Some Methods and Models for Analyzing Time-Series Gene Expression Data

by

Arvind K. Jammalamadaka

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author ....
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by ....
David K. Gifford
Professor
Thesis Supervisor

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

Some Methods and Models for Analyzing Time-Series Gene Expression Data

by

Arvind K. Jammalamadaka

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Experiments in a variety of fields generate data in the form of a time-series. Such time-series profiles, collected sometimes for tens of thousands of experiments, are a challenge to analyze and explore. In this work, motivated by gene expression data, we provide several methods and models for such analysis. The methods developed include new clustering techniques based on nonparametric Bayesian procedures, and a confirmatory methodology to validate that the clusters produced by any of these methods have statistically different mean paths.

Thesis Supervisor: David K. Gifford
Title: Professor

Acknowledgments

I express my deep sense of gratitude to my advisor, Professor David Gifford, who provided me the moral and intellectual support I needed, and judiciously nudged me to the completion of this work. I also thank Professor Tommi Jaakkola for introducing me to various topics in machine learning and for suggesting several improvements to the current thesis. Any statement about my days at MIT would not be complete without mentioning the solid support I received from Professor George Verghese, whose advice for my Master's thesis and beyond was very crucial in my intellectual development. I have also received helpful advice in my work from Dr. Kaushik Ghosh of University of Nevada at Las Vegas, continuing discussions with whom helped clarify the ideas in many places. I cannot imagine having a better group of advisors and friends and I thank them all. Finally, I would like to thank my family for their love and affection, and for encouraging me to excel in whatever I do.

Contents

1 Introduction and Overview **17**

2 Dirichlet Process Mixtures and Clustering **21**

2.1 Introduction 21

2.2 The Dirichlet Process and Some Properties 22

2.3 Dirichlet Process Mixtures and Clustering 24

2.4 Clustering in a Dirichlet Process 27

3 Clustering Based on Trajectories **29**

3.1 Introduction 29

3.2 Trajectory Models 29

3.2.1 Basic Trajectory Model 31

3.2.2 A General Trajectory Model 31

3.3 Prior information 32

3.4 Computational Details 34

3.4.1 Posterior conditionals 35

3.5 Data Analysis 37

3.5.1 Synthetic Data 37

3.5.2 Real Data 43

4 Clustering Based on Expression Profiles **47**

4.1 Introduction 47

4.2 Model description 47

4.2.1 Heteroscedastic model 48

4.2.2 Homoscedastic model 51

4.3 Data Analysis 53

4.3.1 Synthetic Data 53

4.3.2 Real Data 54

4.4 Conclusion 63

5 Clustering Based on Distributions 65

5.1 Introduction 65

5.2 Clustering of Distributions with Nested Dirichlet Process 65

5.2.1 Introduction 65

5.2.2 The Nested Dirichlet Process 66

5.2.3 Nested Clustering Versus Biclustering 68

5.3 Clustering Based on Distributions of Slope Parameters 71

5.3.1 The Model 71

5.3.2 Computational Details 74

5.3.3 Guidelines for the Choice of Prior Parameters 80

5.3.4 Data Analysis 82

6 Growth Curve Models 93

6.1 Analysis of growth curves 96

6.2 Fitting Growth Curves 97

6.3 Growth Curve Analysis of Yeast Cell-Cycle Data 99

7 Conclusions and Future Work 103

7.1 Summary 103

7.2 Analysis and Discussion 105

7.3 Further work and possible extensions 109

A Code 111

A.1 Chap. 3 111

A.1.1 Basic Trajectory Model 111

A.1.2 General Trajectory Model 130
A.2 Chap. 4 153
A.2.1 Homoscedastic model 153
A.2.2 Heteroscedastic model 168
A.3 Chap. 5 184
A.3.1 NDP Model 184
A.3.2 Slope model 186
A.4 Chap. 6 195

List of Figures

3-1 Heatmap of clusters of the synthetic data based on the average incidence matrix from the basic trajectory model. Higher intensity of red denotes more likelihood of clustering together. 39

3-2 Heatmap of clusters of the synthetic data based on the average incidence matrix from the general trajectory model. Higher intensity of red denotes more likelihood of clustering together. 40

3-3 Estimated parameters for class 1. Along the x-axis, in order, are $[\delta_{i0}, \delta_{i1}, \beta_{i1}, \cdots, \beta_{i8}]$. 41

3-4 Estimated parameters for class 2. Along the x-axis, in order, are $[\delta_{i0}, \delta_{i1}, \beta_{i1}, \cdots, \beta_{i8}]$. 42

3-5 Estimated parameters for class 3. Along the x-axis, in order, are $[\delta_{i0}, \delta_{i1}, \beta_{i1}, \cdots, \beta_{i8}]$. 42

3-6 Histogram of the number of clusters generated using the trajectory model. 44

3-7 Heatmap of clusters of the 798 genes based on the average incidence matrix from the trajectory model. Higher intensity of red denotes more likelihood of clustering together. 45

3-8 Mean curves of top 5 clusters found by the trajectory model. 46

4-1 Heatmap of clusters of the synthetic data based on the average incidence matrix from the heteroscedastic model. Higher intensity of red denotes more likelihood of clustering together. 54

4-2 Heatmap of clusters of the synthetic data based on the average incidence matrix from the homoscedastic model. Higher intensity of red denotes more likelihood of clustering together. 55

4-3 Histogram of number of clusters generated using the homoscedastic model. 56

4-4 Heatmap of clusters of the 798 genes based on the average incidence matrix for the homoscedastic model. Higher intensity of red denotes more likelihood of clustering together. 57

4-5 Mean curves of top 5 clusters found by the homoscedastic model. . . 58

4-6 Histogram of number of clusters generated using the heteroscedastic model. 59

4-7 Heatmap of clusters of the 798 genes based on the average incidence matrix for the heteroscedastic model. Higher intensity of red denotes more likelihood of clustering together. 60

4-8 Mean curves of top 5 clusters found by the heteroscedastic model. . . 61

4-9 Mean curves of top 5 clusters found by the heteroscedastic model with hyperparameters set for a mean of 5 clusters. 62

5-1 Heat map of average incidence matrix for nested model. 69

5-2 Mean curves of top 5 clusters found by the nested model. 70

5-3 Autocorrelation of selected variables. 86

5-4 Cross-correlation of selected variables. 87

5-5 Geweke Z-scores for selected variables. 88

5-6 Dendrogram for synthetic data. 89

5-7 Histogram of number of clusters generated using the slope-based model. 90

5-8 Heatmap of clusters of the 798 genes based on the average incidence matrix for the slope-based model. Higher intensity of red denotes more likelihood of clustering together. 91

5-9 Mean curves of top 5 clusters found by the slope-based model. 92

6-1 Observed Means and Fitted Means with $q = 6$ 100

6-2 Model-based Mean curves for the 5 groups 101

7-1 Mean curves of 5 clusters found by the spline model of Bar-Joseph, et al. 108

List of Tables

3.1 Synthetic data trajectory parameters. 38

3.2 Mean squared error for trajectory parameter estimates on synthetic data. The columns represent 1) small variance, small sample size, 2) small variance, large sample size, 3) large variance, small sample size, and 4) large variance, large sample size. 43

6.1 MANOVA for Test of Specification 98

6.2 MANOVA table for Testing H_1 99

7.1 Pairwise Rand scores for different models. 107

7.2 Pairwise Jaccard scores for different models. 107

7.3 Distance from mean for different models. 109

Chapter 1

Introduction and Overview

Experimental results often take the form of a time series. This happens in so-called longitudinal studies or correlational studies that involve repeated observations of the same items or subjects over periods of time. Longitudinal studies are used in psychology to study developmental trends over time, in sociology to study events throughout lifetimes or generations, and in microarray experiments where the expression level of the same gene is observed at different time-points. Our goal in this work is to deal with such time-series data and help cluster subjects with similar profiles over time.

For definiteness, we will relate our ideas to gene expression profiles although they are applicable to any such longitudinal study. Through the use of expression (and other) data, biologists have been trying to arrange genes into functionally related groups, and cluster analysis has been used as an important tool for identifying groups of genes with similar expression patterns, which in turn could point to the discovery of molecular mechanisms that underlie the biological processes. Such clustering methods provide us with an important first attempt at modeling the underlying biological processes that regulate the functioning of these genes and are often the key to understanding cell mechanisms.

Time-series gene expression experiments have been used to identify the complete set of genes that participate in the system over time as well as infer causal relationships and interactions among these genes (see [42]). Time-course microarray experiments remain quite common ([60]), and as [49] report, a large fraction– over 30%– of gene expression experiments are indeed time-series studies (Stanford Microarray Database, `http://genome-www5.stanford.edu`).

While such time-course data provide more information than that obtained at a single time point, analyzing them presents several special and novel challenges. A number of papers including [32], [36], and [37] present methods of clustering profiles, time-series or otherwise, by treating them as multivariate vectors. Their methods cluster the subjects based on their expression profile vectors and thus make the inherent assumption that all of them have the same number of measurements taken, and at the same time-points. Such methods do not take into account missing data that leads to unequal numbers of observations in each subject, and/or unequally spaced time points, which might vary for each subject. The profile vector, given the parameters, is assumed to be multivariate normal with independent components in many of these papers.

An alternate approach which fits spline-based curves and uses the spline coefficients for further analysis, including clustering, can be found in [3]. In [24], the authors also use spline-fits and consider a Bayesian model-based hierarchical or agglomerative clustering. More recently, [50] provides a wavelet-based clustering of time-series gene expression data. Again, while these methods provide smooth curve-fits to the data and are an improvement over earlier methods, a more explicit acknowledgement of the time-series progression or a natural method of selecting the number of clusters is still missing in such models.

To capture this structure, and generate a clustering in which the number of clusters is flexibly chosen in a natural way from the data itself, we make use of Dirichlet process (DP) priors (see [18]) and its various extensions. We propose a "semi-parametric" setup, i.e. a parametric model for the data and a nonparametric model (Dirichlet process) for the distributions which generate its parameters. If secondary interacting variables, or covariates, are present, these models can be readily generalized to incorporate them as well. Unlike spline coefficients, this has the advantage that the parameters can be physically interpretable quantities, which can be helpful in explaining past and future trends in the expression values. Our proposed methods explicitly take into account the time-series nature of the data, incorporating both the expression value as well as the time at which this value was taken, thus capturing the structure and dynamics of data.

Efficient computational techniques make the Dirichlet process a widely used nonparametric model for random distributions in a Bayesian setting. As it assigns probability only on the space of discrete distributions, it is not used directly to model the data, but rather as a prior for a mixing distribution. We also focus on the Dirichlet process as a way to model collections of dependent distributions, rather than just as a tool for modeling exchangeable samples from a single unknown distribution. To accomplish this, we use hierarchical and nested Dirichlet process models introduced and discussed in [52] and [45].

Given the complex nature of these models, it is nearly impossible to find explicit closed form expressions of the posterior distributions (which is a common problem with complex Bayesian modeling). We thus employ Markov chain Monte Carlo (MCMC) methods to fit the models to observed data.

In Chapter 2, we provide an introduction to Dirichlet process and its clustering property. In Chapter 3, we model the individual-level trajectories as a function of time and use such models to classify subject profiles into clusters with similar trajectories. This model is very flexible in allowing one to have time-points that are not equidistant, or subject profiles with missing data values. In Chapter 4, we model the joint distribution of the expression levels of a subject over time as a multivariate vector with a certain individual mean and a correlated covariance structure. We consider two cases depending on whether all the subjects have the same covariance or have their own individual covariances. After putting appropriate priors depending on the context, we develop clustering methods for these subjects. Such dependent covariance structures extend what has been done in the literature so far. In Chapter 5.2 we use a Nested Dirichlet process, a concept introduced very recently in [45], which allows subjects with similar distributions to cluster together. Although this first-stage clustering (of subjects) is the primary objective here, this model also provides clusters of time-points that are nested within such subject clusters. In Chapter 5.3 we discuss what might be conceptually considered a special case of this, by clustering subjects based on their summary statistics, rather than on their distributions. In Chapter 6, we provide a confirmatory technique based on growth curve modeling to say how significant any of these clusters are, from a statistical testing perspective. Each of the models developed and the corresponding confirmatory tests, have been applied to a portion of the yeast time-series data due to [51] primarily for illustrative and comparative purposes. We conclude in Chapter 7 with some comparisons and comments regarding the different models introduced and by suggesting directions for possible future work.

Chapter 2

Dirichlet Process Mixtures and Clustering

2.1 Introduction

Parametric modeling and inference concerns situations where the data is assumed to come from a family of distributions whose functional form is known, but which may involve a finite number of unknown parameters. The goal then is to infer (i.e. estimate, test hypotheses etc.) about these unknown parameters, based on the data. This is how classical inference proceeds and even Bayesian inference in such a context is rather straightforward because of the finite-dimensional nature of the parameter space which allows computation of the posterior distributions either theoretically or computationally. Nonparametric or model-free inference, on the other hand, involves data that is assumed to come from a general (completely unspecified) distribution. A Bayesian analysis in such a nonparametric context involves putting a prior on a rich class of distributions.

Since a prior based on a Dirichlet process plays a prominent role in all our subsequent discussion and developments, we give a brief introduction and review of this important stochastic process in this chapter. We also outline different methodologies that one can use to obtain clusters based on the Dirichlet process mixture models.

Dirichlet process has been applied in a variety of contexts and some recent applications include Finance ([27]), Econometrics ([12]), Genetics ([36], [15]), Medicine ([28], [6]) and Auditing ([31]).

2.2 The Dirichlet Process and Some Properties

The Dirichlet Process (DP) is a very useful prior in the context of nonparametric Bayesian modeling of data, and was introduced by Ferguson ([18]). Let $\{\Omega, \mathcal{B}, P\}$ be a probability space and $\mathbf{P}$ be the space of probability measures on $\{\Omega, \mathcal{B}\}$ so that $P \in \mathbf{P}$. The DP prior is a distribution on the set of all probability distributions $\mathbf{P}$. The name Dirichlet process comes from the fact that it gives rise to Dirichlet distributed finite dimensional marginal distributions, just as the Gaussian process has Gaussian distributed finite dimensional marginals. Recall that for any $a_1, \ldots, a_{m+1} > 0$, an m-dimensional random vector $(X_1, \ldots, X_m)$ is said to have a Dirichlet distribution, denoted by $(X_1, \ldots, X_m) \sim \text{Dir}(a_1, \ldots, a_m; a_{m+1})$ if it has the pdf

$$f(x_1, \ldots, x_m) = \frac{\Gamma(a_1 + \cdots + a_{m+1})}{\Gamma(a_1) \cdots \Gamma(a_{m+1})} x_1^{a_1 - 1} \ldots x_m^{a_m - 1} (1 - x_1 - \ldots - x_m)^{a_{m+1} - 1}$$

for $\{x_i \geq 0, \sum_{i=1}^m x_i \leq 1\}$. This reduces to the Beta distribution when m=1. For a good source of reference on the Dirichlet distribution and its properties, see [59].

Given a probability distribution G_0 and a scalar parameter $M > 0$, we say that a random distribution G with induced probability measure P given by $P((-\infty, a]) = G(a)$ has a Dirichlet process prior with baseline distribution G_0 and precision parameter M, denoted by

$$G|(M, G_0) \sim \mathcal{D}(M, \ G_0)$$

if, for any finite partition $B_1, \ldots, B_{m+1}$ of the sample space over which $G(\cdot)$ is defined,

$$(G(B_1), \ldots, G(B_m)) \sim \mathrm{Dir}(MG_0(B_1), \ldots, MG_0(B_m); MG_0(B_{m+1}))$$

Blackwell and MacQueen ([8]) characterize the DP in terms of its predictive distribution as follows: if $(\theta_1, \ldots, \theta_{n-1})$ is an iid sample from G and $G|(M, G_0) \sim \mathcal{D}(M,\ G_0)$, then after integrating out the unknown G, one can show that the conditional predictive distribution of a new observation

$$\theta_n | \theta_1, \ldots, \theta_{n-1} \sim \frac{M}{M+n-1} G_0 + \sum_{i=1}^{n-1} \frac{1}{M+n-1} \delta_{\theta_i}, \tag{2.1}$$

where δ_θ denoted a probability distribution with point mass at θ. This result relating the Dirichlet process to a Polya urn scheme is the basis for many standard computational tools used to fit models based on the Dirichlet process. Another computationally useful formulation of the DP is the "stick-breaking representation" ([47, 46]). In this formulation, a distribution $G \sim \mathcal{D}(M,\ G_0)$ can be represented in the form

$$G(\cdot) = \sum_{k=1}^{\infty} \pi_k \delta_{\theta_k}(\cdot)$$

where $\theta_k \overset{iid}{\sim} G_0$, and the weights π_k are given by $\pi_k = v_k \Pi_{s=1}^{k-1}(1 - v_s)$ with $v_k \overset{iid}{\sim} Beta(1, M)$. The infinite sum in the definition of $G(\cdot)$ above, can be approximated by a large enough finite sum for practical computational purposes. This formulation goes on to demonstrate the surprising and somewhat unwelcome consequence that the DP assigns probability 1 to the subset of discrete distributions.

This "stick-breaking" formulation has been exploited to generate efficient alternative MCMC algorithms for sampling from the posterior distribution. It has also been the starting point for the definition of many generalizations of DP that allow dependence across a collection of distributions, as we do later in defining a Nested DP.

Conjugacy is one of the appealing properties of the Dirichlet process. It has been shown that [2], if $\theta_1, \ldots, \theta_n \overset{id}{\sim} G$ and $G|(M, G_0) \sim \mathcal{D}(M,\ G_0)$ then the posterior distribution of G is given by

$$G|(\theta_1, \ldots, \theta_n) \sim \mathcal{D}(M+n, \frac{M}{M+n}G_0 + \frac{1}{M+n}\sum_{i=1}^{n}\delta_{\theta_i}).$$

As a consequence, the optimal estimator of $G(\cdot)$ under squared error loss, is given by

$$\hat{G}(\cdot) = \frac{M}{M+n}G_0(\cdot) + \frac{1}{M+n}\sum_{i=1}^{n}\delta_{\theta_i}(\cdot)$$

which converges to the empirical cdf as the sample size $n \longrightarrow \infty$.

2.3 Dirichlet Process Mixtures and Clustering

In typical parametric Bayesian modeling, the observations are assumed to come from a distribution $F(\cdot)$, which is characterized by a few parameters. One puts a prior on these parameters to obtain their posterior distribution and inference is done using this posterior distribution. In nonparametric Bayesian modeling, one puts a prior on this entire distribution F, and the goal is to infer F, given the observed data.

