) = 1
END IF

IF ( PRESENT(NP) .OR. PRESENT(NQ) .OR. PRESENT(M) ) THEN
    LIWORK_I = 20 + NP_I + NQ_I*( NP_I + M_I )
    IF ( ALLOCATED(IWORK_I) ) THEN; DEALLOCATE(IWORK_I); END IF
    ALLOCATE( IWORK_I( LIWORK_I ) )
END IF

! set arrays

IF ( PRESENT(BETA) .AND. ALLOCATED(BETA_I) ) THEN
    BETA_I(1:NP_I) = BETA(1:NP_I)
    ARRAYS_INITIALIZED = IAND( ARRAYS_INITIALIZED, B110 )
END IF
IF ( PRESENT(SCLB) .AND. ALLOCATED(SCLB_I) ) THEN
    SCLB_I(1:NP_I) = SCLB(1:NP_I)
END IF
IF ( PRESENT(SCLD) .AND. ALLOCATED(SCLD_I) ) THEN
    SCLD_I(1:MAX(LDSCLD_I,N_I),1:M_I) = SCLD(1:MAX(LDSCLD_I,N_I),1:M_I)
END IF
IF ( PRESENT(STPB) .AND. ALLOCATED(STPB_I) ) THEN
    STPB_I(1:NP_I) = STPB(1:NP_I)
END IF
IF ( PRESENT(STPD) .AND. ALLOCATED(STPD_I) ) THEN
    STPD_I(1:MAX(LDSTPD_I,N_I),1:M_I) = STPD(1:MAX(LDSTPD_I,N_I),1:M_I)
END IF
IF ( PRESENT(WD) .AND. ALLOCATED(WD_I) ) THEN
    WD_I(1:MAX(LDWD_I,N_I),1:MAX(LD2WD_I,1),1:M_I) = &
        WD(1:MAX(LDWD_I,N_I),1:MAX(LD2WD_I,1),1:M_I)
END IF
IF ( PRESENT(WE) .AND. ALLOCATED(WE_I) ) THEN
    WE_I(1:MAX(LDWE_I,N_I),1:MAX(LD2WE_I,1),1:NQ_I) = &
        WE(1:MAX(LDWE_I,N_I),1:MAX(LD2WE_I,1),1:NQ_I)
END IF
    IF ( PRESENT(WORK) .AND. ALLOCATED(WORK_I) ) THEN
        WORK_I(1:LWORK_I) = WORK(1:LWORK_I)
    END IF
    IF ( PRESENT(X) .AND. ALLOCATED(X_I) ) THEN
        X_I(1:MAX(LDX_I,N),1:M_I) = &
            X(1:MAX(LDX_I,N),1:M_I)
        ARRAYS_INITIALIZED = IAND( ARRAYS_INITIALIZED, B101 )
    END IF
    IF ( PRESENT(Y) .AND. ALLOCATED(Y_I) ) THEN
        Y_I(1:MAX(LDY_I,N_I),1:NQ_I) = &
            Y(1:MAX(LDY_I,N_I),1:NQ_I)
        ARRAYS_INITIALIZED = IAND( ARRAYS_INITIALIZED, B011 )
    END IF
    IF ( PRESENT(IFIXB) .AND. ALLOCATED(IFIXB_I) ) THEN
        IFIXB_I(1:NP) = IFIXB(1:NP)
    END IF
    IF ( PRESENT(IFIXX) .AND. ALLOCATED(IFIXX_I) ) THEN
        IFIXX_I(1:MAX(LDIFX_I,N_I),1:M_I) = &
            IFIXX(1:MAX(LDIFX_I,N_I),1:M_I)
    END IF
    IF ( PRESENT(IWORK) .AND. ALLOCATED(IWORK_I) ) THEN
        IWORK_I(1:LIWORK_I) = IWORK(1:LIWORK_I)
    END IF

END SUBROUTINE ODRPACK_SET




SUBROUTINE ODRPACK_GET( WORK, BETA )
    REAL (KIND=R8), INTENT(OUT), OPTIONAL :: WORK(:), BETA(:)
    IF ( PRESENT(WORK) ) THEN; WORK(1:LWORK_I) = WORK_I(1:LWORK_I); END IF
    IF ( PRESENT(BETA) ) THEN; BETA(1:NP_I) = BETA_I(1:NP_I); END IF
END SUBROUTINE ODRPACK_GET
```

```
SUBROUTINE ODRPACK_RUN()

    USE OBJ_FUNC_MOD

    ! Ensure the user has set all the manditory variables.

    IF ( ARRAYS_INITIALIZED .NE. 0 ) THEN
        IF ( IAND( ARRAYS_INITIALIZED, B001 ) .GT. 0 ) THEN
            WRITE( 6, * ) "ERROR: ODRPACK_MOD: BETA NOT INITIALIZED"
        ELSEIF ( IAND( ARRAYS_INITIALIZED, B010 ) .GT. 0 ) THEN
            WRITE( 6, * ) "ERROR: ODRPACK_MOD: X NOT INITIALIZED"
        ELSEIF ( IAND( ARRAYS_INITIALIZED, B100 ) .GT. 0 ) THEN
            WRITE( 6, * ) "ERROR: ODRPACK_MOD: Y NOT INITIALIZED"
        ELSE
            WRITE( 6, * ) "ERROR: ODRPACK_MOD: UNKNOWN ERROR RELATED TO ARRAY",&
                          " INITIALIZATION"
        END IF
        RETURN
    END IF

    IF ( N_I .LE. 0 ) THEN
        WRITE( 6, * ) "ERROR: ODRPACK_MOD: N <= 0"
        RETURN
    END IF

    IF ( NP_I .LE. 0 ) THEN
        WRITE( 6, * ) "ERROR: ODRPACK_MOD: NP <= 0"
        RETURN
    END IF

    ! Run ODRPACK

    IF ( DEBUG ) THEN
        WRITE(*,*) "N_I = ", N_I
        WRITE(*,*) "M_I = ", M_I
        WRITE(*,*) "NP_I = ", NP_I
        WRITE(*,*) "NQ_I = ", NQ_I
        WRITE(*,*) "BETA_I = ", BETA_I
        WRITE(*,*) "Y_I = ", Y_I
        WRITE(*,*) "MAX(LDY_I,N_I) = ", MAX(LDY_I,N_I)
        WRITE(*,*) "X_I = ", X_I
        WRITE(*,*) "MAX(LDX_I,N_I) = ", MAX(LDX_I,N_I)
        WRITE(*,*) "WE_I = ", WE_I
        WRITE(*,*) "MAX(LDWE_I,N_I) = ", MAX(LDWE_I,N_I)
        WRITE(*,*) "MAX(LD2WE_I,1) = ", MAX(LD2WE_I,1)
        WRITE(*,*) "WD_I = ", WD_I
        WRITE(*,*) "MAX(LDWD_I,N_I) = ", MAX(LDWD_I,N_I)
        WRITE(*,*) "MAX(LD2WD_I,1) = ", MAX(LD2WD_I,1)
        WRITE(*,*) "IFIXB_I = ", IFIXB_I
        WRITE(*,*) "IFIXX_I = ", IFIXX_I
        WRITE(*,*) "MAX(LDIFX_I,N_I) = ", MAX(LDIFX_I,N_I)
        WRITE(*,*) "JOB_I = ", JOB_I
        WRITE(*,*) "NDIGIT_I = ", NDIGIT_I
        WRITE(*,*) "TAUFAC_I = ", TAUFAC_I
        WRITE(*,*) "SSTOL_I = ", SSTOL_I
        WRITE(*,*) "PARTOL_I = ", PARTOL_I
        WRITE(*,*) "MAXIT_I = ", MAXIT_I
        WRITE(*,*) "IPRINT_I = ", IPRINT_I
        WRITE(*,*) "LUNERR_I = ", LUNERR_I
        WRITE(*,*) "LUNRPT_I = ", LUNRPT_I
        WRITE(*,*) "STPB_I = ", STPB_I(1)
        WRITE(*,*) "STPD_I = ", STPD_I(1,1)
        WRITE(*,*) "LDSTPD_I = ", LDSTPD_I
        WRITE(*,*) "SCLB_I = ", SCLB_I(1)
        WRITE(*,*) "SCLD_I = ", SCLD_I(1,1)
        WRITE(*,*) "LDSCLD_I = ", LDSCLD_I
        WRITE(*,*) "WORK_I = ", WORK_I(1:100)
        WRITE(*,*) "LWORK_I = ", LWORK_I
        WRITE(*,*) "IWORK_I = ", IWORK_I(1:47)
        WRITE(*,*) "LIWORK_I = ", LIWORK_I
        WRITE(*,*) "INFO_I = ", INFO_I
    END IF

    CALL DODRC( &
        ODRPACK_FCN,                                &
        N_I, M_I, NP_I, NQ_I,                       &
```

```
        BETA_I,                                     &
        Y_I, MAX(LDY_I,N_I), X_I, MAX(LDX_I,N_I),   &
        WE_I, MAX(LDWE_I,N_I), MAX(LD2WE_I,1),      &
        WD_I, MAX(LDWD_I,N_I), MAX(LD2WD_I,1),      &
        IFIXB_I, IFIXX_I, MAX(LDIFX_I,N_I),         &
        JOB_I, NDIGIT_I, TAUFAC_I,                  &
        SSTOL_I, PARTOL_I, MAXIT_I,                 &
        IPRINT_I, LUNERR_I, LUNRPT_I,               &
        STPB_I, STPD_I, MAX(LDSTPD_I,N_I),          &
        SCLB_I, SCLD_I, MAX(LDSCLD_I,N_I),          &
        WORK_I, LWORK_I, IWORK_I, LIWORK_I,         &
        INFO_I                                      &
    )