Mixture models, which are able to accommodate several data anomalies like multimodality, heavy tails, etc, have often been used in model-based clustering of data. They are indeed a natural choice if the observed population can be thought of as a combination of several subgroups and the number of subgroups is known. See for example [55] for Bayesian analysis of models of this type. However such finite mixture models can end up with problems of "identification"; ie., the interplay between the number of components in the mixture and the parameters in the component-models are sometimes not fully identified by the data. A potentially appealing alternative to such finite-mixture models is to take continuous mixtures of the form

$$f(y|g) = \int f(y|\theta)g(\theta)d\theta \tag{2.2}$$

with an appropriate mixing distribution $g(\cdot)$. Such a $g(\cdot)$ can be a member of a parametric family, the choice of which again involves some level of arbitrariness. An even better approach is when this mixing distribution and the number of components can be suggested nonparametrically by the data itself, as is done when using a Dirichlet Process. The resulting mixture distribution is called a Dirichlet Process mixture and is described below.

The DP mixture model induces a prior on $f(\cdot)$ indirectly through a prior on the mixing distribution G. A popular choice is a mixture of Gaussians where $\theta = (\mu, \Sigma)$ $f(.|\theta) = N_p(.|\mu, \Sigma)$, the p-variate normal distribution.

Antoniak ([2]) utilized DP in the context of mixture models, by replacing Equation 2.2 with

$$f(y|G) = \int f(y|\theta)dG(\theta)$$

where $G \sim \mathcal{D}(M,\ G_0)$. This makes the family $f(.)$ a nonparametric mixture family where $G(\theta)$ plays the role of the conditional distribution of θ given G.

As always, we start with data $Y_1, \ldots, Y_n$ which are independent observations that come from unknown underlying densities

$$Y_i|\theta_i \overset{indep.}{\sim} f(y_i|\theta_i),$$

$$\theta_i|G \overset{iid}{\sim} G$$

and

$$G|(M, G_0) \sim \mathcal{D}(M,\ G_0)$$

where G serves as a DP mixture over the vector parameter θ. Data points are clustered by virtue of their sharing identical values of the parameter θ_i.

Clustering property: The Polya urn representation given in Equation 2.1 implies that with positive probability, random samples obtained from a distribution G are not all distinct, when $G \sim \mathcal{D}(M,\ G_0)$. Let $\theta_1^*,\ \ldots,\ \theta_m^*$ be the distinct values of $\theta_1,\ \ldots,\ \theta_n$ with the corresponding multiplicities being $n_1,\ \ldots,\ n_m$ respectively, where $n_1 + \cdots + n_m = n$. This induces a partitioning of the set $\{1,\ \ldots,\ n\}$ into m different groups, this can be used in natural clustering of data in the following sense. We will say that observations Y_i and Y_j cluster together if and only if $\theta_i = \theta_j$. The predictive distribution in Equation (2.1) can be thus rewritten as:

$$\theta_{n+1}|\theta_1,\ \ldots,\ \theta_n \sim \frac{M}{M+n}G_0 + \sum_{k=1}^{m} \frac{n_k}{M+n}\delta_{\theta_k^*}$$

indicating that past frequent distinct values are more likely to recur.

Remark: The parameter M in a DP $\mathcal{D}(M,\ G_0)$ is called the precision parameter. The larger the value of M, the more certain we can be that G_0 is indeed the true value of G, while small M results in a large uncertainty in the randomness of G. M also guides the amount of clustering in the θ_i's, with larger M giving rise to larger number of clusters. The choice of the precision parameter M in a DP is a matter of debate since a high value of M favors the adoption of the base measure G_0 as the posterior for the θ_i making it smoother, while small values will favor using the empirical cdf of the data as this posterior. One can get around this issue partially, by putting another prior on this parameter M.

Since the expected number of clusters is given by

$$E(m) = \sum_{i=1}^{n} \frac{M}{M+i-1},$$

one can use this as a guideline in choosing M or a prior on M.

2.4 Clustering in a Dirichlet Process

At each iteration of the Gibbs sampler, an "incidence matrix" $E = ((e_{ij}))$ is obtained, where $e_{ij} = 1$ if observations i and j cluster together, 0 otherwise. Such matrices are averaged over, say, 10,000 iterations of the Gibbs sampler to get the matrix $\overline{P} = ((p_{ij}))$, where p_{ij} can be interpreted as the estimated proportion of times observations i and j cluster together. The matrix $D = ((1 - p_{ij}))$ can be interpreted as a matrix of "distances" on which one can use standard hierarchical clustering procedures to obtain clusters of the observations.

Alternatively, one may use the "least-squares clustering" estimator proposed in [14] which goes as follows. Run the Gibbs sampler an additional 10,000 times (say,) and for each iteration, note the cluster structure generated and its corresponding incidence matrix E. Find the cluster structure that results in $||E - \overline{P}||^2 = \sum_{i,j}(e_{ij} - p_{ij})^2$ being the smallest among them. This is chosen to be the cluster structure that represents the data. This empirical approach to finding the clustering that comes closest to the already observed average, has the flavor of minimizing the posterior expected loss, assuming equal costs of clustering mistakes (see [7]) and is not very computationally demanding even with a large number of genes.

On the other hand, it is clear that the incidence matrix E_0 which minimizes $||E - P||^2$ is to take $e_{ij} = 1$ when $p_{ij} > .5$ and 0 when $p_{ij} < .5$. Since this may not result in a cluster graph (a vertex-disjoint union of cliques), however, one might ask what the fewest changes to the edge set of this input graph E_0, so that it becomes a cluster graph would be. This problem has been addressed by [48].

Chapter 3

Clustering Based on Trajectories

3.1 Introduction

In time series gene expression data, one may attempt to model individual level trajectories as a function of time and use such models to classify genes with similar trajectories into subgroups or clusters. In this chapter a Dirichlet process prior is used to model the distribution of the individual trajectories. The Dirichlet process leads to a natural clustering, and thus, sharing of information among genes with similar trajectories. A fully Bayesian approach for model fitting and prediction is implemented using MCMC procedures and as a test case, applied to a selection of time series data due to [51]. This model allows us to pool information from individual genes with similar profiles, so as to obtain improved parameter estimates.

This idea is similar in spirit to modeling longitudinal data as is done in [57], [9], [25] and [22] while providing an alternative approach to cluster identification in microarray studies as in [36, 24, 32].

3.2 Trajectory Models

In this section we discuss a basic trajectory model with Gaussian errors, and then introduce a more general and elaborate model which allows the error variances for the genes to be different, and for the errors to come from a scale mixture of Gaussians, allowing for fat-tailed error distributions like the t-distribution. We assume that the observed data set consists of information on N genes and on each gene, measurements have been recorded at p time points. Let y_{ij} denote the expression level of the ith gene measured at j^{th} time point, i.e., at time denoted by t_{ij} with $i = 1, \ldots, N$; $j = 1, \ldots, p$).

Our "change-point model" assumes the following form:

$$y_{ij} = \psi_i(t_{ij}) + e_{ij} \tag{3.1}$$

where $\psi_i(\cdot)$ is the trajectory function of the expression level for the ith gene and e_{ij} is the random error associated with the jth measurement on the ith gene. The trajectory function is assumed to have the following form:

$$\psi_i(t) = \delta_{i0} + \delta_{i1} t + \sum_{l=1}^{p} \beta_{il}(t - t_{il})_+, \tag{3.2}$$

where the notation u_+ is used to define the spline:

$$u_+ = \begin{cases} u & \text{if } u > 0, \\ 0 & \text{otherwise.} \end{cases}$$

t_{il} $(1 \leq l \leq p)$ is the l^{th} time point of measurement and is known. For gene i, δ_{i0} can be interpreted as the random intercept, δ_{i1} as the random slope and β_{il}'s as the random changes in slopes at the respective time points of measurement. To allow for the possibility that different genes (trajectories) may have different numbers and locations of changepoints, we assume that the slope changes β_{il} have a distribution with point mass at zero. This very general and novel formulation thus includes the possibility of having anywhere between 0 and p changepoints for any trajectory.

3.2.1 Basic Trajectory Model

In this model, the random errors e_{ij} are assumed to have independent zero-mean normal distributions i.e.

$$e_{ij}|(\sigma_e^2) \overset{iid}{\sim} N\left(0,\ \sigma_e^2\right), \qquad (j = 1,\ \ldots,\ p)$$

where σ_e^2 is the common error variance. In this case, if $\boldsymbol{Y}_i = (y_{i1},\ \ldots,\ y_{i,p})'$ denotes the vector of observations on the ith gene, its contribution to the likelihood is given by:

$$f(\boldsymbol{Y}_i|\psi_i(\cdot),\ \boldsymbol{\theta}_i,\ \sigma_e^2) \propto \left(\frac{1}{\sigma_e^2}\right)^{p/2} \exp\left[\frac{-1}{2\sigma_e^2}\sum_{j=1}^{p}\{y_{ij} - \psi_i(t_{ij})\}^2\right], \tag{3.3}$$

where $\boldsymbol{\theta}_i = (\delta_{i0},\ \delta_{i1},\ \beta_{i1},\ \ldots,\ \beta_{ip})'$.

3.2.2 A General Trajectory Model

In a more general setting, it is possible that the error variance σ_e^2 does not stay the same across all the genes, and a Gaussian error distribution is inadequate to model the errors. There is considerable literature to indicate that scale mixtures of Gaussians provide a very general class of error distributions in such a case (see for instance [1], [56].)

The random errors e_{ij} are then assumed to have independent zero-mean normal distributions as follows:

$$e_{ij}|(\sigma_e^2,\ \eta_i) \overset{iid}{\sim} N\left(0,\ \frac{\sigma_e^2}{\eta_i}\right), \qquad (j = 1,\ \ldots,\ p)$$

where σ_e^2 is the common error variance and η_i is the gene-specific scale factor to allow for possible heterogeneity in error variances.

Define $\boldsymbol{Y}_i = (y_{i1},\ \ldots,\ y_{i,p})'$ to be the vector of observations on the ith gene. The contribution of the observations on the ith gene to the likelihood is then:

$$f(\boldsymbol{Y}_i|\psi_i(\cdot),\ \boldsymbol{\theta}_i,\ \eta_i,\ \sigma_e^2) \propto \left(\frac{\eta_i}{\sigma_e^2}\right)^{p/2} \exp\left[\frac{-\eta_i}{2\sigma_e^2}\sum_{j=1}^{p}\{y_{ij}-\psi_i(t_{ij})\}^2\right], \qquad (3.4)$$

where $\boldsymbol{\theta}_i = (\delta_{i0},\ \delta_{i1},\ \beta_{i1},\ \ldots,\ \beta_{ip})'$.

3.3 Prior information

As above, let the random effects for the ith gene be denoted by $\boldsymbol{\theta}_i$. We assume a nonparametric prior for $\boldsymbol{\theta}_i$ as follows:

$$\boldsymbol{\theta}_i|G_\theta \overset{iid}{\sim} G_\theta$$

where

$$G_\theta|(M_\theta, G_{0\theta}) \sim \mathcal{D}(M_\theta,\ G_{0\theta}).$$

This choice of prior is made primarily to make the model robust to mis-specifications and also flexible enough to accommodate various shapes of trajectories. Other advantages are the rich class of algorithms for drawing posterior samples as well as the natural clustering property of random effects giving rise to clusters of genes with same mean trajectory. We assume, under $G_{0\theta}$,

$$\left.\begin{array}{lll} \delta_{il} & \sim & N(\delta_l,\ \sigma_{\delta l}^2) \qquad\qquad\qquad (l=0,\ 1), \\ \beta_{il} & \sim & p_l\delta_{\{0\}} + (1-p_l)N(\beta_l,\ \sigma_{\beta l}^2) \quad (l=1,\ \ldots,\ p), \end{array}\right\} \qquad (3.5)$$

where $\delta_{\{a\}}$ denotes the point mass at a. The $(p+2)$ components of equation (3.5) are assumed to be mutually independent for each i under the baseline distribution $G_{0\theta}$. This choice for the baseline distribution of the random slope-changes β_{il} allows for the possibility that with positive probability p_l, there is no change in slope in the ith trajectory at the potential changepoint t_{il} (thereby making it a non-changepoint), with these probabilities potentially differing for the different changepoints. The advantage of such a model specification is to allow for varying number of changepoints for different genes.

In the simpler model with Gaussian errors, the error variance σ_e^2 is assumed to have an

$$\sigma_e^2 \sim IG(a_\sigma,\ b_\sigma)$$

prior, whereas in the more general model, in addition, we assume the error scale factors to have the following prior distribution:

$$\eta_i | G_\eta \overset{iid}{\sim} G_\eta,$$

$$G_\eta | (M_\eta,\ G_{0\eta}) \sim \mathcal{D}(M_\eta,\ G_{0\eta})$$

and $G_{0\eta} \equiv \chi_r^2/r$. Thus, under the baseline distribution of the Dirichlet process prior, the errors are assumed to have a t distribution with r degrees of freedom. The result of this prior choice is the robustification of the error distribution allowing for "fatter tails" than a normal distribution as well as heterogeneity of error variances.

We further assume

$$\sigma_{\delta 0}^2,\ \sigma_{\delta 1}^2 \overset{iid}{\sim} IG(a_\delta,\ b_\delta),$$

$$\sigma_{\beta 1}^2,\ \ldots,\ \sigma_{\beta p}^2 \overset{iid}{\sim} IG(a_\beta,\ b_\beta)$$

and

$$p_1,\ \ldots,\ p_p \overset{iid}{\sim} \rho\delta_{\{0\}} + (1-\rho)U(0,\ 1).$$

We also assume that

$$\delta_l \sim N(d_l,\ \sigma_{dl}^2), \qquad (l = 0,\ 1)$$

and

$$\beta_l \sim N(b_l,\ \sigma_{bl}^2), \qquad (l = 1,\ \ldots,\ p).$$

Finally, we assume

$$M_\theta \sim G(a_\theta,\ b_\theta),$$

$$M_\eta \sim G(a_\eta,\ b_\eta),$$

and $r \sim DU(1,\ H)$, where "$DU(a,\ b)$" is the discrete uniform distribution on the set $\{a,\ a{+}1,\ \ldots,\ b\}$. The hyperparameters $a_\delta,\ b_\delta,\ a_\beta,\ b_\beta,\ \rho,\ d_l,\ \sigma^2_{dl},\ b_l,\ \sigma^2_{bl},\ a_\theta,\ b_\theta,\ a_\eta,\ b_\eta$ and H are assumed to be known. In practice, lacking additional information, these hyperparameters will be chosen so that the priors are as non-informative as possible, while maintaining propriety of the corresponding priors.

3.4 Computational Details

Due to lack of a closed form expression, the exact posterior distribution of the model parameters is intractable. However, the availability of simple univariate conditionals allows easy sampling from the posterior through the Gibbs sampler. The expressions of the univariate conditionals of the model parameters are given in Section 3.4.1.

Due to the non-conjugate baseline prior specification, the random effect vectors $\boldsymbol{\theta}_i$ are updated using Algorithm 8 of [39] as follows: At any step of the iteration of the Gibbs sampler, let $\boldsymbol{\theta}_1,\ \ldots,\ \boldsymbol{\theta}_N$ be the current values of the changepoint vectors associated with the N individuals. Due to clustering property of DP [2], there will be k_θ $(1 \le k_\theta \le N)$ distinct random effects. Let them be denoted by $\boldsymbol{\theta}^*_1,\ \ldots,\ \boldsymbol{\theta}^*_{k_\theta}$ with the jth distinct value being $\boldsymbol{\theta}^*_j = (\delta^*_{j0},\ \delta^*_{j1},\ \beta^*_{j1},\ \ldots,\ \beta^*_{jp})$. Let $\boldsymbol{c}^\theta = (c^\theta_1,\ \ldots,\ c^\theta_N)$ be the associated "configuration vector" indicating cluster membership, where $c^\theta_i = s$ if and only if $\boldsymbol{\theta}_i = \boldsymbol{\theta}^*_s$. The configuration vector can be used to identify the cluster structure of the changepoints, and, when combined with the distinct values, can recreate the values associated with each subject. Algorithm 8 proceeds in two steps – first it updates the configuration vector and then it updates the distinct values arising from the resulting cluster structure. The configuration vector is updated by introducing temporary auxiliary variables, corresponding to components that are not associated with any of the observations, such that the marginal distribution remains invariant. Each distinct value $\boldsymbol{\theta}^*_c$ is updated component-wise, using the Adaptive Rejection Sampling (ARS) method of [23], since the corresponding densities are log-concave. Since each β^*_{jl} has a 2-component mixture distribution, we use an associated variable ω^*_{jl} indicating the component membership.

Updating of the scale factors η_i proceeds in a similar fashion, except that due to conjugacy, we use Algorithm 2 of [39] to update the cluster structure and the distinct values. Updating details of all parameters and hyperparameters of the model are given in Section 3.4.1. All code for fitting this model was written in `C` and can be found in Appendix A.

3.4.1 Posterior conditionals

For the posterior updates here as well throughout this work, we need the following proposition which can be checked rather easily using Bayes Theorem.

Proposition 1 *(i) If*

$$X|\mu \sim N\left(\mu, \sigma^2\right)$$

and

$$\mu \sim N\left(\mu_0, \tau^2\right),$$

then

$$\mu|X = x \ \sim N\left(\frac{\frac{x}{\sigma^2} + \frac{\mu_0}{\tau^2}}{\frac{1}{\sigma^2} + \frac{1}{\tau^2}}, \frac{1}{\frac{1}{\sigma^2} + \frac{1}{\tau^2}}\right).$$

(ii) If

$$\left(\frac{V}{\sigma^2}\right)|\sigma^2 \sim \chi^2_\nu$$

and

$$\sigma^2 \sim IG(a, b),$$

then

$$\sigma^2|V = v \ \sim IG(a^*, b^*),$$

where $a^* = a + \frac{\nu}{2}$ *and* $b^* = \left(\frac{1}{b} + \frac{\nu}{2}\right)^{-1}$.

■

Let $\boldsymbol{\theta}_j^* = (\delta_{j0}^*,\ \delta_{j1}^*,\ \beta_{j1}^*,\ \ldots,\ \beta_{jp}^*)'$ be the jth distinct value of the random effect vector $(j = 1,\ \ldots,\ k_\theta)$ and η_j^* be the jth distinct value of the scale factor η, $(j = 1,\ \ldots,\ k_\eta)$. The configuration structure $c^\theta = (c_1^\theta,\ \ldots,\ c_N^\theta)$ of the random effects is updated using [39]'s Algorithm 8. Similarly, the configuration structure $d^\eta = (d_1^\eta,\ \ldots,\ d_N^\eta)$ of the η values is updated using Algorithm 2 of [39].

1. Sample M_θ in 2 steps [17]:

   (a) Sample latent variable $\phi_\theta | (k_\theta,\ M_\theta) \sim B(M_\theta + 1,\ N)$

   (b) Sample $M_\theta | (\phi_\theta,\ k_\theta) \sim \pi_\theta G(a_\theta + k_\theta, (b_\theta^{-1} - \log \phi_\theta)^{-1}) + (1 - \pi_\theta) G(a_\theta + k_\theta - 1,\ (b_\theta^{-1} - \log \phi_\theta)^{-1})$

2. Similarly, sample M_η in 2 steps.

3. Sample $\sigma_e^2 | \cdots \sim IG\left(a_\sigma + \frac{Np}{2},\ \left[\frac{1}{b_\sigma} + \frac{1}{2}\sum_{i=1}^N \eta_i \sum_{j=1}^p (y_{ij} - \psi_i(t_{ij}))^2\right]^{-1}\right)$

4. Sample $\sigma_{\delta l}^2 | \cdots \sim IG\left(a_\delta + \frac{k_\theta}{2},\ \left\{b_\delta^{-1} + \frac{1}{2}\sum_{j=1}^{k_\theta} (\delta_{jl}^* - \delta_l)^2\right\}^{-1}\right)$

5. Sample $\sigma_{\beta l}^2 | \cdots \sim IG\left(a_\beta + \frac{1}{2}\sum_{j=1}^{k_\theta}(1 - \omega_{jl}^*),\ \left\{b_\beta^{-1} + \frac{1}{2}\sum_{j=1}^{k_\theta}(1 - \omega_{jl}^*)(\beta_{jl}^* - \beta_l)^2\right\}^{-1}\right)$

6. Sample $\omega_{jl}^* | \cdots \sim \begin{cases} \delta_{\{0\}} \text{ if } \beta_{jl}^* \neq 0 \\ Bernoulli\left(\dfrac{p_l}{p_l + \frac{(1-p_l)}{\sqrt{2\pi\sigma_{\beta l}^2}} \exp(\frac{-\beta_l^2}{2\sigma_{\beta l}^2})}\right) & \text{otherwise} \end{cases}$

7. Sample $p_l | \cdots \sim \begin{cases} \delta_{\{0\}} & \text{if } u_l = 1 \\ B(\sum_{j=1}^{k_\theta} \omega_{jl}^* + 1,\ k_\theta - \sum_{j=1}^{k_\theta} \omega_{jl}^* + 1) & \text{otherwise} \end{cases}$

8. Sample $u_l | \cdots \sim \begin{cases} \delta_{\{0\}} & \text{if } p_l \neq 0 \\ Bernoulli(\rho) & \text{otherwise} \end{cases}$

9. Sample $\delta_l | \cdots \sim N\left(\frac{\frac{\sum_{j=1}^{k_\theta} \delta^*_{jl}}{\sigma^2_{\delta l}} + \frac{d_l}{\sigma^2_{dl}}}{\frac{k_\theta}{\sigma^2_{\delta l}} + \frac{1}{\sigma^2_{dl}}}, \left(\frac{k_\theta}{\sigma^2_{\delta l}} + \frac{1}{\sigma^2_{dl}} \right)^{-1} \right)$

10. Sample $\beta_l | \cdots \sim N\left(\frac{\frac{\sum_{j=1}^{k_\theta} \beta^*_{jl}(1-\omega^*_{jl})}{\sigma^2_{\beta l}} + \frac{b_l}{\sigma^2_{bl}}}{\frac{\sum_{j=1}^{k_\theta}(1-\omega^*_{jl})}{\sigma^2_{\beta l}} + \frac{1}{\sigma^2_{bl}}}, \left(\frac{\sum_{j=1}^{k_\theta}(1-\omega^*_{jl})}{\sigma^2_{\beta l}} + \frac{1}{\sigma^2_{bl}} \right)^{-1} \right)$

11. Sample $\eta^*_d | \cdots \sim G\left(\frac{r}{2} + \frac{1}{2}\sum_{i:d_i=d} p, \left[\frac{r}{2} + \frac{1}{2\sigma^2_e} \sum_{i:d_i=d} \sum_{j=1}^{p} \{y_{ij} - \psi_i(t_{ij})\}^2 \right]^{-1} \right)$

12. Sample r from $\log f(r|\cdots) = \frac{rk_\eta}{2}(\log r - \log 2) + \frac{r}{2}(\sum_{j=1}^{k_\eta} \log \eta^*_j - \sum_{j=1}^{k_\eta} \eta^*_j) - k_\eta \log \Gamma(\frac{r}{2}), r \in \{1,\ 2,\ \ldots,\ H\}$

13. Sample δ^*_{ml} from $\log f(\delta^*_{ml}|\cdots) = \text{const.} - \frac{1}{2\sigma^2_e} \sum_{i:c^\theta_i=m} \eta_i \sum_{j=1}^{p} \{y_{ij} - \psi_i(t_{ij})\}^2 - \frac{(\delta^*_{ml} - \delta_l)^2}{2\sigma^2_{\delta l}}$

14. If $\omega^*_{ml} = 1$, set $\beta^*_{ml} = 0$, otherwise draw β^*_{ml} from $\log f(\beta^*_{ml}|\cdots) = \text{const.} - \frac{1}{2\sigma^2_e} \sum_{i:c^\theta_i=m} \eta_i \sum_{j=1}^{p} \{y_{ij} - \psi_i(t_{ij})\}^2 - \frac{(\beta^*_{ml} - \beta_l)^2}{2\sigma^2_{\beta l}}$

3.5 Data Analysis

3.5.1 Synthetic Data

First, the basic and general models were compared using a synthetic data set. This data consists of $N = 60$ genes each measured at $p = 10$ timepoints, generated from 3 different classes (clusters) of size 20 each. The class means follow slightly different trajectories (the trajectory parameters are given in Table 3.1), and the class variances are all different: 0.25, 1, and 0.04 respectively. The clustering results, namely the heatmap and dendrogram, for the basic and general trajectory models on this data are shown in Figures 3-1 and 3-2. We fully recover the original clustering with the general model 3-2, but the high variance cluster is fragmented into smaller groups (see 3-1) when one uses the basic model (which assumes equal variances) for such data. This demonstrates the ability of the more general model to take into account inherent variance differences across genes, resulting in superior clustering of the data.

	Class 1	Class 2	Class 3
δ_{i0}	0	0	-1.0
δ_{i1}	0.5	0.5	0.2
β_{i1}	0	0	0
β_{i2}	0	0	0
β_{i3}	0	0	0
β_{i4}	0	0	0
β_{i5}	0	-1.5	0
β_{i6}	0	0	0
β_{i7}	0	0	0
β_{i8}	0	0	0

Table 3.1: Synthetic data trajectory parameters.

Next, in order to assess how well our general model clusters and recovers parameters from data with this basic structure in the presence of noise, we examined a few variations on the synthetic data set. In particular, we chose to examine 1) the effect of sample size, and 2) the effect of varying noise. One could also look at the effect of varying the distance between trajectories of the different classes, but this is related to varying the noise, as it is the combination, i.e., the distance between trajectories with respect to noise (Mahalanobis distance of sorts), which is the relevant quantity. We continue to use 3 classes, with trajectory parameters as given in Table 3.1, which divide our data set into 3 equal clusters. However, we use 2 different sample sizes, of $N = 60$ and 120 (small sample size/large sample size), and 2 different class variances, of $(0.25, 1, 0.04)$ and $(1, 4, 0.16)$ (small variance/large variance) respectively, resulting in 4 variations in all.

We note that in the low variance cases, the clustering coincides with the true class assignments (similar to 3-2) regardless of the sample size, but in the high variance cases we see from $2 - 6\%$ misclassification error. The results of parameter estimation for these data sets are seen in Figures 3-3, 3-4, and 3-5 for the 3 classes respectively. The parameter estimates shown in these figures are calculated using the clusters we obtained from our model. In order to quantify the overall parameter estimation accuracy, we calculate the mean squared error of our estimates to their corresponding

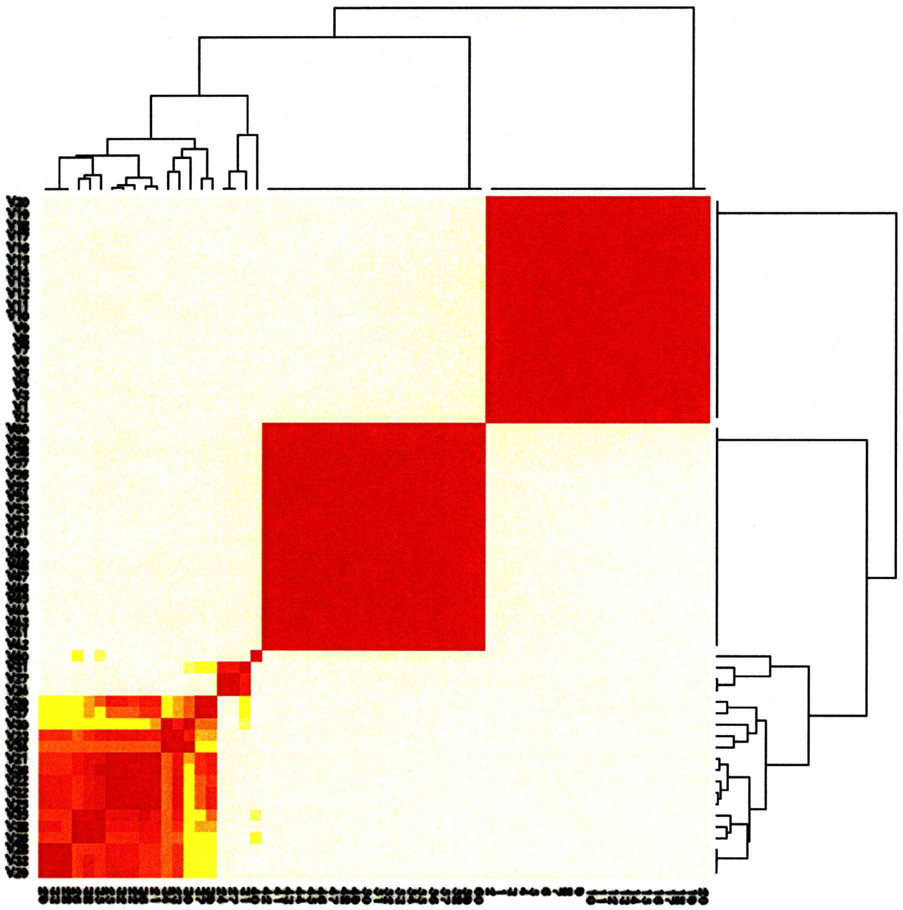

Figure 3-1: Heatmap of clusters of the synthetic data based on the average incidence matrix from the basic trajectory model. Higher intensity of red denotes more likelihood of clustering together.

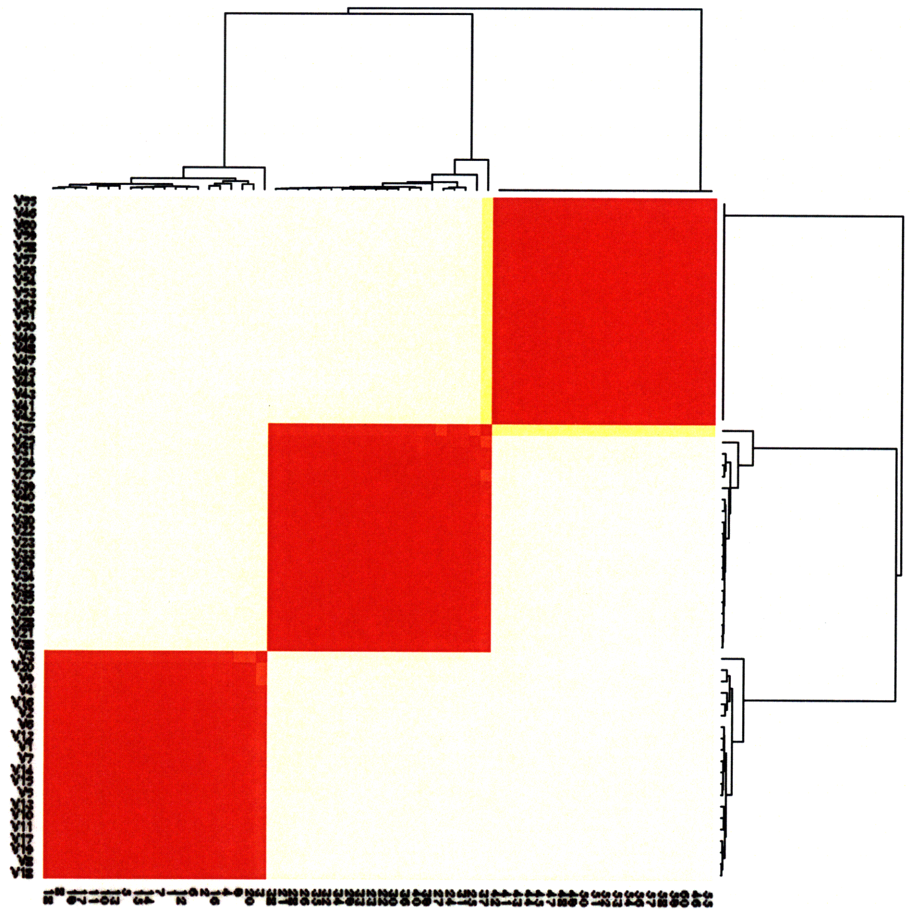

Figure 3-2: Heatmap of clusters of the synthetic data based on the average incidence matrix from the general trajectory model. Higher intensity of red denotes more likelihood of clustering together.

true values, averaging across all genes in each data set. The results of this can be seen in Table 3.2 and are generally in agreement with the expectation that this error should decrease when either the sample size is increased or when the variances are smaller.

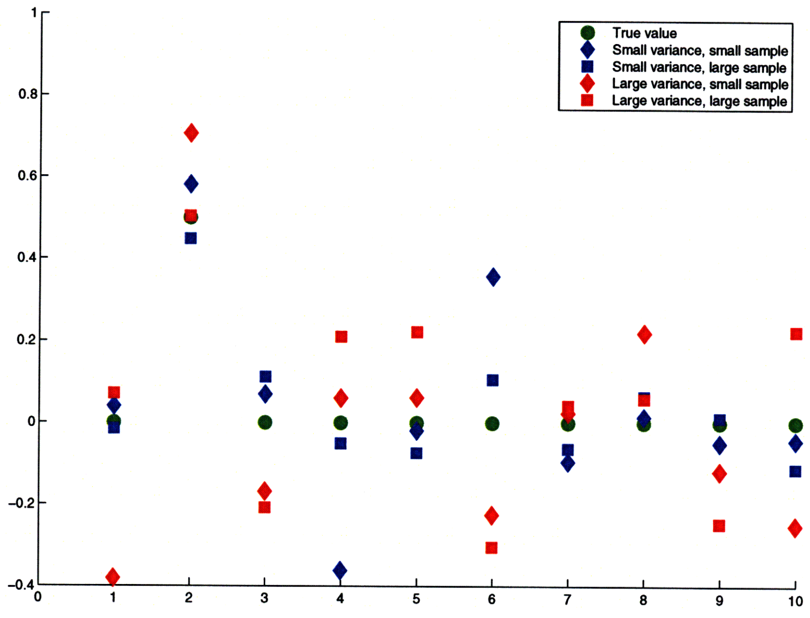

Figure 3-3: Estimated parameters for class 1. Along the x-axis, in order, are $[\delta_{i0}, \delta_{i1}, \beta_{i1}, \cdots, \beta_{i8}]$.

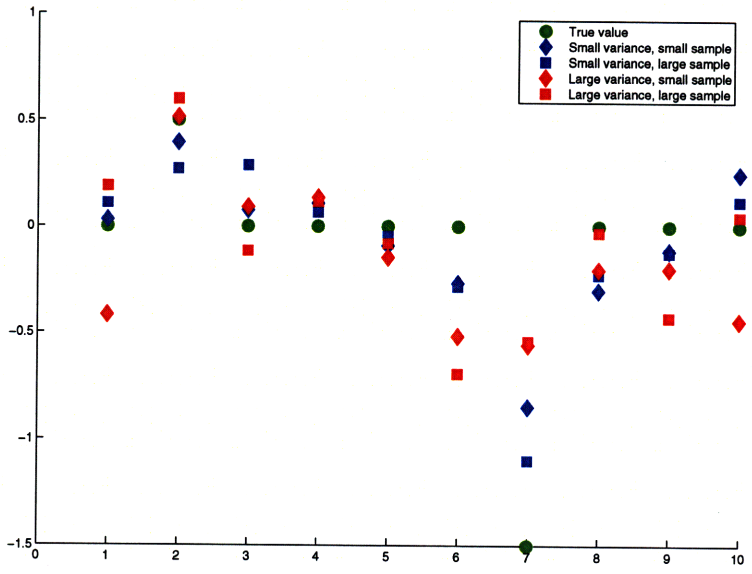

Figure 3-4: Estimated parameters for class 2. Along the x-axis, in order, are $[\delta_{i0}, \delta_{i1}, \beta_{i1}, \cdots, \beta_{i8}]$.

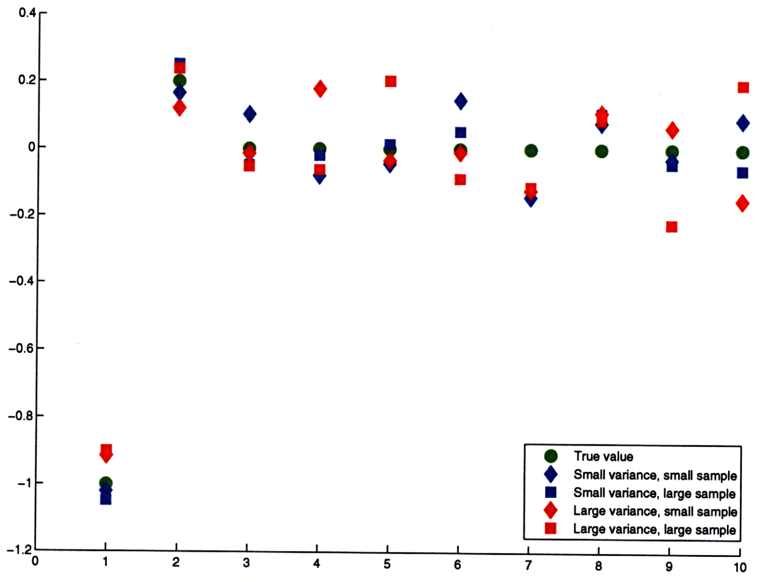

Figure 3-5: Estimated parameters for class 3. Along the x-axis, in order, are $[\delta_{i0}, \delta_{i1}, \beta_{i1}, \cdots, \beta_{i8}]$.