END SUBROUTINE ODRPACK_RUN
```

```
!-----------------------------------------------------------------------------
!
!  ODRPACK_OBJFCN
!
!  Calculate the ODRPACK WSOS at BETA with X, Y, WD, and WD from this module.
!  Use XPLUSD as input to FCN and the output of FCN for F.
!
REAL (KIND=R8) FUNCTION ODRPACK_OBJFCN( BETA, XPLUSD )
    USE OBJ_FUNC_MOD
    ! arguments
    REAL (KIND=R8) :: BETA(:), XPLUSD(:,:)
    ! local vars
    REAL (KIND=R8) :: PARTS(N_I)
    CALL ODRPACK_OBJFCN_PARTS( BETA, XPLUSD, PARTS )
    ODRPACK_OBJFCN = SUM( PARTS )
    RETURN
END FUNCTION ODRPACK_OBJFCN
```

```
!-----------------------------------------------------------------------------
!
!  ODRPACK_OBJFCN_PARTS
!
!  Calculate the ODRPACK WSOS at BETA with X, Y, WD, and WD from this module.
!  Use XPLUSD as input to FCN and the output of FCN for F.  Output the parts of
!  the objective function ready to be added to give the whole function.
!
SUBROUTINE ODRPACK_OBJFCN_PARTS( BETA, XPLUSD, PARTS )
    USE OBJ_FUNC_MOD
    ! arguments
    REAL (KIND=R8) :: BETA(:), XPLUSD(:,:)
    REAL (KIND=R8), INTENT(OUT) :: PARTS(:)
    ! local variables
    REAL (KIND=R8) :: F(N_I,NQ_I), FJACB(N_I,NP_I,NQ_I), FJACD(N_I,M_I,NQ_I)
    INTEGER :: IDEVAL = 1, ISTOP
    IF ( SIZE(BETA) .LT. NP_I ) THEN
        WRITE(*,*) "ERROR: ODRPACK_OBJFCN_PARTS: SIZE OF BETA LESS THAN NP"
        PARTS(:) = -1.0_R8
        RETURN
    END IF
    IF ( SIZE(XPLUSD,1) .LT. N_I ) THEN
        WRITE(*,*) "ERROR: ODRPACK_OBJFCN_PARTS: SIZE OF XPLUSD LESS THAN N IN ",&
            "FIRST DIMENSION"
        PARTS(:) = 1.0_R8
        RETURN
    END IF
    IF ( SIZE(XPLUSD,2) .NE. 1 ) THEN
        WRITE(*,*) "WARNING: ODRPACK_OBJFCN_PARTS: SIZE OF XPLUSD IS NOT 1 IN THE",&
            " SECOND DIMENSION.  SIZES OTHER THAN 1 ARE NOT YET SUPPORTED."
    END IF
    IF ( SIZE(BETA) .GT. NP_I ) THEN
        WRITE(*,*) "WARNING: ODRPACK_OBJFCN_PARTS: SIZE OF BETA GREATER THAN NP"
    END IF
    CALL ODRPACK_FCN( N_I, M_I, NP_I, NQ_I, N_I, M_I, NP_I, BETA, &
```

```
        XPLUSD, IFIXB_I, &
        IFIXX_I, LDIFX_I, IDEVAL, F, FJACB, FJACD, ISTOP )
    PARTS = WD_I(1:N_I,1,1)*(X_I(1:N_I,1)-XPLUSD(1:N_I,1))**2 + &
        WE_I(1:N_I,1,1)*(Y_I(1:N_I,1)-F(1:N_I,1))**2
END SUBROUTINE ODRPACK_OBJFCN_PARTS




END MODULE ODRPACK_MOD
```

prod.f90

```
MODULE PROD_MOD

!==============================================================================
!
!  PROD
!
!  Tests: Global optimizer.  Results should be similar to run 0135
!
!  Problem: 3 equation frog egg model
!
!  Output: ODRPACK report file, and stdout
!
!  Description: The results of this test case yield similar parameters to run
!  0135.  The results are not identical.  Care was not taken to ensure the input
!  was identical.
!
!==============================================================================

    USE REAL_PRECISION
    USE STRINGS

    INTEGER :: MODEL_EXP_COUNT = 0

    INTEGER, PARAMETER :: NUM_VARS = 5
    INTEGER, PARAMETER :: NUM_PARAMS = 13

    ! PARAMS
    REAL (KIND=R8)   :: VCP, VCPP, VCPPP, &
                        VWP, VWPP, VWPPP, &
                        KMC, KMCR,        &
                        KMW, KMWR,        &
                        VC,  VW,          &
                        DILUTION
    REAL (KIND=R8)   :: MODEL_PARAMS( NUM_PARAMS )
    EQUIVALENCE( VCP,      MODEL_PARAMS( 1) )
    EQUIVALENCE( VCPP,     MODEL_PARAMS( 2) )
```

```
EQUIVALENCE( VCPPP,    MODEL_PARAMS( 3) )
EQUIVALENCE( VWP,      MODEL_PARAMS( 4) )
EQUIVALENCE( VWPP,     MODEL_PARAMS( 5) )
EQUIVALENCE( VWPPP,    MODEL_PARAMS( 6) )
EQUIVALENCE( KMC,      MODEL_PARAMS( 7) )
EQUIVALENCE( KMCR,     MODEL_PARAMS( 8) )
EQUIVALENCE( KMW,      MODEL_PARAMS( 9) )
EQUIVALENCE( KMWR,     MODEL_PARAMS(10) )
EQUIVALENCE( VC,       MODEL_PARAMS(11) )
EQUIVALENCE( VW,       MODEL_PARAMS(12) )
EQUIVALENCE( DILUTION, MODEL_PARAMS(13) )

! VARS
REAL (KIND=R8)   :: M, C, W, L, L2
REAL (KIND=R8)   :: MODEL_VARS(NUM_VARS)
INTEGER, PARAMETER :: &
    M_I  = 1, &
    C_I  = 2, &
    W_I  = 3, &
    L_I  = 4, &
    L2_I = 5
EQUIVALENCE( M , MODEL_VARS(M_I ) )
EQUIVALENCE( C , MODEL_VARS(C_I ) )
EQUIVALENCE( W , MODEL_VARS(W_I ) )
EQUIVALENCE( L , MODEL_VARS(L_I ) )
EQUIVALENCE( L2, MODEL_VARS(L2_I) )

! VAR DERIVATIVES
REAL (KIND=R8)   :: MDOT, CDOT, WDOT, LDOT, L2DOT
REAL (KIND=R8)   :: MODEL_DERS(NUM_VARS)
EQUIVALENCE( MDOT , MODEL_DERS(1) )
EQUIVALENCE( CDOT , MODEL_DERS(2) )
EQUIVALENCE( WDOT , MODEL_DERS(3) )
EQUIVALENCE( LDOT , MODEL_DERS(4) )
EQUIVALENCE( L2DOT, MODEL_DERS(5) )

! INDEPENDENT VARS
REAL (KIND=R8)   :: TOTALCYCLIN
```

```
! Experiement variables
INTEGER, PARAMETER :: NUM_EXP = 10
CHARACTER(MAXSTR), PARAMETER  :: EXP_NAMES(NUM_EXP) = (/ &
    "MOORE_TIMELAG                                    ",&
    "MOORE_THRESHOLD                                  ",&
    "KUMAGAI_AND_DUMPHY_MPF_ACT_MPHASE                ",&
    "KUMAGAI_AND_DUMPHY_MPF_ACT_INTERPHASE            ",&
    "KUMAGAI_AND_DUMPHY_MPF_INACT_MPHASE              ",&
    "KUMAGAI_AND_DUMPHY_MPF_INACT_INTERPHASE          ",&
    "KUMAGAI_AND_DUMPHY_CDC25_ACT                     ",&
    "KUMAGAI_AND_DUMPHY_CDC25_INACT                   ",&
    "TANG_WEE1_INACT                                  ",&
    "TANG_WEE1_ACT                                    " &
/)

INTEGER, PARAMETER :: NUM_XDATA = 31, NUM_YDATA = 31
INTEGER, PARAMETER :: EXP_NXDATA(NUM_EXP) = (/ &
    4, 2, 3, 3, 2, 3, 4, 4, 4, 2              &
/)
INTEGER, PARAMETER :: EXP_NYDATA(NUM_EXP) = (/ &
    4, 2, 3, 3, 2, 3, 4, 4, 4, 2              &
/)
REAL (KIND=R8), PARAMETER :: EXP_XDATA(NUM_XDATA) = (/ &
    0.20_R8, 0.25_R8, 0.30_R8, 0.50_R8,               &
    0.0_R8, 0.0_R8,                                   &
    2.0_R8, 4.0_R8, 8.0_R8,                           &
    2.0_R8, 4.0_R8, 8.0_R8,                           &
    4.0_R8, 16.0_R8,                                  &
    2.0_R8, 4.0_R8, 16.0_R8,                          &
    1.25_R8, 2.5_R8, 5.0_R8, 10.0_R8,                 &
    5.0_R8, 10.0_R8, 20.0_R8, 40.0_R8,                &
    2.0_R8, 5.0_R8, 7.0_R8, 10.0_R8,                  &
    7.5_R8, 15.0_R8                                   &
/)
REAL (KIND=R8), PARAMETER :: EXP_YDATA(NUM_YDATA) = (/ &
    45.0_R8, 40.0_R8, 30.0_R8, 20.0_R8,               &
    0.18_R8, 3.0_R8,                                  &
    0.75_R8, 0.51_R8, 0.21_R8,                        &
    1.0_R8, 1.0_R8, 0.85_R8,                          &
```