	1	2	3	4
δ_{i0}	0.140	0.119	0.404	0.307
δ_{i1}	0.158	0.175	0.245	0.201
β_{i1}	0.220	0.252	0.304	0.302
β_{i2}	0.301	0.196	0.343	0.293
β_{i3}	0.238	0.181	0.343	0.320
β_{i4}	0.356	0.245	0.467	0.496
β_{i5}	0.451	0.304	0.654	0.660
β_{i6}	0.282	0.229	0.362	0.266
β_{i7}	0.220	0.194	0.339	0.396
β_{i8}	0.289	0.214	0.445	0.335

Table 3.2: Mean squared error for trajectory parameter estimates on synthetic data. The columns represent 1) small variance, small sample size, 2) small variance, large sample size, 3) large variance, small sample size, and 4) large variance, large sample size.

3.5.2 Real Data

The general trajectory model was applied to a standard subset of the well-known data on time series of gene expression of yeast cells due to [51]. We consider $N = 798$ genes measured at $p = 18$ timepoints. After burning the Gibbs sampler for 100,000 iterations, we used posterior samples from a further 20,000 iterations to draw inference. Convergence of the sampler was ascertained using various diagnostics such as traceplots and the Gelman-Rubin statistic. The posterior median of the number of clusters was found to be 69 and a 95% credible interval is (66, 72).

Each iteration l of the Gibbs sampler produced a configuration $c = (c_1, \ldots, c_N)$ of the cluster-structure of the genes. This in turn gave rise to an incidence matrix $Q^{(l)}_{N\times N}$, where $Q^{(l)} = ((q_{i,j}))$ and $q_{i,j} = 1$ if $c_i = c_j$, 0 otherwise. The average incidence matrix $\overline{Q} = \frac{1}{L}\sum_{l=1}^{20,000} Q^{(l)}$ was then computed. An additional $20,000$ iterations were run and the iteration whose incidence matrix was "closest" to the average incidence matrix was noted. The resulting cluster structure was taken to be the cluster structure arising from the data. The method described above is as discussed in Section 2.4 and is due to [14]. The results are provided below.

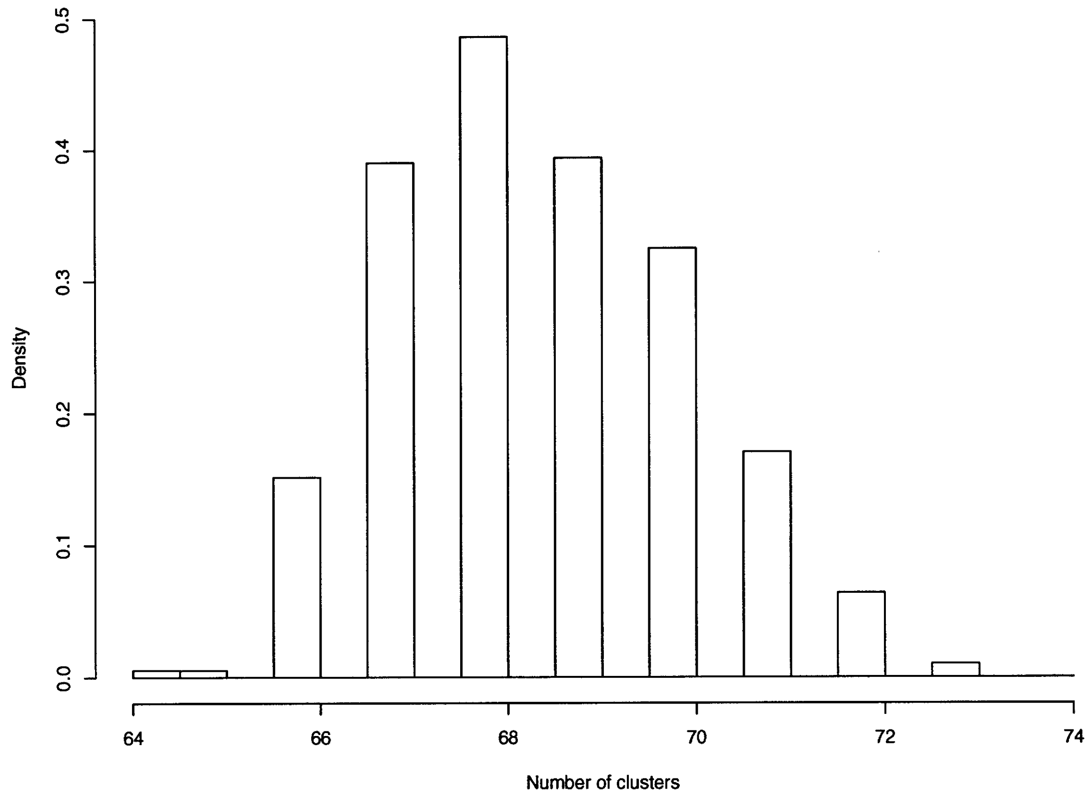

Figure 3-6: Histogram of the number of clusters generated using the trajectory model.

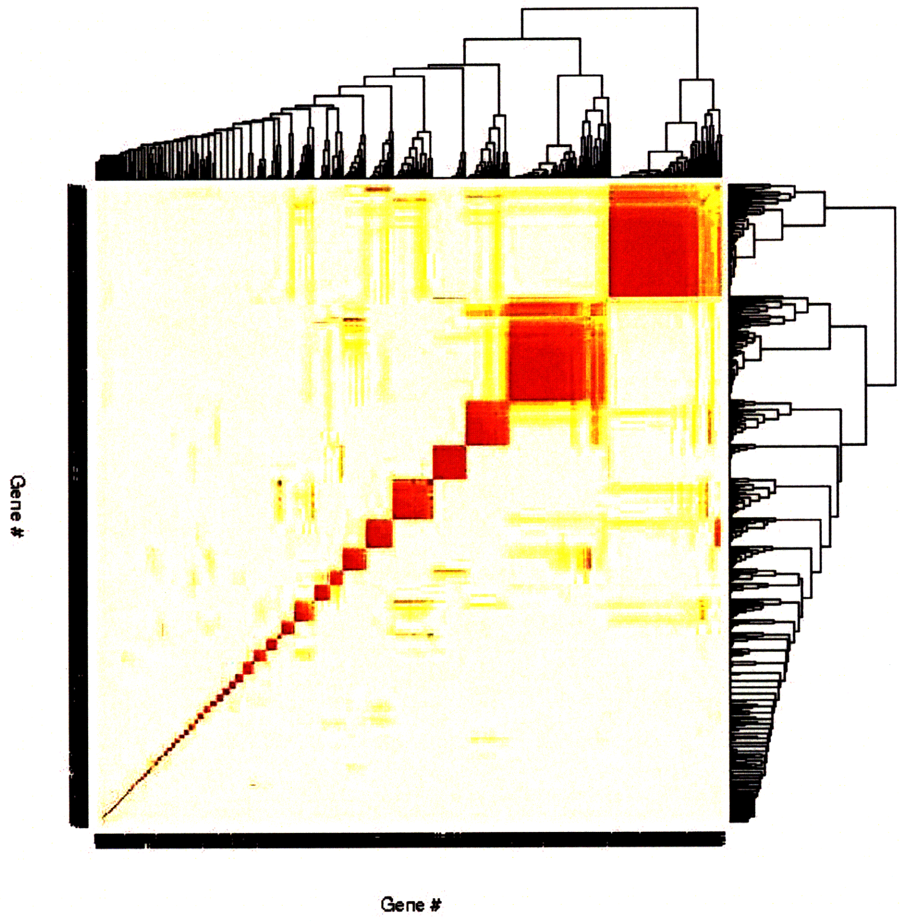

Figure 3-7: Heatmap of clusters of the 798 genes based on the average incidence matrix from the trajectory model. Higher intensity of red denotes more likelihood of clustering together.

In order to compare the various Models that we are introducing and the clusterings produced by them, we look at the top 5 clusters (for definiteness) in the hierarchical clustering, scheme corresponding to the heatmap and the resulting dendrogram. A plot of the mean expression profiles for these top 5 clusters produced by our trajectory model is given in Figure 3-8.

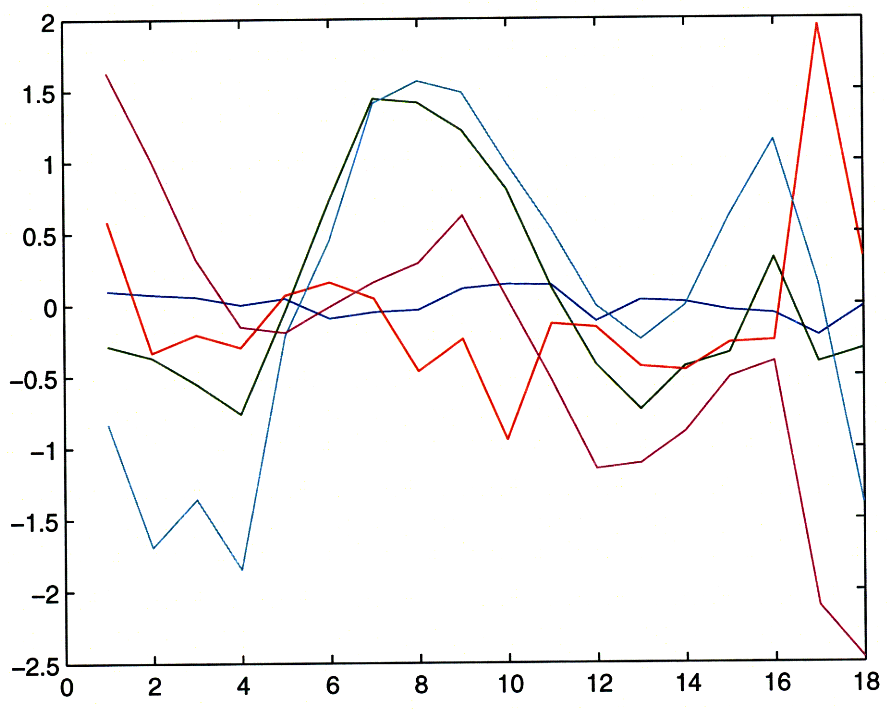

Figure 3-8: Mean curves of top 5 clusters found by the trajectory model.

To get a rough measure of cluster separation, we also consider the distance achieved by each the models from the overall mean of the data $\bar{x}$, as measured by the sum across the 5 clusters and the 18 timepoints,

$$\sum_{i=1}^{5}\sum_{t=1}^{18}(x_{it} - \bar{x}_t)^2$$

which in this case gives us a value of 54.85 (to be compared later with similar separation values for other models).

Chapter 4

Clustering Based on Expression Profiles

4.1 Introduction

In this chapter, instead of looking at the changes from timepoint to timepoint that map the trajectory of the expression curve, we look at the expression profile as a whole, i.e. as a vector of values. In contrast to earlier literature that model profiles (see e.g. [32], [36], and [37]) where the profile vector, given the parameters, is assumed to be multivariate normal with independent components, we allow for potentially rich correlation structures between time points. Such dependence between expression values for the same gene, is quite natural in our opinion.

4.2 Model description

Suppose that a microarray experiment consists of N genes. Each gene is repeatedly measured over time on p occasions, resulting in a p-dimensional measurement vector $\boldsymbol{y}_i = (y_{i1}, \ldots, y_{ip})'$ for gene i ($i = 1, \ldots, N$). Let $\boldsymbol{\mu}_i$ be the mean vector of $\boldsymbol{y}_i$ and $\boldsymbol{\Sigma}_i$ be the corresponding covariance matrix. In what follows, we present two different models for the joint distribution of the profiles. The models differ in how the covariance matrix $\boldsymbol{\Sigma}_i$ is represented.

4.2.1 Heteroscedastic model

We assume that the variance-covariance matrix for gene i is given by $\sigma_i^2 H(\rho_i)$, where $H(\rho)$ is the correlation matrix. Thus, the measurements at various timepoints on gene i have a constant variance σ_i^2 and the correlation matrix is a function of ρ_i, so that correlation between measurements at any two timepoints has the same (known) functional form that possibly varies from one gene to another. Some possible forms of the $H(\cdot)$ matrix might be

$$H(\rho) = ((\rho^{|i-j|}))_{p\times p}$$

or

$$H(\rho) = ((I_{\{i=j\}} + \rho I_{\{i\neq j\}}))_{p\times p}.$$

The first form assumes that the correlation between measurements at any two timepoints decays exponentially as a function of the distance between the two time points, so points further away are less correlated with one another. The second form assumes that all points of time have the same correlation between one another.

We assume that conditional on the mean vector, common variance and one-step correlation, two gene expression profiles are independent multivariate normal random vectors. That is,

$$\boldsymbol{y}_i|(\boldsymbol{\mu}_i, \sigma_i^2,\ \rho_i) \overset{indep}{\sim} N_p(\boldsymbol{\mu}_i,\ \sigma_i^2 H(\rho_i)), \qquad i = 1,\ \ldots,\ N$$

Let $\boldsymbol{\theta}_i = (\boldsymbol{\mu}_i,\ \sigma_i^2,\ \rho_i)$ be the vector of parameters used to describe the joint distribution of the expression profile of ith gene. We will say two genes i and j cluster if their corresponding parameter vectors $\boldsymbol{\theta}_i$ and $\boldsymbol{\theta}_j$ are identical. To this end, we assume that $\boldsymbol{\theta}_i|G \overset{iid}{\sim} G$ and $G|(M, G_0) \sim \mathcal{D}(M,\ G_0)$. As usual, here $\mathcal{D}(M,\ G_0)$ denotes a Dirichlet Process with base measure G_0 and concentration M. We assume that under the baseline prior, $(\boldsymbol{\mu}_i,\ \sigma_i^2,\ \rho_i)$ are independent with multivariate normal, inverted-gamma and beta distributions respectively. In other words, $G_0 = N_p(\boldsymbol{\mu}_0,\ \Sigma_0) \times IG(a_\sigma,\ b_\sigma) \times B(a_\rho,\ b_\rho)$, where $\Sigma_0 = diag(\sigma_{01}^2,\ \ldots,\ \sigma_{0p}^2)$ with $\sigma_{01}^2,\ \ldots,\ \sigma_{0p}^2 \overset{iid}{\sim} IG(a_0,\ b_0)$. In addition, we assume $M \sim Gamma(a_M,\ b_M)$. All the hyperparameters are assumed known and in practice, are chosen to make the priors as noninformative as possible. In particular, we choose $\boldsymbol{\mu}_0 = (0,\ \ldots,\ 0)'$. Due to lack of closed form expression for the posterior distribution and easy availability of univariate conditionals, we will use a Markov chain Monte Carlo approach to estimate the model parameters. The natural clustering property of the Dirichlet Process will lead to clustering of the values of $\boldsymbol{\theta}_i$. To summarize,

$$y_i|(\mu_i, \sigma_i^2, \rho_i) \sim^{indep} N_p(\mu_i, \sigma_i^2 H(\rho_i)), \quad i = 1, \cdots, N,$$

where $N = \#$ of genes,

$$H(\rho) = ((e^{|i-j|}))_{p \times p}$$

or

$$H(\rho) = ((1_{i=j} + \rho 1_{i=j}))_{p \times p}$$

is the correlation matrix, and $p = \#$ of observations per gene.
Let $\theta_i = (\mu_i, \sigma_i^2, \rho_i)$.
Assume $\theta_i|G \overset{iid}{\sim} G$

$$G|M, G_0 \sim DP(M, G_0)$$

$$G_0 = N_p(\mu_0, \Sigma_0) \times IG(a_\sigma, b_\sigma) \times Beta(a_\rho, b_\rho)$$

$$\Sigma_0 = diag(\sigma_{01}^2, \cdots, \sigma_{0p}^2)$$

$$\sigma_{01}^2, \cdots, \sigma_{0p}^2 \overset{iid}{\sim} IG(a_0, b_0)$$

$$\mu_0 = (0, \cdots, 0)^T$$

$$M \sim Gamma(a_M, b_M).$$

Posterior Sampling We can update the model parameters according the following scheme:

1. $$\sigma_{0j}^2|rest \sim IG\left(a_0 + \frac{N}{2}, \left[\frac{1}{b_0} + \frac{1}{2}\sum_{i=1}^{N}(\mu_j - \mu_{0j})^2\right]^{-1}\right),$$

   $j = 1, \cdots, p$

2. Let there be k distinct values, $\theta_1^*, \cdots, \theta_k^*$. Let $\boldsymbol{c} = (c_1, \cdots, c_N)$ be the configuration vector where $c_i = j$ if and only if $\theta_i = \theta_j^*$, $j = 1, \cdots, k$. Let n_j be the number of occurrences of θ_j^*. Update the configuration vector using algorithms of Neal (2000).

3. Update the distinct values θ_j^* as follows $(j = 1, \cdots, k)$

$$f(\mu_j^*, \sigma_j^{*^2}, \rho_j^*|rest) \propto \prod_{i:c_i=j} f(y_i|\mu_j^*, \sigma_j^{*^2}, \rho_j^*) f(\mu_j^*|\mu_0, \Sigma_0) f(\sigma_j^{*^2}|a_\sigma, b_\sigma) f(\rho_j^*|a_\rho, b_\rho)$$

   It is easy to show that

   (a) $\mu_j^*|rest \sim N(\eta_j^*, \Gamma_j^*)$ where

$$\Gamma_j^* = \left[\left(\frac{\sigma_j^{*^2}}{n_j} H(\rho_j^*)\right)^{-1} + \Sigma_0^{-1}\right]^{-1}$$

   and

$$\eta_j^* = \Gamma_j^* \left[\left(\frac{\sigma_j^{*^2}}{n_j} H(\rho_j^*)\right)^{-1} \left(\frac{1}{n_j}\sum_{i:c_i=j} y_i\right) + \Sigma_0^{-1}\mu_0\right]$$

(b)

$$\sigma_j^{*2}|rest \sim IG\left(a_\sigma + \frac{pn_j}{2}, \left[\frac{1}{b_\sigma} + \frac{1}{2}\sum_{i:c_i=j}(y_i - \mu_j^*)^T H(\rho_j^*)^{-1}(y_i - \mu_j^*)\right]^{-1}\right),$$

(c)

$$f(\rho_j^*|rest) \propto \frac{1}{|H(\rho_j^*)|^{\frac{n_j}{2}}} e^{-\left[\frac{1}{2\sigma_j^{*2}}\sum_{i:c_i=j}(y_i-\mu_j^*)^T H(\rho_j^*)^{-1}(y_i-\mu_j^*)\right]} (\rho_j^*)^{a_\rho-1}(1-\rho_j^*)^{b_\rho-1} \tag{4.1}$$

The components of θ_j^* can be updated one at a time using the above results. Updating of ρ_j^* using (4.1) can be done using Adaptive Rejection Sampling (ARS) or Adaptive Rejection Metropolis Sampling (ARMS).

4. Update M using a mixture of Gammas as outlined in [17].

4.2.2 Homoscedastic model

In this case, we assume the variance of measurements is same across all genes and across all timepoints. Here, the clustering of genes is achieved based on the mean vector and the correlation. Thus, we have

$$\boldsymbol{y}_i|(\boldsymbol{\mu}_i, \sigma^2, \rho_i) \overset{indep}{\sim} N_p(\boldsymbol{\mu}_i, \sigma^2 H(\rho_i)), \quad i = 1, \cdots, N,$$

where $N = \#$ of genes,

$$H(\rho) = ((\rho^{|i-j|}))_{p\times p}$$

is the correlation matrix, and $p = \#$ of observations per gene.

$$(\boldsymbol{\mu}_i, \rho_i) \equiv \theta_i \overset{iid}{\sim} G$$

$$G \sim DP(M, G_0)$$

$$G_0 = N_p(\boldsymbol{\mu}_0, \Sigma_0) \times Beta(a_\rho, b_\rho)$$

$$\Sigma_0 = diag(\sigma_1^2, \cdots, \sigma_p^2)_{p \times p}$$

$$\sigma_1^2, \cdots, \sigma_p^2 \overset{iid}{\sim} IG(a^*, b^*)$$

$$\boldsymbol{\mu}_0 = (0, \ \cdots, \ 0)^T$$

$$M \sim Gamma(a_M, b_M)$$

$$\sigma^2 \sim IG(a_\sigma, b_\sigma)$$

Posterior Sampling

1.
$$\sigma^2 | rest \sim IG\left(a_\sigma + \frac{np}{2}, \left[\frac{1}{b_\sigma} + \frac{1}{2}\sum_{i=1}^{N}(y_i - \mu_i)H(\rho_i)^{-1}(y_i - \mu_i)\right]^{-1}\right)$$

2.
$$\sigma_j^2 | rest \sim IG\left(a^* + \frac{N}{2}, \left[\frac{1}{b^*} + \frac{1}{2}\sum_{i=1}^{N}(\mu_{ij} - \mu_{0j})^2\right]^{-1}\right),$$

$j = 1, \cdots, p$

3. Let there be k distinct values, $\theta_1^*, \cdots, \theta_k^*$. Let $\boldsymbol{c} = (c_1, \cdots, c_N)$ be the configuration vector where $c_i = j$ if and only if $\theta_i = \theta_j^*$, $j = 1, \cdots, k$. Let n_j be the number of occurrences of θ_j^*. Update the configuration vector using algorithms of Neal (2000).

4. Update the distinct values as follows:

$$\boldsymbol{\mu}_j^* | rest \sim N_p(\boldsymbol{\mu}_j^*, \Gamma_j^*),$$

where

$$\Gamma_j^* = \left[\left(\frac{\sigma^2}{n_j} H(\rho_j) \right)^{-1} + \Sigma_0^{-1} \right]^{-1}$$

and

$$\mu_j^* = \Gamma_j^* \left[\left(\frac{\sigma^2}{n_j} H(\rho_j) \right)^{-1} \left(\frac{1}{n_j} \sum_{i:c_i=j} y_i \right) + \Sigma_0^{-1} \mu_0 \right]$$

$$f(\rho_j^* | rest) \propto \frac{1}{|H(\rho_j^*)|^{\frac{n_j}{2}}} e^{-\left[\frac{1}{2\sigma^2} \sum_{i:c_i=j} (y_i - \mu_j^*)^T H(\rho_j^*)^{-1} (y_i - \mu_j^*)\right]} (\rho_j^*)^{a_\rho - 1} (1 - \rho_j^*)^{b_\rho - 1}$$

Use Adaptive Rejection Sampling (ARS) or Adaptive Rejection Metropolis Sampling (ARMS).

5. Update M using a mixture of Gammas, along the lines of [17].

4.3 Data Analysis

4.3.1 Synthetic Data

The heteroscedastic and homoscedastic models were tested using a synthetic data set. This data consists of $N = 60$ genes at $p = 10$ timepoints, generated from 3 different classes, with differing means and variances as in Chapter 3. However, they also each have a correlation of $\rho = 0.3$ between time-points. The results on this data are shown in Figures 4-1 and 4-2. We note that these models generally seem to perform very similarly on test data, except in extreme cases.

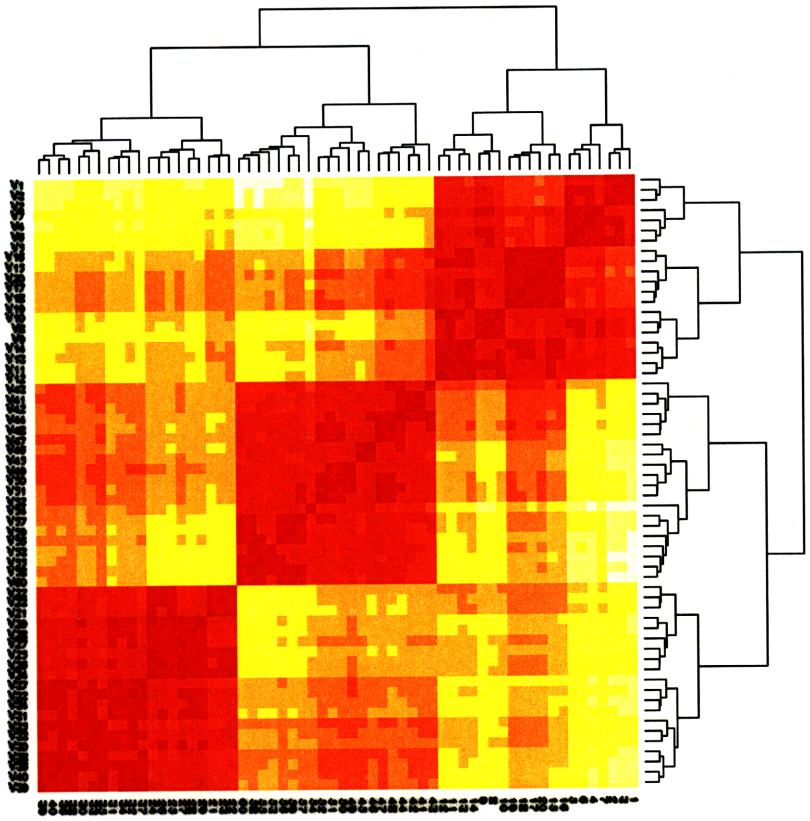

Figure 4-1: Heatmap of clusters of the synthetic data based on the average incidence matrix from the heteroscedastic model. Higher intensity of red denotes more likelihood of clustering together.

4.3.2 Real Data

The proposed models were applied to a standard subset of the well-known data on time series of gene expression of yeast cells due to [51]. We consider $N = 798$ genes measured at $p = 18$ timepoints. A burn-in of 10,000 iterations stabilized the sampler. After that, an additional 5000 iterations were used to estimate the model param-

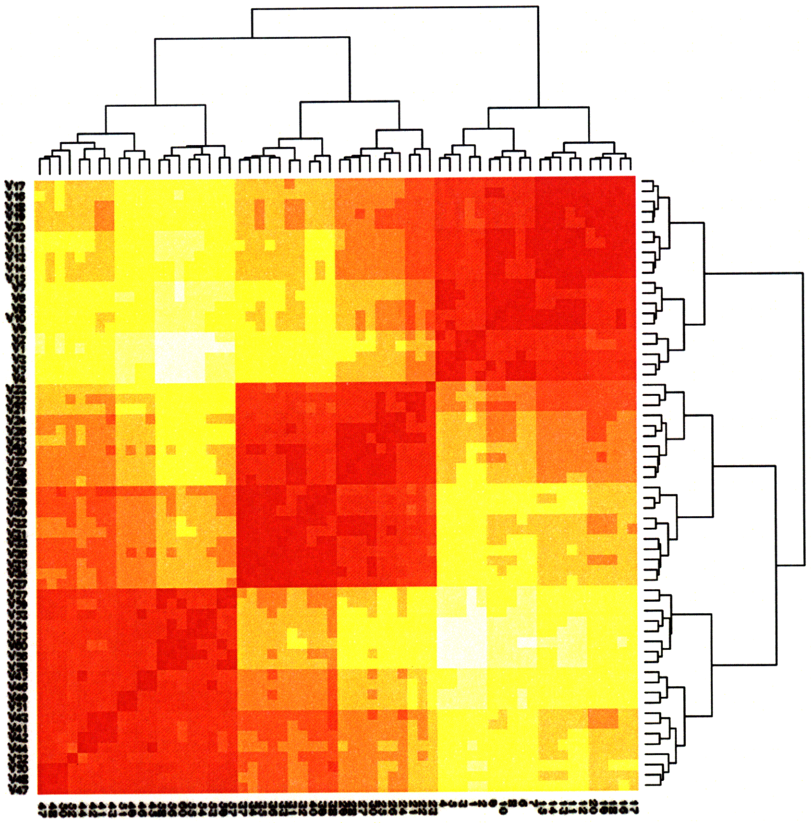

Figure 4-2: Heatmap of clusters of the synthetic data based on the average incidence matrix from the homoscedastic model. Higher intensity of red denotes more likelihood of clustering together.

eters. Each iteration produced a configuration structure of the genes that cluster together, giving rise to an incidence matrix. The average incidence matrix was used to ascertain the "average cluster structure" of the genes. The results are provided in Figures 4-4 and 4-7. The median number of clusters using the homoscedastic model is 5 with a 95% credible interval being (1, 13) and the corresponding figures for the heteroscedastic model were 7 and (2, 14) respectively. The histograms of the number of clusters are given in Figures 4-3 and 4-6. The above models were coded in C, and can be found in Appendix A.

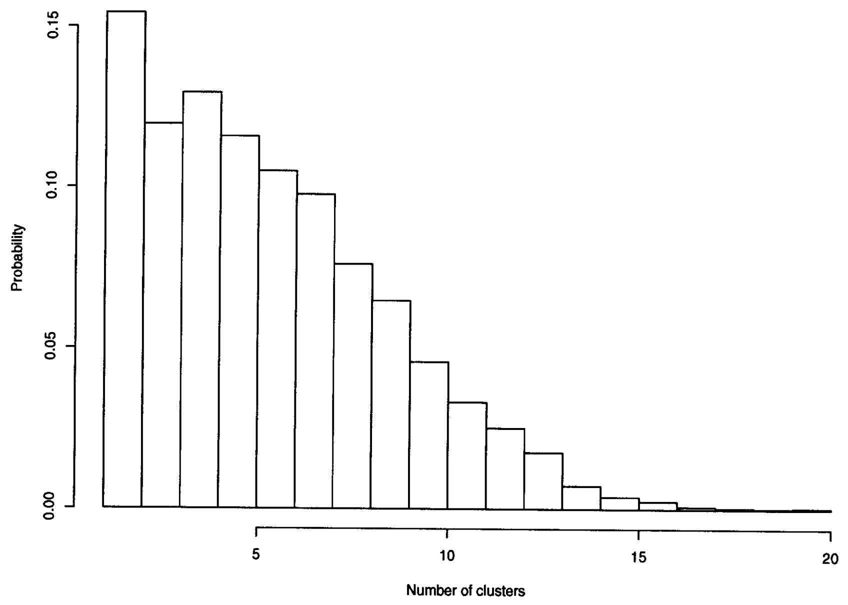

Figure 4-3: Histogram of number of clusters generated using the homoscedastic model.

A plot of the mean expression profiles for the top 5 clusters corresponding to the heatmap and the resulting dendrogram for this homoscedastic model is given in Figure 4-5.

The measure of separation between clusters in this model gives us a value of 40.13.

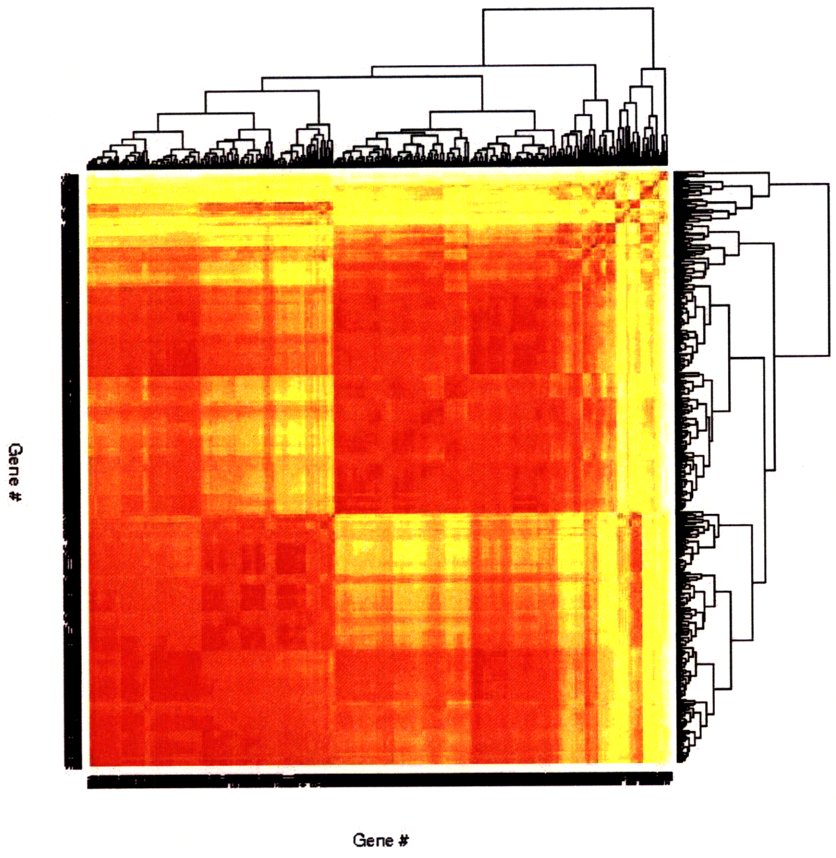

Figure 4-4: Heatmap of clusters of the 798 genes based on the average incidence matrix for the homoscedastic model. Higher intensity of red denotes more likelihood of clustering together.

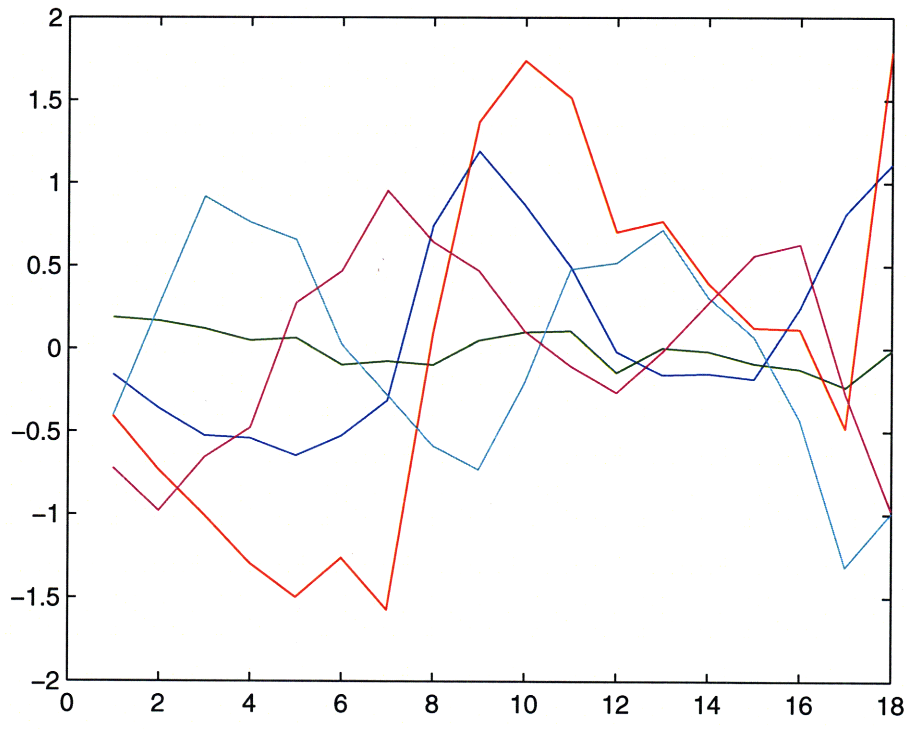

Figure 4-5: Mean curves of top 5 clusters found by the homoscedastic model.

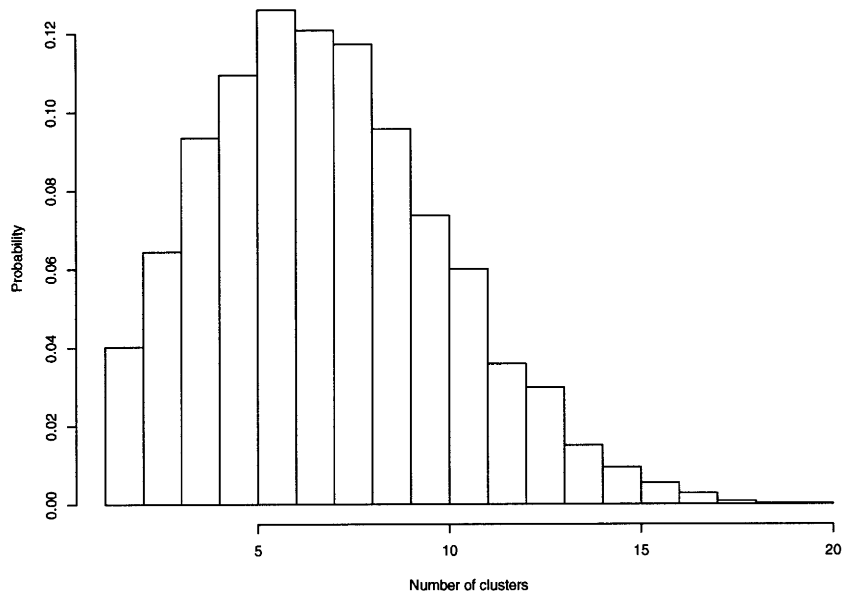

Figure 4-6: Histogram of number of clusters generated using the heteroscedastic model.

A plot of the mean expression profiles for the top 5 clusters corresponding to the heatmap and the resulting dendrogram for this heteroscedastic model is given in Figure 4-8.

The between-cluster separation measure has a a value of 39.68, barely different from that for the homoscedastic model. The closeness of the results from these models can also be seen visually from Figures 4-5 and 4-8. As a check of how robust our choice of prior parameters on M are, we changed these to yield an expected number of clusters that is closer to 5. This results in the following mean curves for the top 5 clusters, which are nearly indistinguishable from the previous ones. This assures us that the hyperparameters do not play a very significant role, which is generally a point of concern in Bayes methods.