```
    0.0_R8, 0.0_R8,                                  &
    1.0_R8, 1.0_R8, 1.0_R8,                          &
    0.8_R8, 0.9_R8, 1.0_R8, 1.0_R8,                  &
    0.75_R8, 0.5_R8, 0.1_R8, 0.0_R8,                 &
    0.5_R8, 0.0_R8, 0.0_R8, 0.0_R8,                  &
    0.5_R8, 1.0_R8                                   &
/)


! Special variables
! MPF_THRESHOLD is the the MPF concentration above which MPF is considered
! to have been activated in the time lag calculations.  When looking for the
! timelag for MPF inactivation the it is the concentration below which MPF
! is considered to have been inactivated.
REAL (KIND=R8) :: MPF_THRESHOLD



CONTAINS





!-----------------------------------------------------------------------------
!
!  TIMELAG
!
!  Calculates the time lag for MPF activation or inactivation.  The initial
!  conditions are set outside this routine.  The routine assumes if MPF
!  reaches TOTALCYCLIN/2 then it has changed its activation state.
!
REAL (KIND=R8) FUNCTION TIMELAG( DEFINED )
    USE LSODAR_MOD
    INTEGER, INTENT(OUT) :: DEFINED

    ! local variables
    INTEGER :: ISTATE
    REAL (KIND=R8) :: T, MINF, MINFPD, AT, SAVED_VARS(NUM_VARS)
    REAL (KIND=R8) :: UNDEF_VALUE

    UNDEF_VALUE = -1440.0_R8 * TOTALCYCLIN + 2880.0_R8

    IF ( TOTALCYCLIN .LE. 0 ) THEN
        TIMELAG= UNDEF_VALUE
        DEFINED = 0
        RETURN
    END IF

    SAVED_VARS(:) = MODEL_VARS(:)
    ! Calculate infinite MPF value
    AT = 1440_R8
    CALL LSODAR_RECALL()
    CALL LSODAR_SET(             &
        Y          = MODEL_VARS, &
        TOUT       = AT          &
    )
    CALL LSODAR_RUN()
    CALL LSODAR_GET( Y = MODEL_VARS )
    MINF = M
    CALL LSODAR_SET( TOUT = AT*1.01 )
    CALL LSODAR_RUN()
    CALL LSODAR_GET( Y = MODEL_VARS )
    MINFPD = M
    DO WHILE ( (MINF-MINFPD)/MINF .GT. 1E-8_R8 .AND. AT .LT. 1E5_R8 )
        AT = AT*2
        CALL LSODAR_SET( TOUT = AT )
        CALL LSODAR_RUN()
        CALL LSODAR_GET( Y = MODEL_VARS )
        MINF = M
        CALL LSODAR_SET( TOUT = AT*1.01 )
        CALL LSODAR_RUN()
        CALL LSODAR_GET( Y = MODEL_VARS )
        MINFPD = M
    END DO
    MODEL_VARS(:) = SAVED_VARS(:)
    IF ((MINF .GT. TOTALCYCLIN/2 .AND. M .NE. 0.0_R8) .OR. &
        (MINF .LT. TOTALCYCLIN/2 .AND. M .EQ. 0.0_R8)) THEN
```

```
        TIMELAG = UNDEF_VALUE
        DEFINED = 0
        RETURN
    END IF
    IF ( M .EQ. 0.0_R8 ) THEN
        MPF_THRESHOLD = MINF/2
    ELSE
        MPF_THRESHOLD = (TOTALCYCLIN-MINF)/2+MINF
    END IF

    CALL LSODAR_RECALL()
    CALL LSODAR_SET(             &
        Y          = MODEL_VARS, &
        FIND_ROOTS = 1,          &
        TOUT       = 1440.0_R8   &
    )
    CALL LSODAR_RUN()
    CALL LSODAR_GET(             &
        ISTATE     = ISTATE,     &
        T          = T           &
    )
    IF ( ISTATE .NE. 3 ) THEN
        TIMELAG = UNDEF_VALUE
        DEFINED = 0
        RETURN
    END IF
    TIMELAG = T
    DEFINED = 1
    RETURN

END FUNCTION TIMELAG



!-----------------------------------------------------------------------------
!
!
!
REAL (KIND=R8) FUNCTION THRESHOLD( DEFINED )
    USE LSODAR_MOD
    INTEGER :: DEFINED

    ! local variables
    !  LOWER the lower bound of the threshold being calculated
    !  UPPER the upper bound of the threshold being calculated
    !  TOL   the tolerence for the threshold being calculated, this is the
    !        allowed relative error in the threshold calculation.
    !  NEXT  the bisection point of LOWER and UPPER.  after each iteration
    !        either LOWER or UPPER will be assigned to next and next will
    !        be reassigned (UPPER-LOWER)/2.
    !  ACTIVATION specifies whether THRESHOLD is looking for an activation
    !        threshold or inactivation threshold.  It is .TRUE. when looking
    !        for an activation threshold and .FALSE. when looking for an
    !        inactivation threshold.
    !  ISTATE is the state returned by lsodar.  Used for checking errors
    !        returned by lsodar.
    REAL (KIND=R8) :: SAVED_MODEL_VARS(NUM_VARS)
    REAL (KIND=R8) :: LOWER, UPPER, NEXT, TRASH
    REAL (KIND=R8), PARAMETER :: &
        TOL = 1E-10_R8,          &
        MAX_UPPER = 10.0_R8,     &
        MIN_LOWER = 0.0005_R8
    LOGICAL :: ACTIVATION
    INTEGER :: ISTATE

    ! Check if THRESHOLD should look for an activation or inactivation
    ! threshold.  It depends on the initial value of MPF.  A value of zero
    ! indicates an activation threshold is desired and an inactivation
    ! threshold otherwise.
    ACTIVATION = M .EQ. 0.0
!   WRITE( 6, * ) "ACTIVATION = ", ACTIVATION

    ! Save the initial conditions for restoring between iterations.
    SAVED_MODEL_VARS(:) = MODEL_VARS(:)

    ! Set initial guess for bounds on the threshold.
```

```
        LOWER = 0.001_R8
        UPPER = 1_R8

        ! Check that the lower bound LOWER is really a lower bound.  If it isn't
        ! then half it and set UPPER to the old LOWER.
        MODEL_VARS(:) = SAVED_MODEL_VARS(:)
        TOTALCYCLIN = LOWER
        IF ( .NOT. ACTIVATION ) THEN; M = TOTALCYCLIN; END IF
        TRASH = TIMELAG(DEFINED)
        CALL LSODAR_GET( ISTATE = ISTATE )
        DO WHILE (                                                  &
            ( ACTIVATION .AND. TRASH < 1440_R8 ) .AND.       &
            ( .NOT. ACTIVATION .AND. TRASH >= 1440_R8 ) .AND. &
            LOWER .GT. TOL &
        )
            LOWER = LOWER/2_R8
            TOTALCYCLIN = LOWER
!           WRITE(*,*) "NEW LOWER = ", LOWER
            MODEL_VARS(:) = SAVED_MODEL_VARS(:)
            IF ( .NOT. ACTIVATION ) THEN; M = TOTALCYCLIN; END IF
            TRASH = TIMELAG(DEFINED)
            CALL LSODAR_GET( ISTATE = ISTATE )
        END DO
        IF ( LOWER .LE. TOL ) THEN
            WRITE( 6, * ) "WARNING: WHILE CALCULATING THE THRESHOLD, THE ",&
                "MINIMUM LOWER BOUND OF ", LOWER, " WAS REACHED."
            DEFINED = 0
            THRESHOLD = LOWER
            RETURN
        END IF

        ! Check that the upper bound UPPER is really an upper bound.  If it
        ! isn't then double it and set LOWER to the old UPPER.
        MODEL_VARS(:) = SAVED_MODEL_VARS(:)
        TOTALCYCLIN = UPPER
        IF ( .NOT. ACTIVATION ) THEN; M = TOTALCYCLIN; END IF
        TRASH = TIMELAG(DEFINED)
        CALL LSODAR_GET( ISTATE = ISTATE )
        DO WHILE (                                                  &
            ( ACTIVATION .AND. TRASH >= 1440_R8 ) .AND.       &
            ( .NOT. ACTIVATION .AND. TRASH < 1440_R8 ) .AND. &
            UPPER .LT. 1_R8/TOL &
        )
            UPPER = UPPER*2_R8
            TOTALCYCLIN = UPPER
!           WRITE(*,*) "NEW UPPER = ", UPPER
            MODEL_VARS(:) = SAVED_MODEL_VARS(:)
            IF ( .NOT. ACTIVATION ) THEN; M = TOTALCYCLIN; END IF
            TRASH = TIMELAG(DEFINED)
            CALL LSODAR_GET( ISTATE = ISTATE )
        END DO
        IF ( UPPER .GE. 1_R8/TOL ) THEN
            WRITE( 6, * ) "WARNING: WHILE CALCULATING THE THRESHOLD, THE ",&
                "MAXIMUM UPPER BOUND OF ", UPPER, " WAS REACHED."
            DEFINED = 0
            THRESHOLD = UPPER
            RETURN
        END IF

        ! Bissect the interval [LOWER, UPPER] until TOL is reached.
        DO WHILE ( (UPPER-LOWER)/LOWER > TOL )
            !WRITE(*,*) "NEXT = ", NEXT
            MODEL_VARS(:) = SAVED_MODEL_VARS(:)
            NEXT = (UPPER+LOWER)/2
            TOTALCYCLIN = NEXT
            IF ( .NOT. ACTIVATION ) THEN; M = TOTALCYCLIN; END IF
            TRASH = TIMELAG(DEFINED)
            CALL LSODAR_GET( ISTATE = ISTATE )
            IF (                                                &
                ( ACTIVATION .AND. TRASH >= 1440_R8 ) .OR.  &
                ( .NOT. ACTIVATION .AND. TRASH < 1440_R8 ) &
            ) THEN
                LOWER = NEXT
            ELSE
                UPPER = NEXT
            END IF
        END DO
```