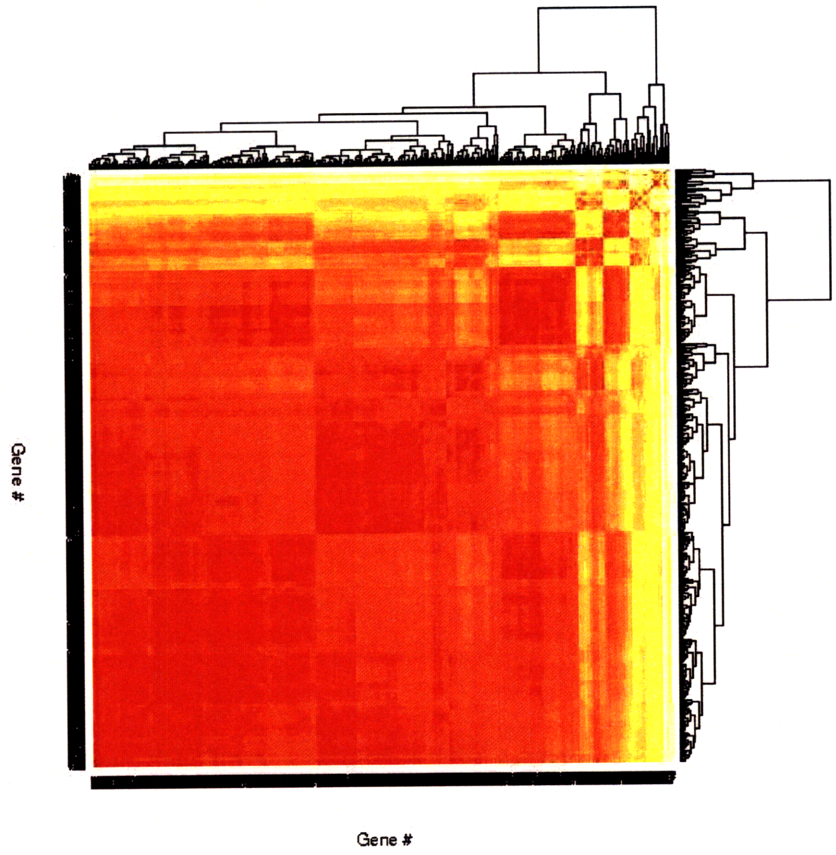

Figure 4-7: Heatmap of clusters of the 798 genes based on the average incidence matrix for the heteroscedastic model. Higher intensity of red denotes more likelihood of clustering together.

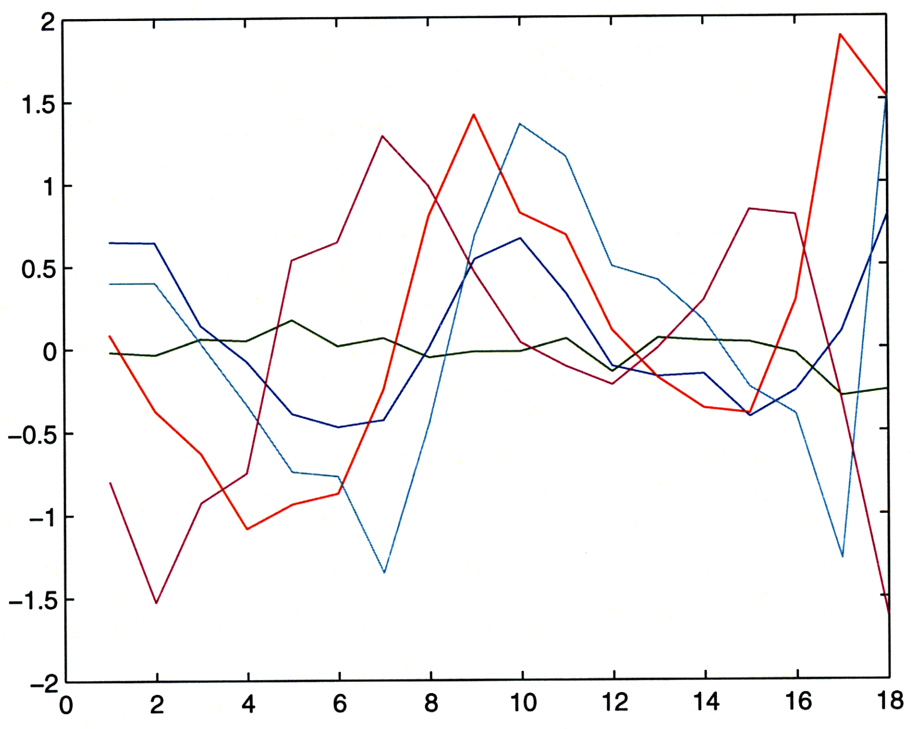

Figure 4-8: Mean curves of top 5 clusters found by the heteroscedastic model.

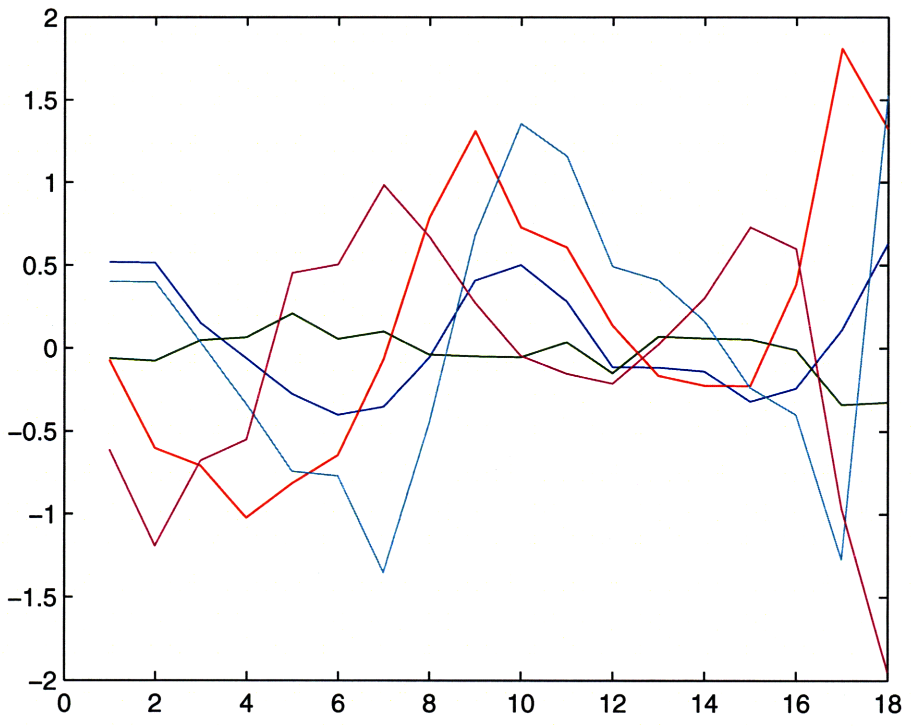

Figure 4-9: Mean curves of top 5 clusters found by the heteroscedastic model with hyperparameters set for a mean of 5 clusters.

4.4 Conclusion

We have provided a semiparametric Bayesian approach to clustering genes based on their expression profiles, which we assume have certain correlation structure. This method is easily implemented using standard computer languages and produces results in a reasonable period of time (it took about 2 hours to cluster the 798 genes on a PC). One can easily add additional steps to the Gibbs sampler to make the model more robust to mis-specifications (e.g. using a multivariate t instead of multivariate normal). The model can also be used to handle missing data by replacing them with their imputed values or as an additional MCMC step.

Chapter 5

Clustering Based on Distributions

5.1 Introduction

In this chapter, we consider two different approaches for clustering gene profiles. The first of these is based on the recently developed Nested Dirichlet Process which in its primary step, clusters together genes if they have a common distributional profile. It can also provide a clustering of time points nested within such gene-clusters. This is a purely nonparametric Bayesian model. The second model is analytically simpler but conceptually similar in that it clusters genes based on the parameter values of the distributions, like for instance the mean and variance in a Gaussian distribution. This is a semi-parametric model and uses Dirichlet process mixtures as discussed before.

5.2 Clustering of Distributions with Nested Dirichlet Process

5.2.1 Introduction

In this part of the chapter we consider an approach which models the data for each gene with a nonparametric distribution, and clusters those genes which have similar distributional profiles, while at the same time looking for possible clustering of time points "nested" within such gene-clusters. This approach is based on Nested Dirichlet Processes (NDPs) developed recently by Rodriguez et al [45] which we discuss below.

5.2.2 The Nested Dirichlet Process

Suppose there are N genes under consideration and we have p_i time-series measurements on the expression level of gene i. Let y_{ij} denote the expression value for gene i at time t_{ij}, $(j = 1, \ldots, p_i,\ i = 1, \ldots, N)$. The observed data are then given by

$$\{(y_{ij},\ t_{ij}),\ j = 1,\ \ldots,\ p_i,\ i = 1,\ \ldots,\ N\}$$

where the y_{ij}s are assumed to be random and t_{ij}s are fixed.

Note that the time-points t_{ij} need not be regularly spaced, and the number and location of time-points can vary from gene to gene (allowing for missing data), thus making our framework very general, and capable of capturing a wide variety of contexts.

The model that we are about to discuss assumes that the expression values within a gene are exchangeable; that is, $y_{ij}|F_i \overset{iid}{\sim} F_i,\quad j = 1, \ldots, p_i$ for $i = 1, \ldots, N$. In the time-series context that we are dealing with, it is more reasonable to make this assumption of the transition slopes:

$$b_{ij} = \frac{y_{i,j+1} - y_{i,j}}{t_{i,j+1} - t_{i,j}}$$

and follow the NDP model for this transformed set of slope vectors, as we do.

One approach that we have already used before and will use again in the next part of this chapter, is to parameterize F_i in terms of the finite-dimensional parameter θ_i, and then borrow information by assuming that these θ_i are iid with some known distribution F_0, possibly having unknown parameters. We then cluster genes having similar random effects, θ_i. The "borrowing of information" in this case is restricted to the parametric model being used and might ignore certain subtle aspects, like eg. the differences in the tails of the distribution. As we have done before, this can be overcome by using a Dirichlet process mixture i.e. clustering across

$$F_i(.) = \int p(.|\theta)dG_i(\theta)$$

where the $G_i, \quad i = 1, ..., N$ are iid from $\mathcal{D}(M, \; G_0)$. We take this one step further here by taking $G_i, \quad i = 1, ..., N$ as iid from a Nested Dirichlet Process. Such an NDP prior, can be placed on the collection of distributions for the different genes.

An easy way to formulate a Nested Dirichlet Process is to start with the usual stick-breaking representation of a Dirichlet process, but replace the draws with random probability measures drawn from another Dirichlet process. Formally, we say a collection of distributions $G_i, \quad i = 1, ..., N$ follows a Nested Dirichlet Process with parameters α, β, H and denote it by NDP(α, β, H) if

$$G_j(.) \sim \sum_{k=1}^{\infty} \pi_k^* \delta_{G_k^*(.)}$$

and

$$G_k^*(.) = \sum_{l=1}^{\infty} w_{lk}^* \delta_{\theta_{lk}^*}(.)$$

with $\theta_{lk}^* \sim H$, $w_{lk}^* = u_{lk}^* \Pi_{s=1}^{l-1}(1 - u_{sk}^*)$, $\pi_k^* = v_k^* \Pi_{s=1}^{k-1}(1 - v_s^*)$, $v_k^* \sim beta(1, \alpha)$, and $u_{lk}^* \sim beta(1, \beta)$. Thus conditional on H, the distinct atoms G_k^* are independent in an NDP.

In the spirit of Dirichlet process mixture models, we say a collection $F_i, \quad i = 1, ..., N$ follows a NDP mixture model if

$$y_{ij} \sim p(y|\theta_{ij}), \qquad \theta_{ij} \sim G_i, \qquad (G_1, \ldots, G_N) \sim NDP(\alpha, \beta, H)$$

ie. the collection $(G_1, \ldots, G_N)$ is used as the mixing distribution.

Remark: It is worth noting that the $\{G_i\}$ drawn from a Nested Dirichlet Process have either the same atoms with the same weights – for instance the genes i and i' are clustered together when $G_i = G_{i'} = G_k^*$ for some k, or have completely different atoms and weights. One can verify that for an NDP

$$P(G_i = G_{i'}) = \frac{1}{\alpha + 1}.$$

This is in contrast to the $\{G_i\}$ drawn from a Hierarchical Dirichlet Process ([52]), which share the same set of atoms (coming from the common parent distribution $H(.)$) but with distinct weights. As a consequence, for a Hierarchical Dirichlet Process, $P(G_i = G_{i'}) = 0$ and no clustering of distributions occurs. Clustering occurs at the level of observations only. The exact clustering of distributions in an Nested Dirichlet Process can be viewed as an idealization and in practice, one would exact two distributions that are close will be in the same cluster.

5.2.3 Nested Clustering Versus Biclustering

Although the model takes into account all the data (across genes and time-points) initially, the clusters across time-points are obtained separately for each gene-cluster, and could be quite different for each gene-cluster. In an NDP, genes i and i' are clustered together if $G_i = G_{i'}$ while time points j and j' from genes i and i' are clustered together if and only if $G_i = G_{i'}$ and $\theta_{ij} = \theta_{i'j'}$ ie. an NDP clusters time-points within the gene by borrowing information across gene-clusters with similar characteristics. While it may be quite relevant and sensible to cluster j and j' when the data within each gene is iid, such second-level clustering of time-points may not be meaningful in a time-series context because the time point j' in gene i' can be quite distinct from time point j from gene i. This NDP clustering is thus to be distinguished from the standard methods of "biclustering" where the goal is to cluster in both directions i.e. cluster the genes as also the time-points looking all across the genes. See for example [11] or the more recent article by Martella et al [35] and the references cited there. Gerber et al [20] develop an unsupervised computational algorithm for simultaneously organizing genes into overlapping "expression programs" (co-activated genes orchestrating a physiological process) and tissues into groups, and then demonstrate its application on human and mouse body-wide gene expression data. While biclustering is similar in spirit to the two-factor designs in an analysis of variance setting, clustering based on NDPs is analogous to "nested designs" where one factor is nested within the other.

This model is applied to gene expression data using code written with WinBUGS ([33]), which can be found in Appendix A. Figure 5-1 illustrates the results of clustering on the same subset of the Spellman time-series used previously.

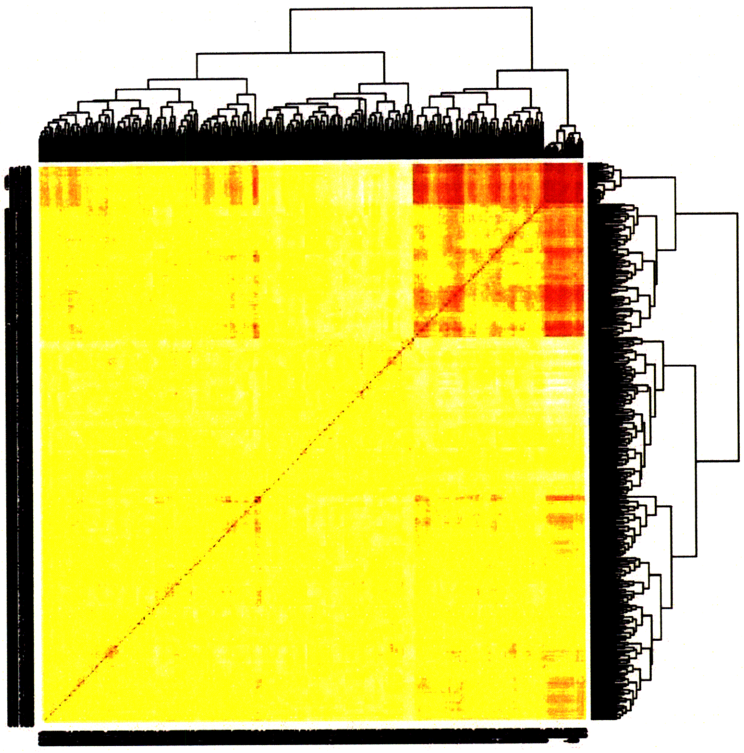

Figure 5-1: Heat map of average incidence matrix for nested model.

A plot of the mean expression profiles for the top 5 clusters corresponding to the heatmap and the resulting dendrogram for this nested model is given in Figure 5-2. with a between-cluster separation value of 0.36, an extremely small value indicating

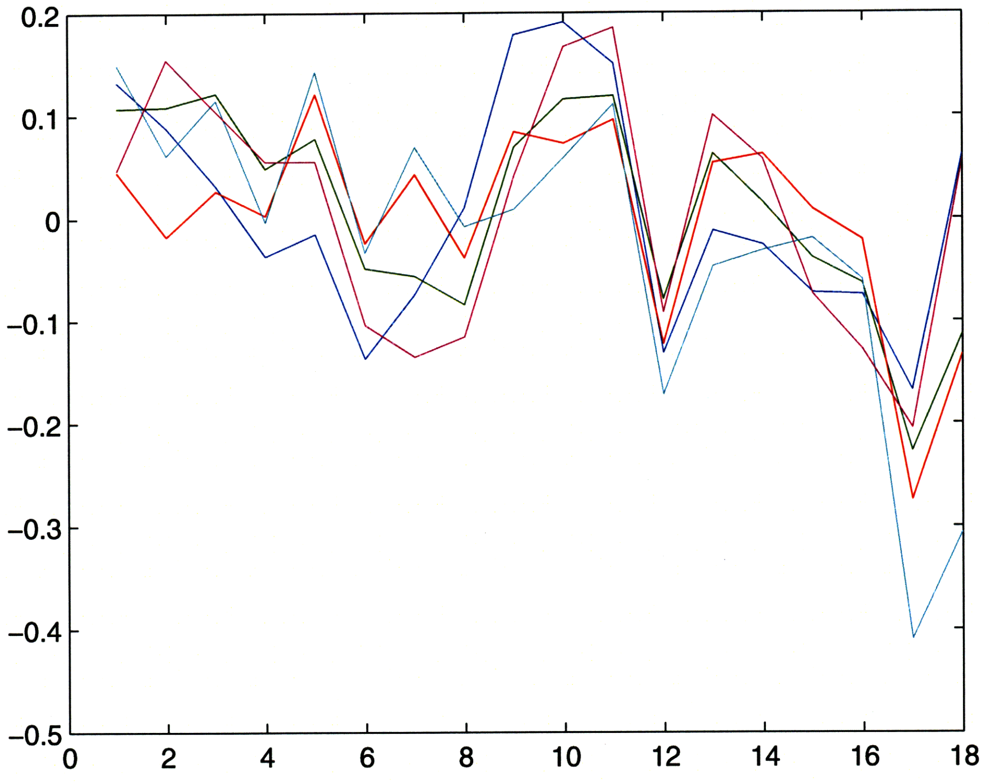

Figure 5-2: Mean curves of top 5 clusters found by the nested model.

that the nested model does not help separate genes in this case. This could be due to a variety of factors, but the most prominent is that this type of model is not necessarily the appropriate one to separate genes on the basis of the metrics we have described. While it may prove useful in other contexts, and is an interesting approach to consider from a modeling perspective, it does not particularly mesh with our goals here.

5.3 Clustering Based on Distributions of Slope Parameters

We consider once again the transition slopes from one time point to the next, for each gene. The clustering of genes in this simpler model, is based on the parameters of the slope distributions eg. its mean and variance if one assumes, as we do, that the slopes for the i^{th} gene have distribution $F_i(.) = N(\mu_i, \sigma_i^2)$. As we remarked earlier in the chapter, this semi-parametric model can be thought of as a special case of the NDP, although there is no second-level clustering occurring in this model.