```
    IF ( ISTATE .LE. 0 ) THEN
        WRITE( 6, * ) "WARNING: THRESHOLD DIDN'T CONVERGE BECAUSE LSODAR ",&
            "RETURNED AN EROR"
        DEFINED = 0
        THRESHOLD = LOWER
        RETURN
    END IF
    DEFINED = 1
    THRESHOLD = LOWER
    RETURN

END FUNCTION THRESHOLD




!-----------------------------------------------------------------------------
!
!  TIMECOURSE
!
!  Calculates time course data at the specified points in XDATA and returns
!  the results in YDATA.  The points in XDATA are assumed to be sorted.
!
SUBROUTINE TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, VAR_I )
    USE LSODAR_MOD
    INTEGER, INTENT(IN)  :: NX, NY
    INTEGER, INTENT(IN)  :: VAR_I
    INTEGER, INTENT(OUT) :: DEFINED(NY)
    REAL (KIND=R8), INTENT(IN)  :: XDATA(NX)
    REAL (KIND=R8), INTENT(OUT) :: YDATA(NY)

    ! Local variables
    INTEGER :: I, ISTATE

    DEFINED(:) = 0

    IF ( NX .NE. NY ) THEN
        WRITE( 6, * ) "WARNING: TIMECOURSE CALCULATIONS REQUIRE THE ",&
            "NUMBER OF X AND Y DATA TO BE EQUAL. NX = ", NX, &
            " NY = ", NY
    END IF

    IF ( MIN( NX, NY ) .LE. 0 ) THEN
        WRITE( 6, * ) "ERROR: AT LEAST ONE TIMECOURSE CALCULATION MUST BE",&
            "PERFORMED."
        RETURN
    END IF

    CALL LSODAR_RECALL()
    CALL LSODAR_SET( Y = MODEL_VARS )
    DO I = 1, MIN( NX, NY )
        CALL LSODAR_SET( TOUT = XDATA(I) )
        CALL LSODAR_RUN()
        CALL LSODAR_GET( ISTATE = ISTATE, Y = MODEL_VARS )
        DO WHILE ( ISTATE .EQ. 3 )
            ISTATE = 2
            CALL LSODAR_SET( ISTATE = ISTATE )
            CALL LSODAR_RUN()
            CALL LSODAR_GET( ISTATE = ISTATE, Y = MODEL_VARS )
        END DO
        YDATA(I) = MODEL_VARS(VAR_I)
        IF ( ISTATE .EQ. 2 .OR. ISTATE .EQ. 1 ) THEN
            DEFINED(I) = 1
        ELSE
            WRITE(*,*) "WARNING: ISTATE=", ISTATE, " IS AN ERROR STATE."
        END IF
    END DO

END SUBROUTINE TIMECOURSE




END MODULE PROD_MOD
```

```
PROGRAM PROD

    USE PROD_MOD
    USE LSODAR_MOD
    USE REAL_PRECISION
    USE ODRPACK_MOD
    USE OBJ_FUNC_MOD
    USE VTDIRECT_MOD
    USE VTDIRECT_GLOBAL

    ! Local variables
    INTEGER, PARAMETER :: NUM_POINTS = 200
    REAL (KIND=R8) :: XDATA(NUM_POINTS), YDATA(NUM_POINTS), &
        SAVED_PARAMS(NUM_PARAMS), &
        MPFACT(1), MPFINACT(1), &
        MAXX(NUM_EXP)
    INTEGER :: DEFINED(NUM_POINTS), I, J
    INTEGER :: IFIXB(NUM_PARAMS), IFIXX(NUM_XDATA)
    REAL (KIND=R8) :: WE(NUM_YDATA), WD(NUM_XDATA)
    LOGICAL :: GRAPH = .FALSE., GLOBAL = .TRUE., RESTART = .FALSE., &
               GLOBAL_ONLY = .FALSE., OD_ON = .TRUE.
    REAL (KIND=R8) :: RWORK(10000), PARTS(NUM_XDATA)
    REAL (KIND=R8) :: LOWER_BOUND(NUM_PARAMS), UPPER_BOUND(NUM_PARAMS)
    INTEGER        :: L_DEF(NUM_PARAMS), U_DEF(NUM_PARAMS)
    REAL (KIND=R8) :: X(NUM_PARAMS), FMIN, BETA(NUM_PARAMS)
    INTEGER        :: STATUS
    CHARACTER (100) :: FILENAME
    INTEGER, PARAMETER :: MAX_NUM_BOX = 100
    TYPE(HyperBox) :: BOX_SET(MAX_NUM_BOX)
    INTEGER        :: NUM_BOX, MAX_EVL = 10000
    REAL (KIND=R8) :: MIN_SEP, REL_MIN_SEP=0.5_R8

    NAMELIST / PARAMETER_NML / VCP, VCPP, VCPPP, VWP, VWPP, VWPPP, &
        KMC, KMCR, KMW, KMWR, VC, VW, DILUTION
    NAMELIST / FIXED_NML /  VCP, VCPP, VCPPP, VWP, VWPP, VWPPP, &
        KMC, KMCR, KMW, KMWR, VC, VW, DILUTION, IFIXX
    NAMELIST / MISC_NML / WE, WD, GRAPH, BETA, GLOBAL, RESTART, GLOBAL_ONLY, &
        MAX_EVL, REL_MIN_SEP, OD_ON
    NAMELIST / OUTPUT / RWORK, MODEL_PARAMS


    LOWER_BOUND(1:NUM_PARAMS) = (/   &
        0.0_R8,  0.0_R8,   0.00_R8,  & ! VCP, VCPP, VCPPP,
        0.0_R8,  0.0_R8,   0.00_R8,  & ! VWP, VWPP, VWPPP,
        0.001_R8,  0.001_R8,         & ! KMC, KMCR,
        0.001_R8,  0.001_R8,         & ! KMW, KMWR,
        0.0_R8,                      & ! VC
        0.0_R8,                      & ! VW
        1.0_R8 /)                      ! DILUTION
    UPPER_BOUND(1:NUM_PARAMS) = (/   &
        0.2_R8, 02.0_R8, 0.20_R8,    &
        0.2_R8, 02.0_R8, 0.20_R8,    &
        20.0_R8, 020.0_R8,           &
        02.0_R8, 020.0_R8,           &
        080.0_R8,                    &
        020.0_R8,                    &
        02.0_R8 /)
    L_DEF(1:NUM_PARAMS) = 1
    U_DEF(1:NUM_PARAMS) = 1


    ! Set independent variables.
    TOTALCYCLIN = 0.25_R8

    ! Set initial conditions.
    ! These conditions are for no experiment in particular
    M  = 0.0_R8
    C  = 0.0_R8
    W  = 1.0_R8
    L  = 1.0_R8
    L2 = 0.0_R8
```

```
    ! Read name lists.
    OPEN( UNIT=5, FILE="prod.nml" )
    READ( UNIT=5, NML=PARAMETER_NML )
    READ( UNIT=5, NML=MISC_NML )
    CLOSE( UNIT=5 )
    BETA = MODEL_PARAMS

    MIN_SEP = REL_MIN_SEP * SQRT(DOT_PRODUCT(   &
        UPPER_BOUND(:) - LOWER_BOUND(:),        &
        UPPER_BOUND(:) - LOWER_BOUND(:)         &
    ))



    ! Set LSODAR parameters
    CALL LSODAR_SET(              &
        NEQ         = NUM_VARS,   &
        Y           = MODEL_VARS, &
        TOUT        = 100.0_R8,   &
        NG          = 1,          &
        FIND_ROOTS  = 0,          &
        MAXWORK     = 1000000,    &
        RTOL        = 1E-10_R8,   &
        ATOL        = 1E-10_R8,   &
        ROOT_TOL    = 1E-10_R8,   &
        FILE_UNIT   = 8           &
!        PRINT_EVERY = 0.02_R8     &
    )

    ! Save lsodar parameters for later recalling in the experiment or transform
    ! sections.
    CALL LSODAR_SAVE()

    RWORK(:) = 0.0_R8

    IF ( RESTART ) THEN
        OPEN( UNIT=8, FILE="output.nml" )
        READ( UNIT=8, NML=OUTPUT )
        CLOSE( UNIT=8 )
        BETA(1:NUM_PARAMS) = MODEL_PARAMS(1:NUM_PARAMS)
    END IF

    SAVED_PARAMS(:) = MODEL_PARAMS(:)

    IF ( .NOT. GRAPH ) THEN

        ! Set fixed parameters equal to 0.0_R8 here.  This will not be the
        ! parameter value, it just indicates to fix this parameter.  Values of
        ! 1.0_R8 indicate the parameter should be varied.
        MODEL_PARAMS(:) = 1.0_R8
        OPEN( UNIT=5, FILE="prod.nml" )
        READ( UNIT=5, NML=FIXED_NML )
        IFIXB(:) = MODEL_PARAMS(:)
        MODEL_PARAMS(:) = SAVED_PARAMS(:)
        CLOSE( UNIT=5 )

        CALL OBJ_FUNC_SET(                        &
            NEXP      = NUM_EXP,                  &
            NXDATA    = EXP_NXDATA,               &
            NYDATA    = EXP_NYDATA,               &
            EXP_XDATA = EXP_XDATA,                &
            EXP_YDATA = EXP_YDATA,                &
            EXP_NAMES = EXP_NAMES,                &
            NP        = NUM_PARAMS,               &
            L         = LOWER_BOUND,              &
            L_DEF     = L_DEF,                    &
            U         = UPPER_BOUND,              &
            U_DEF     = U_DEF,                    &
            OD_ON     = OD_ON,                    &
            WX        = WD,                       &
            WY        = WE,                       &
!            OD_AFCTOL = 1E-20_R8 ,                   &
!            OD_RFCTOL = 1E-12_R8 ,                   &
!            OD_XCTOL  = 1E-12_R8 ,                   &
            ACCEPT_UNDEF = (/ 3, 3, 3, 3, (0, I=5,NUM_XDATA) /) &
        )
        OPEN( UNIT=9, FILE="prod.REPORT")
```

```
    IF ( GLOBAL .OR. GLOBAL_ONLY ) THEN
        WRITE(*,*) "START"
        CALL VTDIRect( NUM_PARAMS, LOWER_BOUND, UPPER_BOUND, X, FMIN, &
                       STATUS, CONTINUOUS_OBJ_FUNC,   &
                       EPS = 0.30_R8, BOX_SET=BOX_SET, &
                       MAX_EVL=MAX_EVL,  &
                       NUM_BOX=NUM_BOX,                                &
                       MIN_SEP=MIN_SEP )
        WRITE(*,*) "STOP"
        WRITE(*,*) "VTDIRECT BEST: ", FMIN, " AT ", X
        WRITE(*,*) "VTDIRECT STATUS: ", STATUS
        WRITE(*,*) "VTDIRECT NUM_BOX: ", NUM_BOX
        WRITE(*,*) "VTDIRECT MIN_SEP: ", MIN_SEP
        DO I = 1, NUM_BOX
            WRITE(*,*) ""
            WRITE(*,*) "VTDIRECT BOX: ", I
            WRITE(*,*) "    VAL : ", BOX_SET(I) % VAL
            WRITE(*,*) "    C   : ", BOX_SET(I) % C(:)
            WRITE(*,*) "    SIDE: ", BOX_SET(I) % SIDE(:)
            WRITE(*,*) "    DIAM: ", BOX_SET(I) % DIAM
        END DO
        BETA = X
    END IF
    IF ( .NOT. GLOBAL_ONLY ) THEN
        CALL OBJ_FUNC_SET( U_DEF = (/ ( 0, I = 1, NUM_PARAMS ) /) )
        CALL ODRPACK_SET(                                &
            NP    = NUM_PARAMS,                          &
            BETA  = BETA,                                &
            IFIXB = IFIXB,                               &
            WD    = RESHAPE( WD, (/NUM_XDATA,1,1/) ),    &
            WE    = RESHAPE( WE, (/NUM_YDATA,1,1/) ),    &
            N     = NUM_XDATA,                           &
            X     = RESHAPE( EXP_XDATA, (/NUM_XDATA,1/) ), &
            Y     = RESHAPE( EXP_YDATA, (/NUM_YDATA,1/) ), &
            IFIXX = RESHAPE( IFIXX, (/NUM_XDATA,1/) ),   &
            IPRINT = 6616,                               &
            MAXIT = 10000,                               &
!           NDIGIT = 6,                                  &
!           PARTOL = SQRT(EPSILON(1.0_R8)),              &
            LUNRPT = 9,                                  &
            JOB    = 01000,                              &
            WORK   = RWORK                               &
        )


        CALL ODRPACK_RUN()

        ! get RWORK
        CALL ODRPACK_GET( WORK = RWORK, BETA = BETA )
        ! write namelist header
        OPEN( UNIT=8, FILE="output.nml" )
        WRITE(8,"('&OUTPUT')")
        ! write RWORK DELTAs and MODEL_PARAMS
        WRITE(8,"('RWORK(1:',I4,')=')",ADVANCE='NO') NUM_XDATA
        DO I = 1, NUM_XDATA - 1
            WRITE(8,"(E22.15,',')",ADVANCE='NO') RWORK(I)
        END DO
        WRITE(8,*) RWORK(NUM_XDATA)
        WRITE(8,"('MODEL_PARAMS(1:',I4,')=')",ADVANCE='NO') NUM_PARAMS
        DO I = 1, NUM_PARAMS - 1
            WRITE(8,"(E22.15,',')",ADVANCE='NO') BETA(I)
        END DO
        WRITE(8,*) BETA(NUM_PARAMS)
        ! write namelist footer
        WRITE(8,*) "/"
        CLOSE( UNIT=8 )

        CLOSE( UNIT=9 )
    END IF
ELSE

    MAXX(:) = (/ 1.0, 0.3, 20.0, 20.0, 20.0, 20.0, 20.0, 40.0, 20.0, 20.0 /)
    DO I = 1, NUM_EXP
        OPEN( UNIT=9, FILE=TRIM(EXP_NAMES(I))//".dat" )
        IF ( I .EQ. 2 ) THEN
            CALL MODEL_EXPERIMENT( &
                NUM_PARAMS, SAVED_PARAMS, EXP_NAMES(I),                  &
```

```
        2, 2,                                                   &
        XDATA(1:2), &
        YDATA(1:2), &
        DEFINED(1:2) &
    )
    WRITE( 9, * ) YDATA(1:2)
    OPEN( UNIT=8, FILE=TRIM(EXP_NAMES(I))//"_ACT.dat" )
    DO J = 1, NUM_POINTS
        TOTALCYCLIN = J*MAXX(I)/NUM_POINTS
        M = 0.0
        C = 0.0
        W = 1.0
        L = 0.0
        L2 = 0.0
        CALL TIMECOURSE( &
            1, 1, (/ 1440.0_R8 /), MPFACT, DEFINED(1:1), M_I &
        )
        WRITE( 8, * ) TOTALCYCLIN, MPFACT
    END DO
    CLOSE( UNIT=8 )
    OPEN( UNIT=8, FILE=TRIM(EXP_NAMES(I))//"_INACT.dat" )
    DO J = 1, NUM_POINTS
        TOTALCYCLIN = J*MAXX(I)/NUM_POINTS
        M = TOTALCYCLIN
        C = 1.0
        W = 0.0
        L = 0.0
        L2 = 0.0
        CALL TIMECOURSE( &
            1, 1, (/ 1440.0_R8 /), MPFINACT, DEFINED(1:1), M_I  &
        )
        WRITE( 8, * ) TOTALCYCLIN, MPFINACT
    END DO
    CLOSE( UNIT=8 )
ELSE
    XDATA(:) = (/ ( J*MAXX(I)/NUM_POINTS, J = 1, NUM_POINTS ) /)
    CALL MODEL_EXPERIMENT(                                      &
        NUM_PARAMS, SAVED_PARAMS, EXP_NAMES(I),                 &
        NUM_POINTS, NUM_POINTS,                                 &
        XDATA(:), &
        YDATA(:), &
        DEFINED(:) &
    )
    DO J = 1, NUM_POINTS
        WRITE( 9, * ) XDATA(J), YDATA(J)
    END DO
  END IF
  CLOSE( UNIT=9 )
END DO

END IF


END PROGRAM PROD


SUBROUTINE MODEL_RHS ( NEQ, T, Y, YDOT )
    USE REAL_PRECISION
    USE PROD_MOD
    INTEGER        :: NEQ
    REAL (KIND=R8) :: YDOT(NEQ), Y(NEQ), T

    ! Local variables.
    REAL (KIND=R8)   :: KW, KC

    !  Scaled parameters:
    REAL (KIND=R8)   :: VCP_, VCPP_, VCPPP_, &
                        VWP_, VWPP_, VWPPP_, &
                        KMC_, KMCR_,         &
                        KMW_, KMWR_,         &
                        VC_,  VW_
    !  Concentration totals
    REAL (KIND=R8)   :: CDC2T, CDC25T, WEET, PT
```

```
MODEL_VARS(1:NEQ) = Y(1:NEQ)

! Dilute concentrations
CDC2T  = DILUTION
CDC25T = DILUTION
WEET   = DILUTION
PT     = DILUTION

! Scale parameters
VCP_   = VCP   * CDC25T
VCPP_  = VCPP  * CDC25T
VCPPP_ = VCPPP * PT / CDC2T
VWP_   = VWP   * WEET
VWPP_  = VWPP  * WEET
VWPPP_ = VWPPP * PT / CDC2T
KMC_   = KMC   / CDC25T
KMCR_  = KMCR  / CDC25T
KMW_   = KMW   / WEET
KMWR_  = KMWR  / WEET
VC_    = VC    * CDC2T  / CDC25T
VW_    = VW    * CDC2T  / WEET

! DEFINE AUXILARY FUNCTIONS
KC = VCP_*(1-C) + VCPP_*C
KW = VWP_*(1-W) + VWPP_*W

! DEFINE ODES
MDOT = KC*(TOTALCYCLIN-M) - KW*M
CDOT = VC_ * (                                          &
             + ( M      * ( 1 - C ) ) / ( KMC_  + ( 1 - C ) ) &
             - ( VCPPP_ *       C   ) / ( KMCR_ +       C   ) &
             )
WDOT = VW_ * (                                          &
             - ( M      *       W   ) / ( KMW_  +       W   ) &
             + ( VWPPP_ * ( 1 - W ) ) / ( KMWR_ + ( 1 - W ) ) &
             )
LDOT = -KC*L
L2DOT = KW * ( 1 - L2 )

    !  END USER DEFINE MODEL

    YDOT(1:NEQ) = MODEL_DERS(1:NEQ)

    RETURN

END SUBROUTINE MODEL_RHS




SUBROUTINE MODEL_EXPERIMENT ( NP, PARAMS, EXP_NAME, &
                              NX, NY, XDATA, YDATA, &
                              DEFINED               &
)
    USE PROD_MOD
    USE REAL_PRECISION
    USE STRINGS
    INTEGER, INTENT(IN)            :: NP, NX, NY
    CHARACTER (MAXSTR), INTENT(IN) :: EXP_NAME
    REAL (KIND=R8), INTENT(IN)     :: PARAMS(NP), XDATA(NX)
    REAL (KIND=R8), INTENT(OUT)    :: YDATA(NY)
    INTEGER, INTENT(OUT)           :: DEFINED(NY)