5.3.1 The Model

The model we propose is now described below: We assume that for any particular gene, the transition of expression level from one time-point to the next occurs in a linear fashion in time with the local slope being represented by β_{ij}, and is given by:

$$y_{i,j+1}|(y_{i1}, \ldots, y_{ij}), \beta_{ij} \sim N(y_{ij} + \beta_{ij}(t_{i,j+1} - t_{ij}),\ \sigma^2(t_{i,j+1} - t_{i,j})^2). \tag{5.1}$$

This implies that each measurement is normally distributed around the previous measurement plus a distance depending on how far apart these measurements are made in time. Also, the variance term allows for a larger dispersion around this mean if the next time-point $t_{i,j+1}$ is farther away from the current point t_{ij}.

Further, the slopes β_{ij} of the individual transitions for a particular gene are assumed to have a normal distribution with fixed mean and variance, which may vary from one gene to the next. The goal is to cluster together those genes which have the same mean and variance for these transition slopes. This clustering is achieved by assuming a Dirichlet process prior on the joint distribution of the mean-variance pair.

Before writing down the model formally, recall that a random variable is said to have a gamma distribution, written as $\theta \sim \text{Gamma}(a, b)$ if it has the probability density function (pdf)

$$p(\theta) = \frac{1}{b^a \Gamma(a)} e^{-(\frac{\theta}{b})} \theta^{(a-1)},\ \theta > 0.$$

The reciprocal of this gamma random variable is called the inverse-gamma, for which we write $\theta \sim IG(a, b)$, and it has the probability density function (pdf)

$$p(\theta) = \frac{1}{b^a \Gamma(a)} e^{-(\frac{1}{b\theta})} \theta^{-(a+1)},\ \theta > 0,$$

where $a > 0$ is called the shape parameter, and $b > 0$ the scale parameter. Note that the inverse-gamma with $a = \frac{\nu}{2}$ and $b = 2$ is the same as the inverse-χ^2 distribution with ν degrees of freedom (which we will also use here).

The model:

- For each i,

$$\beta_{ij} | (\mu_i,\ \sigma_i^2) \overset{iid}{\sim} N(\mu_i,\ \sigma_i^2);\ j = 1,\ \ldots,\ p_i - 1. \tag{5.2}$$

- $$(\mu_i,\ \sigma_i^2) | G \overset{iid}{\sim} G \tag{5.3}$$

- $$G \sim \mathcal{D}(M,\ G_0), \tag{5.4}$$

  where, under G_0

$$\mu_i | \sigma_i^2 \sim N(\mu_0,\ \sigma_i^2) \tag{5.5}$$

  and

$$\sigma_i^2 \sim (\nu \sigma_0^2) Inv.\chi_\nu^2 \tag{5.6}$$

Such a choice of the baseline distribution G_0 is known as the "normal-inverse-χ^2" (NIC) prior with parameters $[\mu_0, \sigma_0^2, \nu]$ and is a natural conjugate prior in our case, leading to somewhat less complex MCMC updating. A fuller Bayesian specification of the model is completed with the two additional steps:

$$\sigma^2 \sim IG(a_\sigma,\ b_\sigma) \tag{5.7}$$

and

$$M \sim \text{Gamma}(a_M,\ b_M). \tag{5.8}$$

To summarize, the idea is to group genes based on similar dynamics of their expression time-series, as characterized by the distribution of their slopes – in particular, the mean and variance of their slopes. Our goal is to use this model to detect clustering of the genes based on their expression profiles. Two genes will be said to belong to the same cluster if the distributions corresponding to the means and variances of their transition slopes are similar.

The above model is an extension of the local linear model proposed by [21] in the context of short-term prediction of cancer mortality counts based on time-series of such count data. In that paper, it was shown that the local quadratic model produced better predictions than the local linear model. Although extension of the local quadratic model to gene expression time-series data is certainly possible, we did not pursue that angle in this paper as our main goal here is clustering rather than prediction.

Extension to covariates

Often, additional information in the form of covariates is available either for each gene, or perhaps even at each point of time for each gene. For example, additional data related to chromatin structure may prove to be important and useful covariate information. In situations like these, it seems natural to incorporate these covariates to improve our clustering. The proposed model can be easily modified to include such covariate information.

Formally, for $j = 0, \dots,\ p_i$, suppose that

$$y_{i,j+1} | (y_{ij},\ \boldsymbol{\beta}_j,\ \sigma^2) \overset{\text{indep.}}{\sim} N(y_{ij} + \boldsymbol{x}_j' \boldsymbol{\beta}_j,\ \sigma^2),$$

where $\boldsymbol{x}_j$ is a r-dimensional covariate vector at time t_j, $\boldsymbol{\beta}_j$ is the corresponding unknown coefficient vector and $'$ denotes the transpose. This covariate vector may include time as one of its components. For example, we might have $r = 2$ and $\boldsymbol{x}_j' = ((t_{i,j+1} - t_{ij}),\ a_{ij})$ where a_{ij} is some abstraction of a chromatin "value" at time t_{ij}. As before, we assume that $\boldsymbol{\beta}_0,\ \dots,\ \boldsymbol{\beta}_{p_i-1} | G \overset{\text{i.i.d.}}{\sim} G$, $G|(M,\ G_0) \sim \mathcal{D}(M,\ G_0)$ and $G_0 \equiv N_r(\boldsymbol{\mu}_i,\ \Sigma_i)$. In this case, $\boldsymbol{\beta}_j$ will be a multidimensional random variable whose mean vectors and covariance matrices jointly will have a Normal-Inverse-Wishart distribution. It is somewhat straightforward to rewrite the model and the Gibbs sampling scheme for this general setup, although we do not provide the details here.

5.3.2 Computational Details

Posterior Sampling

Sampling from the posterior is done using the iterative Markov chain Monte Carlo method. Starting from initial values for the parameters

$$\boldsymbol{\theta} = (\boldsymbol{\beta},\ \sigma^2, \mu_i,\ \sigma_i^2,\ M),$$

the components are repeatedly sequentially updated. For convenience, we denote

$$\tau_i = \begin{pmatrix} \mu_i \\ \sigma_i^2 \end{pmatrix}$$

After "burning-in" the first few iterations, the "chain" of $\boldsymbol{\theta}$ reaches the steady state and further samples generated from this chain are samples from the posterior distribution. This method requires knowledge of the univariate posterior conditionals of any of these parameters given all the rest, which we discuss below.

Due to the hierarchical Bayesian nature of the problem, we will need to sample the slopes from the posterior distribution of $(\mu_i,\ \sigma_i^2)$ at each step. We will use the Gibbs sampler method to sample from the univariate conditionals arising from the posterior distribution.

For implementing the MCMC algorithm, we need to update the univariate conditionals, in the order given below.

Note that in all that follows we write $\theta|...$ to represent the conditional distribution of a parameter θ of interest, given all the rest of the parameters and the data.

I. Update σ^2 (follows from Part (ii) of Proposition 1 and the assumption in Equation (7))

$$\sigma^2| \cdots \sim IG(a_\sigma^*, b_\sigma^*)$$

with

$$a_\sigma^* = a_\sigma + \frac{1}{2}\sum_{i=1}^{N}(p_i - 1),$$

and

$$b_\sigma^* = \left[b_\sigma^{-1} + \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{p_i-1} \frac{\{y_{i,j+1} - y_{i,j} - \beta_{i,j}(t_{i,j+1} - t_{i,j})\}^2}{(t_{i,j+1} - t_{i,j})^2} \right]^{-1}.$$

II. Update slopes (β_{ij}) (follows from Part (i) of Proposition 1 and the assumption in Equation (2))

$$\beta_{i,j}|\cdots \sim N\left(\left(\frac{y_{i,j+1}-y_{i,j}}{\sigma^2(t_{i,j+1}-t_{i,j})}+\frac{\mu_i}{\sigma_i^2}\right)\left(\frac{1}{\sigma^2}+\frac{1}{\sigma_i^2}\right)^{-1}, \left(\frac{1}{\sigma^2}+\frac{1}{\sigma_i^2}\right)^{-1}\right).$$

III. Update τ_i

There are two parts to this. Recall $\tau_i = \begin{pmatrix} \mu_i \\ \sigma_i^2 \end{pmatrix}$. Since the direct algorithm for updating the τ_i's used by [17] may be rather slow, we use a modification used in [58] and [10]. The modification uses the fact that with positive probability, not all $\tau_i = \begin{pmatrix} \mu_i \\ \sigma_i^2 \end{pmatrix}$'s are distinct (see [2]). At any stage, let there be k distinct values $\boldsymbol{\tau}^* = (\tau_1^*, \ldots, \tau_k^*)$ with multiplicities $n_1, \ldots, n_k$, where $n_1 + \cdots + n_k = N$. Let $\mathbf{c} = (c_1, \ldots, c_N)$ be the configuration vector of $\boldsymbol{\tau}$, where $c_j = l$ if and only if $\tau_j = \tau_l^*$. The modified algorithm updates the configuration vector $\mathbf{c}$ and the distinct values τ^* sequentially and avoids the chain getting stuck in a few distinct clusters. This is Algorithm 2 of [39].

a. Update the configuration vector

Proposition 2 *Let $\mathbf{c} = (c_1, \ldots, c_N)$ be the configuration vector and $\phi = (\phi_c : c \in \{c_1, \ldots, c_N\})$ be the set of distinct values of τ_is. For each $i = 1, \ldots, N$, we do the following:*

If $n_{c_i}^{(-i)} = 0$ i.e., c_i does not occur anywhere else on $\mathbf{c}^{(-i)}$, remove ϕ_{c_i} from the state and draw a new value of c_i.

If $c_i = c_j$ for some $j \neq i$, draw a new value for c_i according to the following scheme:

$$\begin{aligned} P(c_i = c|\cdots) &\propto n_c^{(-i)}(\sqrt{2\pi}\sigma_c^*)^{-p_i} \exp\left\{\tfrac{-1}{2\sigma_c^{*2}} \textstyle\sum_{j=1}^{p_i-1}(\beta_{ij} - \mu_{c^*})^2\right\} \\ P(c_i \neq c_j \forall j \neq i|\cdots) &\propto \frac{M}{(\sqrt{2\pi})^{p_i}\sqrt{p_i+1}} \left(\frac{\nu\sigma_0^2}{2}\right)^{\nu/2} \frac{\Gamma(\frac{p_i+\nu}{2})}{\Gamma(\frac{\nu}{2})} \left[\frac{\nu\sigma_0^2+\sum_{j=1}^{p_i-1}\beta_{i,j}^2+\mu_0^2-\frac{(\sum_{j=1}^{p_i-1}\beta_{i,j}+\mu_0)^2}{p_i+1}}{2}\right]^{\frac{-(p_i+\nu)}{2}} \end{aligned}$$

Proof: Writing μ^* in place of μ_i and σ^* in place of σ_i for the common cluster mean and variance, we have

$$\begin{aligned} &P(c_i \neq c_j \forall j \neq i|c^{-i}, \beta_i, \phi) \propto \\ &M \int\int \frac{1}{(\sqrt{2\pi}\sigma^*)^{p_i}} e^{-\frac{1}{2\sigma^{*2}}\sum_{j=1}^{p_i-1}(\beta_{ij}-\mu^*)^2} \frac{1}{\sqrt{2\pi}\sigma^*} e^{-\frac{1}{2\sigma^{*2}}(\mu^*-\mu_0)^2} p(\sigma^{*2}) d\mu^* d\sigma^{*2} \end{aligned}$$

where, from Equation (6),

$$p(\sigma^{*2}) = \frac{1}{(2)^{\frac{\nu}{2}}\Gamma(\frac{\nu}{2})} e^{-\frac{\nu\sigma_0^2}{2\sigma^{*2}}} \frac{(\nu\sigma_0^2)^{\frac{\nu}{2}}}{(\sigma^{*2})^{\frac{\nu}{2}+1}}. \tag{5.9}$$

Focusing first on the terms involving μ^*,

$$\begin{aligned} &\exp\left\{-\tfrac{1}{2\sigma^{*2}}\left[\textstyle\sum_j(\beta_{ij} - \mu^*)^2 + (\mu^* - \mu_0)^2\right]\right\} \\ &= \exp\left\{-\tfrac{1}{2\sigma^{*2}}\left[\textstyle\sum_j(\mu^{*2} + \beta_{ij}^2 - 2\mu^*\beta_{ij}) + (\mu^{*2} + \mu_0^2 - 2\mu^*\mu_0)\right]\right\} \\ &= \exp\left\{-\tfrac{1}{2\sigma^{*2}}\left[(p_i+1)\mu^{*2} - 2\mu^*\left(\textstyle\sum_j \beta_{ij} + \mu_0\right) + \textstyle\sum_j \beta_{ij}^2 + \mu_0^2\right]\right\} \\ &= \exp\left\{-\tfrac{1}{2\sigma^{*2}}(p_i+1)\left[\mu^{*2} - 2\mu^*\left(\tfrac{\sum_j \beta_{ij}+\mu_0}{p_i+1}\right) + (\cdots)^2\right]\right\} \\ &= \exp\left\{-\tfrac{1}{2\sigma^{*2}}(p_i+1)\left[\mu^* - \tfrac{\sum_j \beta_{ij}+\mu_0}{p_i+1}\right]^2\right\} exp\left\{-\tfrac{1}{2\sigma^{*2}}\left[\textstyle\sum_j \beta_{ij}^2 + \mu_0^2 - \left(\tfrac{\sum_j(\beta_{ij}+\mu_0)}{p_i+1}\right)^2(p_i+1)\right]\right\} \end{aligned}$$

After integrating out μ^*, the expression becomes

$$\begin{aligned} &M \int \frac{1}{(\sqrt{2\pi}\sigma^*)^{p_i}} \left(\frac{\nu\sigma_0^2}{2}\right)^{\frac{\nu}{2}} \frac{e^{-\frac{\nu\sigma_0^2}{2\sigma^{*2}}}}{\Gamma(\frac{\nu}{2})} \frac{1}{(\sigma^{*2})^{\frac{\nu}{2}+1}} \frac{1}{\sqrt{p_i+1}} exp\left\{-\frac{1}{2\sigma^{*2}}\left[\textstyle\sum_j \beta_{ij}^2 + \mu_0^2 - \frac{\left(\sum_j \beta_{ij}+\mu_0\right)^2}{p_i+1}\right]\right\} d\sigma^{*2} \\ &= \frac{M}{(\sqrt{2\pi})^{p_i}} \sqrt{\frac{1}{p_i+1}} \left(\frac{\nu\sigma_0^2}{2}\right)^{\frac{\nu}{2}} \frac{1}{\Gamma(\frac{\nu}{2})} \int \frac{1}{(\sigma^{*2})^{\frac{p_i}{2}+\frac{\nu}{2}+1}} exp\left\{-\frac{1}{2\sigma^{*2}}\left[\nu\sigma_0^2 + \textstyle\sum_j \beta_{ij}^2 + \mu_0^2 - \frac{(\sum_j \beta_{ij}+\mu_0)^2}{p_i+1}\right]\right\} d\sigma^{*2} \end{aligned}$$

Putting

$$\frac{1}{2\sigma^{*2}}\left[\nu\sigma_0^2+\sum_j\beta_{ij}^2+\mu_0^2-\frac{(\sum_j\beta_{ij}+\mu_0)^2}{p_i+1}\right]=y$$

with $\frac{1}{2}\left[\nu\sigma_0^2+\sum_j\beta_{ij}^2+\mu_0^2-\frac{(\sum_j\beta_{ij}+\mu_0)^2}{p_i+1}\right]=a$, so that ${\sigma^*}^2=\frac{a}{y}$, the expression becomes

$$\begin{aligned}
&\frac{M}{(\sqrt{2\pi})^{p_i}}\sqrt{\frac{1}{p_i+1}}\left(\frac{\nu\sigma_0^2}{2}\right)^{\frac{\nu}{2}}\frac{1}{\Gamma(\frac{\nu}{2})}\int_0^\infty\left(\frac{y}{a}\right)^{\frac{p_i+\nu}{2}+1}e^{-y}\frac{a}{y^2}dy\\
&=\frac{M}{(\sqrt{2\pi})^{p_i}}\sqrt{\frac{1}{p_i+1}}\left(\frac{\nu\sigma_0^2}{2}\right)^{\frac{\nu}{2}}\frac{1}{\Gamma(\frac{\nu}{2})}\frac{1}{a^{\frac{p_i+\nu}{2}}}\int_0^\infty e^{-y}y^{\frac{p_i+\nu}{2}-1}dy\\
&=\frac{M}{(\sqrt{2\pi})^{p_i}}\sqrt{\frac{1}{p_i+1}}\left(\frac{\nu\sigma_0^2}{2}\right)^{\frac{\nu}{2}}\frac{\Gamma(\frac{p_i+\nu}{2})}{\Gamma(\frac{\nu}{2})}\frac{1}{a^{\frac{p_i+\nu}{2}}}
\end{aligned}$$

as stated in the proposition.