    ! local variables
    INTEGER :: I

!   WRITE(*,*) "CALC EXP: ", TRIM(EXP_NAME)

    MODEL_PARAMS(1:NP) = PARAMS(1:NP)
    DEFINED(1:NY) = 0

    M  = 0.0_R8
    C  = 0.0_R8
    W  = 1.0_R8
    L  = 1.0_R8
    L2 = 0.0_R8
```

```
    TOTALCYCLIN = 0.25_R8

    !WRITE(*,*) "CALCULATING: ", TRIM(EXP_NAME)

    !-----------------------------------------------------------------------
    !
    !  MOORE_TIMELAG
    !
    IF     ( EXP_NAME .EQ. EXP_NAMES(1) ) THEN
        MODEL_EXP_COUNT = MODEL_EXP_COUNT + 1
!        WRITE(*,"(16E11.3)") MODEL_PARAMS
        IF ( NX .NE. NY ) THEN
            WRITE( 6, * ) "WARNING: FOR MOORE_TIMELAG THE NUMBER OF X AND Y",&
                " EXP DATA SHOULD BE EQUAL AND IS NOT: NX = ", NX, " NY = ", NY
        END IF
        DO I = 1, MIN(NX,NY)
            M  = 0.0_R8
            C  = 0.0_R8
            W  = 1.0_R8
            L  = 0.0_R8
            L2 = 0.0_R8
            TOTALCYCLIN = XDATA(I)
            YDATA(I) = TIMELAG(DEFINED(I))
        END DO
    !
    !-----------------------------------------------------------------------

    !-----------------------------------------------------------------------
    !
    !  MOORE_THRESHOLD
    !
    ELSEIF ( EXP_NAME .EQ. EXP_NAMES(2) ) THEN
        ! Calculate the upper threshold.
        IF ( NY .GT. 0 ) THEN
            L  = 0.0_R8
            L2 = 0.0_R8
            YDATA(1) = THRESHOLD(DEFINED(1))
        END IF
        ! Calculate the ratio between the thresholds.
        IF ( NY .GT. 1 ) THEN
            TOTALCYCLIN = 1.0_R8
            M  = 1.0_R8
            C  = 1.0_R8
            W  = 0.0_R8
            L  = 0.0_R8
            L2 = 0.0_R8
            YDATA(2) = YDATA(1)/THRESHOLD(DEFINED(2))
        END IF
        IF ( NY .GT. 2 ) THEN
            WRITE( 6, * ) "WARNING: FOR MOORE_THRESHOLD THERE CAN ONLY BE TWO",&
                " Y DATA POINTS, THERE ARE ", NY
        END IF
    !
    !-----------------------------------------------------------------------

    !-----------------------------------------------------------------------
    !
    !  KUMAGAI_AND_DUMPHY_MPF_ACT_MPHASE
    !
    ELSEIF ( EXP_NAME .EQ. EXP_NAMES(3) ) THEN
        DILUTION = 1.0_R8
        M = 1.0_R8
        C = 1.0_R8
        W = 0.0_R8
        TOTALCYCLIN = 1.0_R8
        CALL TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, L_I )
    !
    !-----------------------------------------------------------------------

    !-----------------------------------------------------------------------
    !
    !  KUMAGAI_AND_DUMPHY_MPF_ACT_INTERPHASE
    !
    ELSEIF ( EXP_NAME .EQ. EXP_NAMES(4) ) THEN
        DILUTION = 1.0_R8
        TOTALCYCLIN = 0.0_R8
        CALL TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, L_I )
    !
```

```
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!
!  KUMAGAI_AND_DUMPHY_MPF_INACT_MPHASE
!
ELSEIF ( EXP_NAME .EQ. EXP_NAMES(5) ) THEN
    DILUTION = 1.0_R8
    M = 1.0_R8
    C = 1.0_R8
    W = 0.0_R8
    TOTALCYCLIN = 1.0_R8
    CALL TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, L2_I )
!
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!
!  KUMAGAI_AND_DUMPHY_MPF_INACT_INTERPHASE
!
ELSEIF ( EXP_NAME .EQ. EXP_NAMES(6) ) THEN
    DILUTION = 1.0_R8
    TOTALCYCLIN = 0.0_R8
    CALL TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, L2_I )
!
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!
!  KUMAGAI_AND_DUMPHY_CDC25_ACT
!
ELSEIF ( EXP_NAME .EQ. EXP_NAMES(7) ) THEN
    DILUTION = 0.83_R8
    TOTALCYCLIN = 0.25_R8
    M = 0.25_R8
    W = 0.0_R8
    CALL TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, C_I )
!
!-----------------------------------------------------------------------
```

```
!-----------------------------------------------------------------------
!
!  KUMAGAI_AND_DUMPHY_CDC25_INACT
!
ELSEIF ( EXP_NAME .EQ. EXP_NAMES(8) ) THEN
    DILUTION = 0.95_R8
    TOTALCYCLIN = 0.0_R8
    M = 0.0_R8
    C = 1.0_R8
    W = 0.0_R8
    CALL TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, C_I )
!
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!
!  TANG_WEE1_INACT
!
ELSEIF ( EXP_NAME .EQ. EXP_NAMES(9) ) THEN
    DILUTION = 0.67_R8
    TOTALCYCLIN = 0.2537_R8
    M = 0.2537_R8
    C = 1.0_R8
    W = 1.0_R8
    CALL TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, W_I )
!
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!
!  TANG_WEE1_ACT
!
ELSEIF ( EXP_NAME .EQ. EXP_NAMES(10) ) THEN
    DILUTION = 0.67_R8
    TOTALCYCLIN = 0.0_R8
    M = 0.0_R8
    C = 1.0_R8
    W = 0.0_R8
```

```
        CALL TIMECOURSE( NX, NY, XDATA, YDATA, DEFINED, W_I )
    !
    !-----------------------------------------------------------------------

    ELSE
        WRITE( 6, * ) "ERROR: UNKNOWN EXPERIMENT: ", TRIM(EXP_NAME)
    END IF

    IF ( ANY( DEFINED(1:NY) .EQ. 0 ) .AND. EXP_NAME .NE. EXP_NAMES(1) ) THEN
        WRITE(0,*) TRIM(EXP_NAME), " UNDEFINED: ", DEFINED(1:NY)
    END IF

END SUBROUTINE MODEL_EXPERIMENT


! Find the root M=TOTALCYCLIN/2.  This time at this root is the timelag for MPF
!  activation, by definition.
SUBROUTINE MODEL_ROOTS ( NEQ, T, Y, NG, GOUT )
    USE PROD_MOD
    USE REAL_PRECISION
    INTEGER        :: NEQ, NG
    REAL (KIND=R8) :: T, Y(NEQ), GOUT(NG)

    MODEL_VARS(1:NEQ) = Y(1:NEQ)

    GOUT(1) = M - MPF_THRESHOLD
    IF ( T > 121 .AND. T < 122 ) THEN
!       WRITE(*,*) "GOUT = ", GOUT(1), M, MPF_THRESHOLD
    END IF

END SUBROUTINE MODEL_ROOTS
```

```
!  There are no actions for roots found in this model
SUBROUTINE MODEL_FOUND_ROOT ( NEQ, T, Y, NG, JROOT )
    USE REAL_PRECISION
    INTEGER, INTENT(IN)           :: NEQ, NG, JROOT(NG)
    REAL (KIND=R8), INTENT(IN)    :: T
    REAL (KIND=R8), INTENT(INOUT) :: Y(NEQ)
END SUBROUTINE MODEL_FOUND_ROOT


!  There are no auxiliary variables for this model
SUBROUTINE MODEL_AUXILIARY ( NA, AUX, NV, VARS, T )
    USE REAL_PRECISION
    INTEGER, INTENT(IN) :: NA, NV
    REAL (KIND=R8), INTENT(OUT) :: AUX(NA)
    REAL (KIND=R8), INTENT(IN)  :: VARS(NV), T
END SUBROUTINE MODEL_AUXILIARY
```

strings.f90

```
MODULE STRINGS

INTEGER, PARAMETER :: MAXSTR = 65000

END MODULE STRINGS
```

transforms.f90

```
MODULE TRANSFORMS

CONTAINS



!-----------------------------------------------------------------------------
!
!  TIMECOURSE_INIT
!
!  This routine generates experimental data and associated meta
!  data arrays suitable for ODRPACK.  This routine generates data for the
!  TIMECOURSE_RUN subroutine.
!
!  TIMECOURSE_INIT will copy the experimental data defined in
!  EXP_DATA_IN into the arrays X_EXP_DATA_OUT and Y_EXP_DATA_OUT.  The first
!  column of EXP_DATA_IN is copied to X_EXP_DATA_OUT and the second is copied to
!  Y_EXP_DATA_OUT.  Similarly the weights are copied from WEIGHTS_IN to
!  X_WEIGHTS_OUT and Y_WEIGHTS_OUT.  IFIXX is set to all 1s.
!
!
!  PARAMETERS
!  ----------
!
! KEY
!
!   ->  IN
!  <-   OUT
!  <->  IN and OUT
!   S   Must be set (a standard parameter).
!   O   Optional input.
!
!-----------------------------------------------------------------------------
!  -> O  LDATA                  Length of EXP_DATA_IN, WEIGHTS_IN,
!                               X_EXP_DATA_OUT, Y_EXP_DATA_OUT, X_WEIGHTS_OUT,
```

```
!                               Y_WEIGHTS_OUT, and IFIXX.
!  -> O  EXP_DATA_IN(:,:)       Contains the experimental data from JigCell.
!                               The input is assumed to be (n:2) where n>=1.
!  -> O  WEIGHTS_IN(:,:)        WEIGHTS_IN(i,j) contains the weight for the
!                               experimental data EXP_DATA_IN(i,j).
! <-  O  LX, LY                 Contains the resulting lengths of
!                               X_EXP_DATA_OUT, Y_EXP_DATA_OUT, X_WEIGHTS_OUT,
!                               Y_WEIGHTS_OUT, and IFIXX after they have been
!                               allocated by TIMECOURSE.  For TIMECOURSE, LX and
!                               LY will always be equal to LDATA.
! <-  O  X_EXP_DATA_OUT(:)      X_EXP_DATA_OUT(i) = EXP_DATA_IN(i,1) and is
!                               suitable for ODRPACK.
! <-  O  Y_EXP_DATA_OUT(:)      Y_EXP_DATA_OUT(i) = EXP_DATA_IN(i,2) and is
!                               suitable for ODRPACK.
! <-  O  X_WEIGHTS_OUT(:)       Y_WEIGHTS_OUT(i) = WEIGHTS_IN(i,1) and is
!                               suitable for ODRPACK.
! <-  O  Y_WEIGHTS_OUT(:)       Y_WEIGHTS_OUT(i) = WEIGHTS_IN(i,1) and is
!                               suitable for ODRPACK.
! <-  O  IFIXX(:)               IFIXX(i) = 1 for i = 1 to n where n is the size
!                               of EXP_DATA_IN and WEIGHTS_IN.
!-------------------------------------------------------------------------------
SUBROUTINE TIMECOURSE_INIT( &
    LDATA,                  &
    EXP_DATA_IN,            &
    WEIGHTS_IN,             &
    LX,                     &
    LY,                     &
    X_EXP_DATA_OUT,         &
    Y_EXP_DATA_OUT,         &
    X_WEIGHTS_OUT,          &
    Y_WEIGHTS_OUT,          &
    IFIXX                   &
)
    USE REAL_PRECISION
    INTEGER,        INTENT(IN) , OPTIONAL :: LDATA
    REAL (KIND=R8), INTENT(IN) , OPTIONAL :: EXP_DATA_IN(:,:)
    REAL (KIND=R8), INTENT(IN) , OPTIONAL :: WEIGHTS_IN(:,:)
    INTEGER       , INTENT(OUT), OPTIONAL :: LX, LY
    REAL (KIND=R8)             , OPTIONAL :: X_EXP_DATA_OUT(:)
    REAL (KIND=R8)             , OPTIONAL :: Y_EXP_DATA_OUT(:)
    REAL (KIND=R8)             , OPTIONAL :: X_WEIGHTS_OUT(:)
    REAL (KIND=R8)             , OPTIONAL :: Y_WEIGHTS_OUT(:)
    INTEGER                    , OPTIONAL :: IFIXX(:)
    POINTER :: X_EXP_DATA_OUT, Y_EXP_DATA_OUT, X_WEIGHTS_OUT, Y_WEIGHTS_OUT, &
               IFIXX

    IF ( &
        ASSOCIATED( X_EXP_DATA_OUT ) .OR. &
        ASSOCIATED( Y_EXP_DATA_OUT ) .OR. &
        ASSOCIATED( X_WEIGHTS_OUT  ) .OR. &
        ASSOCIATED( Y_WEIGHTS_OUT  ) .OR. &
        ASSOCIATED( IFIXX          )      &
    ) THEN
        WRITE(O,*) "WARNING: TIMECOURSE: THE VARIABLES X_EXP_DATA_OUT, &
                   &Y_EXP_DATA_OUT, X_WEIGHTS_OUT, Y_WEIGHTS_OUT, AND &
                   &IFIXX ARE ASSOCIATED IN TIMECOURSE.  &
                   &AT LEAST ONE OF THEM WAS PASSED TO TIMECOURSE ALREADY &
                   &ASSOCIATED."
    END IF
    ALLOCATE( X_EXP_DATA_OUT(LDATA), Y_EXP_DATA_OUT(LDATA) )
    ALLOCATE( X_WEIGHTS_OUT(LDATA),  Y_WEIGHTS_OUT(LDATA)  )
    ALLOCATE( IFIXX(LDATA) )
    X_EXP_DATA_OUT(1:LDATA) = EXP_DATA_IN(1:LDATA,1)
    Y_EXP_DATA_OUT(1:LDATA) = EXP_DATA_IN(1:LDATA,2)
    X_WEIGHTS_OUT(1:LDATA)  = WEIGHTS_IN(1:LDATA,1)
    Y_WEIGHTS_OUT(1:LDATA)  = WEIGHTS_IN(1:LDATA,2)
    IFIXX(1:LDATA) = 1
    LX = LDATA
    LY = LDATA

END SUBROUTINE TIMECOURSE_INIT
```

```
!-------------------------------------------------------------------------------
!
!  TIMECOURSE_RUN
!
!  The Fortran version of the JigCell transform for time course data.
!  This routine calculates time courses and returns the requested time points.
!
!  The model is evaluated for all times in TIMES and the values of
!  MODEL_VARS(VAR_INDEX) stored in YOUT(I) for each TIME(I).
!
!
!  PARAMETERS
!  ----------
!
! KEY
!
!   ->  IN
!  <-   OUT
!  <->  IN and OUT
!   S   Must be set (a standard parameter).
!   O   Optional input.
!
!-------------------------------------------------------------------------------
!  ->    VAR_INDEX              An index into MODEL_VARS.  MODEL_VARS(VAR_INDEX)
!                               will be stored in YOUT as described above.
!  ->    LTIMES                 Length of TIMES, must be specified when TIMES is
!                               specified.
!  ->    LYOUT                  Specifies the length of YOUT.
!  ->    TIMES(:)               Contains the times where a solution to the ODE
!                               system is desired.
! <-     YOUT(:)                Contains the solutions to the ODE system at
!                               TIMES(:) for the variable MODEL_VARS(VAR_INDEX).
!                               YOUT(I) = MODEL_VARS(VAR_INDEX) at TIME(I).
! <-     DEFINED(:)             DEFINED(I) .EQ. 1 if YOUT(I) is defined for
!                               TIMES(I).  DEFINED(I) .EQ. 0 if YOUT(I) is not
!                               defined for TIMES(I).
!-------------------------------------------------------------------------------
SUBROUTINE TIMECOURSE_RUN( &
    LTIMES,                &
    LYOUT,                 &
    TIMES,                 &
    YOUT,                  &
    DEFINED,               &
    VAR_INDEX,             &
    LMODEL_VARS,           &
    MODEL_VARS             &
)
    USE REAL_PRECISION
    USE LSODAR_MOD
    INTEGER       , INTENT(IN)  :: LTIMES
    INTEGER       , INTENT(IN)  :: LYOUT
    REAL (KIND=R8), INTENT(IN)  :: TIMES(LTIMES)
    REAL (KIND=R8), INTENT(OUT) :: YOUT(LYOUT)
    INTEGER       , INTENT(OUT) :: DEFINED(LYOUT)
    INTEGER       , INTENT(IN)  :: VAR_INDEX
    INTEGER       , INTENT(IN)  :: LMODEL_VARS
    REAL (KIND=R8), INTENT(IN)  :: MODEL_VARS(LMODEL_VARS)

    ! Local variables
    INTEGER :: I, ISTATE = 0
    REAL (KIND=R8) :: MODEL_VARS_(LMODEL_VARS), CURRENT_TIME

    MODEL_VARS_(:) = MODEL_VARS(:)

    DEFINED(:) = 0

    IF ( LTIMES .NE. LYOUT ) THEN
        WRITE( 0, * ) "WARNING: TIMECOURSE_RUN: TIMECOURSE CALCULATIONS REQUIRE&
                      & THE &
                      &NUMBER OF X AND Y DATA TO BE EQUAL. LTIMES = ", LTIMES, &
                      " LYOUT = ", LYOUT
    END IF

    IF ( MIN( LTIMES, LYOUT ) .LE. 0 ) THEN
        WRITE( 0, * ) "ERROR: TIMECOURSE_RUN: AT LEAST ONE TIMECOURSE &
                      &CALCULATION MUST BE &
                      &PERFORMED.  LTIMES OR LYOUT IS EQUAL TO ZERO."
        RETURN
```

```
END IF

IF ( VAR_INDEX .LT. 1 .OR. VAR_INDEX .GT. LMODEL_VARS ) THEN
    WRITE( 0, * ) "ERROR: TIMECOURSE_RUN: VAR_INDEX IS OUT OF RANGE WITH A &
                  &VALUE OF ", &
                  VAR_INDEX, ".  VAR_INDEX SHOULD OBEY 1 <= VAR_INDEX <= &
                  &LMODEL_VARS. "
    RETURN
END IF

CALL LSODAR_RECALL()
CALL LSODAR_SET( Y = MODEL_VARS_(:) )
CURRENT_TIME = 0.0_R8
DO I = 1, MIN( LTIMES, LYOUT )
    IF ( CURRENT_TIME .EQ. TIMES(I) ) THEN
        YOUT(I) = MODEL_VARS_(VAR_INDEX)
        IF ( ISTATE .EQ. 2 .OR. I .EQ. 1 ) THEN
            DEFINED(I) = 1
        END IF
        CONTINUE
    END IF
    IF ( CURRENT_TIME .GT. TIMES(I) ) THEN
        WRITE(0,*) "ERROR: TIMECOURSE_RUN: TIME(I) IS LESS THAN THE CURRENT&
                   & INTEGRATION TIME; CANNOT INTEGRATE.  YOUT(I) IS SET TO&
                   & 0 AND DEFINED IS FALSE (0).  SKIPPING TO THE NEXT &
                   &TIME."
        YOUT(I) = 0.0_R8
        DEFINED(I) = 0
        CONTINUE
    END IF
    CALL LSODAR_SET( TOUT = TIMES(I) )
    CALL LSODAR_RUN()
    CALL LSODAR_GET( ISTATE = ISTATE, Y = MODEL_VARS_, T = CURRENT_TIME )
    DO WHILE ( ISTATE .EQ. 3 )
        CALL LSODAR_RUN()
        CALL LSODAR_GET( ISTATE = ISTATE, Y = MODEL_VARS_, T = CURRENT_TIME)
    END DO
    YOUT(I) = MODEL_VARS_(VAR_INDEX)
    IF ( ISTATE .EQ. 2 ) THEN
        DEFINED(I) = 1
    END IF
END DO

END SUBROUTINE TIMECOURSE_RUN


END MODULE TRANSFORMS
```

vtdirect_mod.f90

```
MODULE VTDIRECT_MOD
!==============================================================================
!
!  VTDIRECT_MOD
!
!  This module provides a wrapper to VTDirect with some additional
!  functionality.  The user must supply the routines MODEL_EXPERIMENTS or
!  MODEL_OBJECTIVE.  The user supplies experimental data and parameter ranges.
!
!  The user may optionally let VTDIRECT_MOD use its own internal objective
!  function.  The internal objective function calls MODEL_EXPERIMENTS to
!  calculate simulated experiments used to calculate the objective function
!  value at a point in parameter space.  This function is defined in
!  obj_func_mod.
!
!  If the user specifies MODEL_OBJECTIVE then MODEL_EXPERIMENTS need not be
!  specified.  The user may call his/her own MODEL_EXPERIMENTS or just perform
!  the necessary calculations in MODEL_OBJECTIVE.
!
!  Documentation copied and editted from VTDirect and applies to this wrapper:
!
! On input:
!
! N is the dimension of L, U, and X.
!
! L(1:N) is a real array giving lower bounds on X.
!
! U(1:N) is a real array giving upper bounds on X.
!
! OBJ_FUNC is the name of the real function procedure defining the
!    objective function f(x) to be minimized. OBJ_FUNC(C,IFLAG) returns
!    the value f(C) with IFLAG=0, or IFLAG .NE. 0 if f(C) is not defined.
!    OBJ_FUNC is precisely defined in the INTERFACE block below.
!
! Optional arguments:
!
! SWITCH =
!    1   select potentially optimal boxes on the convex hull of the
!        (box diameter, function value) points (default).
!    0   select as potentially optimal the box of each diameter with the
!        smallest function value. This is an aggressive selection
!        procedure which generates many more boxes to subdivide.
!
! MAX_ITER is the maximum number of iterations (repetitions of Steps 2-3)
!    allowed; defines stopping rule 1. If MAX_ITER is present but <= 0
!    on input, there is no iteration limit and the number of iterations
!    executed is returned in MAX_ITER.
!
! MAX_EVL is the maximum number of function evaluations allowed; defines
!    stopping rule 2. If MAX_EVL is present but <= 0 on input, there is no
!    limit on the number of function evaluations, which is returned in
!    MAX_EVL.
!
! MIN_DIA is the minimum box diameter allowed; defines stopping rule 3.
!    If MIN_DIA is present but <= 0 on input, a minimum diameter below
!    the roundoff level is not permitted, and the box diameter of the
!    box containing the smallest function value FMIN is returned in MIN_DIA.
!
! OBJ_CONV is the smallest acceptable relative improvement in the minimum
!    objective function value 'FMIN' between iterations; defines
!    stopping rule 4. OBJ_CONV must be positive and greater than the round
!    off level.  If absent, it is taken as zero.
!
! EPS is the tolerance defining the minimum acceptable potential
!    improvement in a potentially optimal box.  Larger EPS values eliminate
!    more boxes from consideration as potentially optimal, and bias the
!    search toward exploration.  EPS must be positive and greater than the
!    round off level.  If absent, it is taken as zero.  EPS > 0 is
!    incompatible with SWITCH = 0.
!
! MIN_SEP is the specified minimal distance between the center points of
!    the boxes returned in the optional array BOX_SET. If absent,
!    MIN_SEP is taken as 1/2 the diameter of the box [L, U].
!
! On output:
!
```

```
! X(1:N) is a real vector containing the sampled box center with the
!    minimum objective function value FMIN.
!
! FMIN is the minimum function value.
!
! STATUS is a return status flag. The units decimal digit specifies
!    which stopping rule was satisfied on a successful return. The tens
!    decimal digit indicates a successful return, or an error condition with
!    the cause of the error condition reported in the units digit.
!
! Tens digit =
!  0 Normal return.
!    Units digit =
!     1   Stopping rule 1 (iteration limit) satisfied.
!     2   Stopping rule 2 (function evaluation limit) satisfied.
!     3   Stopping rule 3 (minimum diameter reached) satisfied. The
!         minimum diameter corresponds to the box for which X and
!         FMIN are returned.
!     4   Stopping rule 4 (relative change in 'FMIN') satisfied.
!  1 Input data error.
!    Units digit =
!     0   N < 2.
!     1   Assumed shape array L, U, or X does not have size N.
!     2   Some lower bound is >= the corresponding upper bound.
!     3   MIN_DIA, OBJ_CONV, or EPS is below the roundoff level.
!     4   None of MAX_EVL, MAX_ITER, MIN_DIA, and OBJ_CONV are specified;
!         there is no stopping rule.
!     5   Invalid SWITCH value.
!     6   SWITCH = 0 and EPS > 0 are incompatible.
!  2 Memory allocation failure.
!    Units digit =
!     0   BoxMatrix type allocation.
!     1   BoxLink type allocation.
!     2   int_vector or real_vector type allocation.
!     3   HyperBox type allocation.
! MAX_ITER (if present) contains the number of iterations.
!
! MAX_EVL (if present) contains the number of function evaluations.
!
! MIN_DIA (if present) contains the diameter of the box associated with
!         X and FMIN.
!
! Optional arguments:
!
! MIN_SEP (if present) is unchanged if it was a reasonable value on
!         input. Otherwise, it is reset to the default value.
!
! BOX_SET (if present) is an array of TYPE (HyperBox) containing the
!          best boxes with centers separated by at least MIN_SEP.
!          The number of returned boxes NUM_BOX <= SIZE(BOX_SET) is as
!          large as possible given the requested separation.
!
! NUM_BOX (if present) is the number of boxes returned in the array
!         BOX_SET(1:).
!
!  End documentation from VTDirect.
!

PRIVATE

CONTAINS

SUBROUTINE VTDIRECT_RUN(                                        &
                   N, L, U, X, FMIN, STATUS, OBJ_FUNC, SWITCH, &
                   MAX_ITER, MAX_EVL, MIN_DIA, OBJ_CONV, EPS,  &
```

```
                    MIN_SEP, BOX_SET, NUM_BOX                    &
)

USE OBJ_FUNC_MOD

IMPLICIT NONE

INTEGER, INTENT(IN) :: N
REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: L
REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: U
REAL(KIND = R8), DIMENSION(:), INTENT(OUT) :: X
REAL(KIND = R8), INTENT(OUT) :: FMIN
INTEGER, INTENT(OUT) :: STATUS

INTERFACE
  FUNCTION OBJ_FUNC(C, IFLAG) RESULT(F)
    USE REAL_PRECISION, ONLY : R8
    REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: C
    INTEGER, INTENT(OUT) :: IFLAG
    REAL(KIND = R8) :: F
  END FUNCTION OBJ_FUNC
END INTERFACE

INTEGER, INTENT(IN), OPTIONAL :: SWITCH
INTEGER, INTENT(INOUT), OPTIONAL :: MAX_ITER
INTEGER, INTENT(INOUT), OPTIONAL :: MAX_EVL
REAL(KIND = R8), INTENT(INOUT), OPTIONAL :: MIN_DIA
REAL(KIND = R8), INTENT(IN), OPTIONAL :: OBJ_CONV
REAL(KIND = R8), INTENT(IN), OPTIONAL :: EPS
REAL(KIND = R8), INTENT(INOUT), OPTIONAL :: MIN_SEP
TYPE(HyperBox), DIMENSION(:), INTENT(INOUT), OPTIONAL :: BOX_SET
INTEGER, INTENT(OUT), OPTIONAL :: NUM_BOX

! Local vars
INTEGER :: N_I, I
REAL (KIND=R8) :: L_I(N), U_I(N), X_I(N)
INTEGER :: EST_PARAMS(N)
REAL (KIND=R8) :: PARAMS(N)

CALL OBJ_FUNC_GET( EST_PARAMS = EST_PARAMS, PARAMS = PARAMS )

N_I = 0
DO I = 1, N
    IF ( EST_PARAMS(I) .EQ. 1 ) THEN
        N_I = N_I + 1
        L_I(N_I) = L(I)
        U_I(N_I) = U(I)
    END If
END DO

CALL VTDIRECT(                                          &
                    N_I, L_I, U_I, X_I, FMIN, STATUS, OBJ_FUNC, SWITCH, &
                    MAX_ITER, MAX_EVL, MIN_DIA, OBJ_CONV, EPS,  &
                    MIN_SEP, BOX_SET, NUM_BOX                   &
)

N_I = 0
DO I = 1, N
    IF ( EST_PARAMS(I) .EQ. 1 ) THEN
        N_I = N_I + 1
        X(I) = X_I(N_I)
    ELSE
        X(I) = PARAMS(I)
    END IF
END DO

END SUBROUTINE

END MODULE VTDIRECT_MOD
```

VITA

Jason W. Zwolak was born on May 14, 1977, in Hartford, Connecticut. He earned a Bachelor of computer engineering degree from Virginia Tech in 1999. In August 1999 he began graduate work in the Computer Science department at Virginia Tech. He completed a masters of computer science in October of 2001 and then a doctorate of computer science in December of 2004.