■

b. Update the distinct values of τ_i

Proposition 3 *The distinct values of $(\mu_i,\ \sigma_i^2)$, which have a $NIC(\mu_o,\sigma_0^2,\nu)$ prior, are updated according to $(\mu_c^*,\ \sigma_c^{*2})$ which have a $NIC(\mu_{c0}^*,\ {\sigma_{c0}^*}^2,\ \nu_c^*)$ where*

$$\begin{aligned}
\mu_{c0}^* &= \frac{\mu_0+\sum_{i:c_i=c}\sum_{j=1}^{p_i}\beta_{i,j}}{1+\sum_{i:c_i=c}p_i}\\
{\sigma_{c0}^*}^2 &= \left(\sum_{i:c_i=c}\sum_{j=1}^{p_i}\beta_{i,j}^2+\mu_0^2+\nu\sigma_0^2-\frac{(\mu_0+\sum_{i:c_i=c}\sum_{j=1}^{p_i}\beta_{i,j})^2}{1+\sum_{i:c_i=c}p_i}\right)\\
&\quad\left(\sum_{i:c_i=c}p_i+\nu+1\right)^{-1}\left(1+\sum_{i:c_i=c}p_i\right)^{-1}\\
\nu_c^* &= \sum_{i:c_i=c}p_i+\nu+1
\end{aligned}$$

Proof:

As before, we write μ^* in place of μ_i and σ^* in place of σ_i for the common cluster mean and variance, with the product or sum over i running over the genes belonging to the same cluster. Then, we have

$$\begin{aligned}
&\prod_i\left[\frac{1}{(\sqrt{2\pi}\sigma^*)^{p_i-1}}e^{-\frac{1}{2\sigma^{*2}}\sum_{j=1}^{p_i-1}(\beta_{ij}-\mu^*)^2}\right]\frac{1}{\sqrt{2\pi}\sigma^*}e^{-\frac{1}{2\sigma^{*2}}(\mu^*-\mu_o)^2}p({\sigma^*}^2)\\
&=\frac{1}{(\sqrt{2\pi}\sigma^*)^{\sum_i p_i}}e^{-\frac{1}{2\sigma^{*2}}\sum_i\sum_j(\beta ij-\mu^*)^2}\frac{1}{(\sqrt{2\pi}\sigma^*)}e^{-\frac{1}{(2\sigma^{*2})}(\mu^*-\mu_0)^2}p({\sigma^*}^2).
\end{aligned}$$

Focussing on the terms in the exponent that involve μ^*, in order to complete the quadratic, we have

$$
\begin{aligned}
&\tfrac{1}{\sigma^{*^2}} \textstyle\sum_i \sum_j (\mu^{*^2} + \beta_{ij}^2 - 2\mu^* \beta_{ij}) + \tfrac{1}{\sigma^{*^2}} (\mu^{*^2} + \mu_0^2 - 2\mu^* \mu_0) \\
&= \mu^{*^2} \left(\tfrac{1}{\sigma^{*^2}} + \tfrac{\sum_i p_i}{\sigma^{*^2}} \right) - 2\mu^* \left(\tfrac{\mu_0}{\sigma^{*^2}} + \textstyle\sum_{i,j} \tfrac{\beta_{ij}}{\sigma^{*^2}} \right) + \textstyle\sum_{i,j} \tfrac{\beta_{ij}}{\sigma^{*^2}} + \tfrac{\mu_0^2}{\sigma^{*^2}} \\
&= \left(\tfrac{1}{\sigma^{*^2}} + \tfrac{\sum_i p_i}{\sigma^{*^2}} \right) \left[\mu^{*^2} - 2\mu^* \frac{\left(\frac{\mu_0}{\sigma^{*^2}} + \frac{\sum_{i,j} \beta_{ij}}{\sigma^{*^2}} \right)}{\left(\frac{1}{\sigma^{*^2}} + \frac{\sum_i p_i}{\sigma^{*^2}} \right)} + (\cdots)^2 \right] - (\cdots)^2 \left(\tfrac{1}{\sigma^{*^2}} + \tfrac{\sum i p_i}{\sigma^{*^2}} \right) + \cdots \\
&= \left(\tfrac{1}{\sigma^{*^2}} + \tfrac{\sum_i p_i}{\sigma^{*^2}} \right) \left[\mu^* - \left(\tfrac{\mu_0 + \sum_{i,j} \beta_{ij}}{1 + \sum_i p_i} \right) \right]^2
\end{aligned}
$$

which corresponds to

$$
N\left(\frac{\mu_0 + \sum_{i,j} \beta_{ij}}{1 + \sum_i p_i}, \left(\frac{1}{\sigma^{*^2}} + \frac{\sum_i p_i}{\sigma^{*^2}} \right)^{-1} \right) = N\left(\frac{\mu_0 + \sum_{i,j} \beta_{ij}}{1 + \sum_i p_i}, \frac{\sigma^{*^2}}{1 + \sum_i p_i} \right).
$$

Turning our attention to σ^{*^2} and using its pdf given in Equation (9),

$$
\begin{aligned}
&\frac{1}{(\sigma^{*^2})^{\frac{\sum_i p_i + 1}{2} + \frac{\nu}{2} + 1}} e^{-\frac{\nu \sigma_0^2}{2\sigma^{*^2}}} exp \left\{ -\tfrac{1}{2} \left[\left(\tfrac{1}{\sigma^{*^2}} + \tfrac{\sum_i p_i}{\sigma^{*^2}} \right) \left(\tfrac{\mu_0 + \sum_{i,j} \beta_{ij}}{1 + \sum_i p_i} \right)^2 (-1) + \left(\tfrac{\sum_{i,j} \beta_{ij}^2}{\sigma^{*^2}} + \tfrac{\mu_0}{\sigma^{*^2}} \right) \right] \right\} \\
&= \frac{1}{(\sigma^{*^2})^{\frac{\sum_i p_i + \nu + 1}{2} + 1}} exp \left\{ -\tfrac{1}{2\sigma^{*^2}} \left[\textstyle\sum_{i,j} \beta_{ij}^2 + \mu_0^2 - \left(\tfrac{\mu_0 + \sum_{i,j} \beta_{ij}}{1 + \sum_i p_i} \right)^2 (1 + \textstyle\sum_i p_i) \right] \right\}
\end{aligned}
$$

which corresponds to $(\nu^* \sigma_0^{*^2}) Inv.\chi^2_{\nu^*}$ with the values as given in the statement of the proposition. ■

IV. Update on the precision parameter M

We give the basic idea in updating the precision parameter M and refer to Escobar and West [17] for further details. Let k be the number of distinct τ_is. If $\pi(M)$ denotes the prior density of M, then the posterior density is given by

$$
\begin{aligned}
\pi(M|\boldsymbol{\beta}) &\propto \pi(M) M^k \frac{\Gamma(M)}{\Gamma(M+N)} \\
&\propto \pi(M) M^{k-1} (M+N) \int_0^1 \eta^M (1-\eta)^{N-1} d\eta.
\end{aligned}
$$

Introducing η as a latent variable, the joint posterior density of $(M,\ \eta)$ is given by

$$\pi(M,\ \eta|\boldsymbol{\beta}) \propto \pi(M)M^{k-1}(M+N)\eta^{M}(1-\eta)^{N-1}, \qquad M>0,\ 0<\eta<1.$$

If $\pi(M) \sim \text{Gamma}(a_M,\ b_M)$, samples from the posterior distribution of M can be obtained by sequentially applying the following:

$$\eta|(M,\ \boldsymbol{\beta}) \sim Beta(M+1,\ N)$$

$$M|(\eta,\ \boldsymbol{\beta}) \sim \pi.\text{Gamma}(a_M+k,\ b_M-\log\eta)+(1-\pi).\text{Gamma}(a_M+k-1,\ b_M-\log\eta)$$

where $\frac{\pi}{1-\pi}=\frac{a_M+k-1}{N(b_M-\log\eta)}$.

Thus the steps in updating the precision parameter M are:

1. Generate $\eta \sim Beta(M+1,\ N)$.
2. Get π from $\frac{\pi}{1-\pi}=\frac{a_M+k-1}{N(b_M-\log\eta)}$
3. Generate M from $\pi\text{Gamma}(a_M+k,\ b_M-\log\eta)+(1-\pi)\text{Gamma}(a_M+k-1,\ b_M-\log\eta)$

See [16] for an alternative prior choice for M.

5.3.3 Guidelines for the Choice of Prior Parameters

The basic idea of Bayesian analysis consists of specifying a prior distribution of the model parameters to express uncertainty about their values and combining it with the observed data to obtain the posterior distribution. All further statistical inferences are based on this posterior. Correct choice of a prior is helpful in ensuring rapid convergence of the Gibbs sampler to the posterior distribution. Typically the prior distributions are ascertained from expert opinion or historical information. When the data set is large enough, part of it can be used for prior elicitation and the rest for model fitting. Or one can follow "empirical Bayes" methods in choosing the hyper-parameters, say employing the method of moments. This amounts to equating the theoretical moments of the marginal distributions which involve the hyperparameters, to those observed in the data. For our case, we need the following quantities from the data: we first compute the observed slopes

$$\hat{\beta}_{ij} = (y_{i,j+1} - y_{i,j})/(t_{i,j+1} - t_{i,j})$$

and their mean over all the rows and columns denoted by $\overline{\hat{\beta}_{..}}$ and their variance $s^2_{\hat{\beta}} = \frac{\sum(\hat{\beta}_{ij} - \overline{\hat{\beta}_{..}})^2}{(\sum p_i) - 1}$.

- Since μ_0 represents the mean of the all the slope distributions, the value suggested by the data and the method of moments, is

$$\mu_0 \;=\; \overline{\hat{\beta}_{..}}$$

- Let s_i^2 be the within-variance of the $\hat{\beta}_{ij}$s for the i^{th} gene and $\overline{s^2}$ be the average of such variances over all the genes. From Equation (6) and the moments of the Inverse χ^2_ν distribution, we get ν and σ_0^2 by equating

$$\begin{aligned} \frac{\nu}{\nu - 2}\sigma_0^2 &= \overline{s^2} \\ \frac{2\nu^2}{(\nu-2)^2(\nu-4)}\sigma_0^4 &= Var(s_i^2). \end{aligned}$$

- Equation(7) suggests that we choose a_σ, b_σ such that

$$\begin{aligned} \frac{1}{b_\sigma(a_\sigma - 1)} &= s^2_{\hat{\beta}} \\ \frac{1}{b_\sigma^2(a_\sigma - 1)^2(a_\sigma - 2)} &= 100 \end{aligned}$$

where the second equation suggests a large variance, allowing for considerable flexibility in the choice.

- Recall that the value of M in a Dirichlet prior is related to the expected number of clusters, which is given by the formula (see [2])

$$\sum_{i=0}^{N-1} \frac{M}{(M+i)}.$$

We take $M \sim$ Gamma(.1, .1). Note that $M = 1$ signifies that the probability of generating a new cluster is $\frac{1}{N+1}$, when we have a sample of size N. We also experimented with other choices such as $M \sim$ Gamma(1, 1) and $M \sim$ Gamma(.01, .01) and found the results to be very similar.

Empirical evidence based on our experiments with different data sets suggests that the above guidance provides reasonable answers and rapid convergence for the algorithms used for the clustering problem.

5.3.4 Data Analysis

Synthetic Data With Convergence Analysis

The implementation code used here is written in MATLAB as well as in in C, the latter of which can be found in Appendix A. To analyze the results of the clustering procedure proposed here, we generated a reasonable synthetic test data set. This data consists of 30 gene profiles, each containing 10 time points. The profiles are divided into three groups of 10, with different slope profiles for each group. The slopes of the first group have mean 0 and variance 0.1, those of the second group have mean 1 and variance 0.2, and those of the third group have mean 3 and variance 0.3. Hyperparameters were chosen according to the guidelines given in Section 5.3.3. Convergence of the Gibbs sampler was ascertained by running the output of the sampler through CODA (see [5], [13]), a package in the R environment. Below, we present the results of convergence diagnostics for selected parameters.

The traceplot of the variables suggests a rapid convergence. This is also supported by the results of Raftery and Lewis convergence diagnostic and the Heidelberger and Welch convergence diagnostic. Experimentation with various sites showed that a burn-in of 1000 cycles was more than adequate in each case. Since the autocorrelations died down quite rapidly (Figure 5-3), we used an additional 4,000 iterations of the Gibbs sampler to base our posterior computations on, thinning the result of every 10th iteration. This computation was performed in a matter of seconds.

```
Iterations = 1:5000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 5000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

                   Mean      SD  Naive SE Time-series SE
Sigma^2         0.01352 0.05486 0.0007759       0.001980
Beta(11,1)      0.71456 0.03726 0.0005269       0.001242
Mu_i(11)        0.98620 0.44505 0.0062939       0.007018
Sigma^2_i(11)   0.45553 0.47908 0.0067751       0.008196
M              11.69966 9.49918 0.1343387       0.074286

2. Quantiles for each variable:

                   2.5%     25%     50%      75%    97.5%
Sigma^2        0.009821 0.01091 0.01150  0.01214  0.01361
Beta(11,1)     0.680019 0.70335 0.71423  0.72435  0.74286
Mu_i(11)       0.334579 0.72073 0.91249  1.02333  2.14891
Sigma^2_i(11)  0.113218 0.16579 0.23658  0.40056  1.69373
M              1.667577 5.42220 9.05890 15.04200 37.36075
```

```
HEIDELBERGER AND WELCH STATIONARITY AND INTERVAL HALFWIDTH TESTS
=================================================================

Iterations used = 1:5000
Thinning interval = 1
Sample size per chain = 5000

Precision of halfwidth test = 0.3

$chain1

               Stationarity start     p-value
               test         iteration
Sigma^2        passed         1       0.650
Beta(11,1)     passed       501       0.779
Mu_i(11)       passed         1       0.165
Sigma^2_i(11)  passed         1       0.257
M              passed         1       0.322

               Halfwidth Mean    Halfwidth
               test
Sigma^2        passed      0.0135 0.003880
Beta(11,1)     passed      0.7132 0.000496
Mu_i(11)       passed      0.9862 0.013754
Sigma^2_i(11)  passed      0.4555 0.016063
M              passed     11.6997 0.145600


RAFTERY AND LEWIS CONVERGENCE DIAGNOSTIC
```

```
========================================

Iterations used = 1:5000

Thinning interval = 1

Sample size per chain = 5000


$chain1


Quantile (q) = 0.025

Accuracy (r) = +/- 0.005

Probability (s) = 0.95


               Burn-in  Total Lower bound  Dependence

               (M)      (N)   (Nmin)       factor (I)

 Sigma^2       2        3680  3746         0.982

 Beta(11,1)    2        3620  3746         0.966

 Mu_i(11)      2        3680  3746         0.982

 Sigma^2_i(11) 2        3620  3746         0.966

 M             3        4267  3746         1.140
```

Here, the selected values of hyperparameters were $\mu_0 = 2.33$, $\sigma_0^2 = 0.04$, $a_M = b_M = 0.01$, $a_\sigma = 2.03$, $b_\sigma = 0.59$, and $\nu = 7.68$. These values were drawn from the data as described earlier.

Figure 5-4 shows the cross-correlations for the same selected variables. Figure 5-5 shows the Geweke convergence diagnostic Z-scores. Finally, Figure 5-6 shows the output dendrogram of a standard hierarchical clustering scheme applied to the output configuration vectors of our model for this synthetic data. The 3 groups can be seen to be clustering as expected.

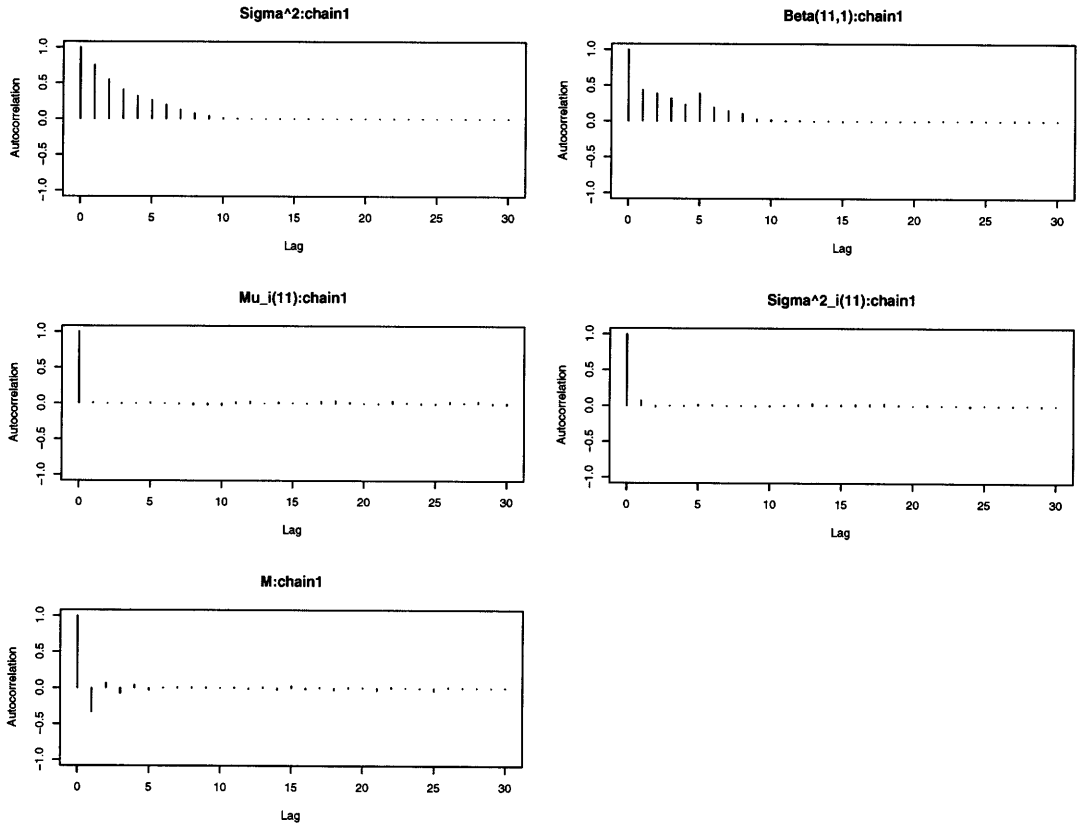

Figure 5-3: Autocorrelation of selected variables.

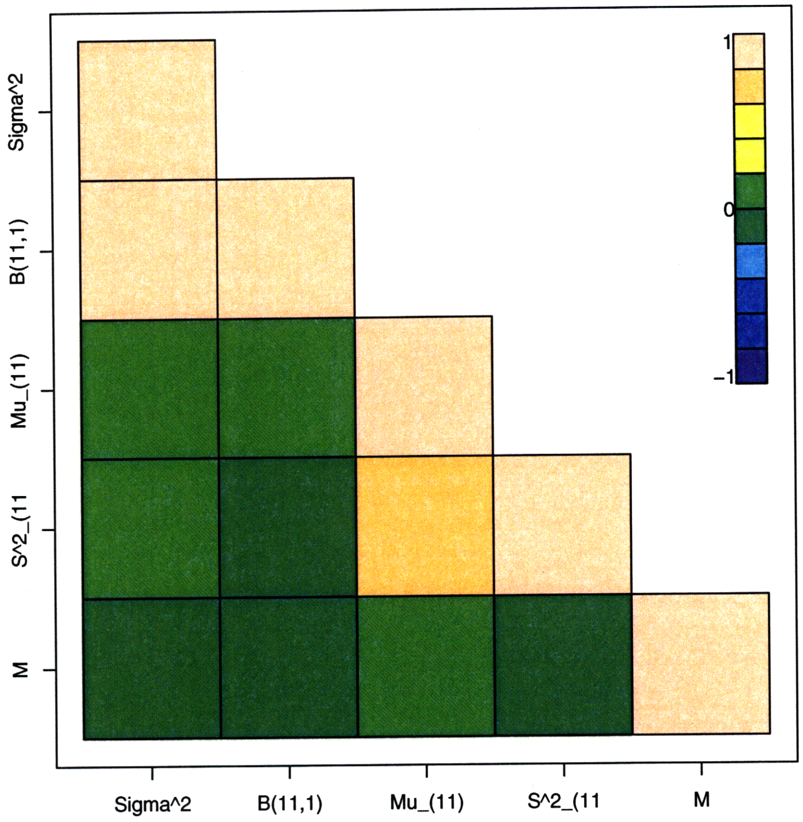

Figure 5-4: Cross-correlation of selected variables.

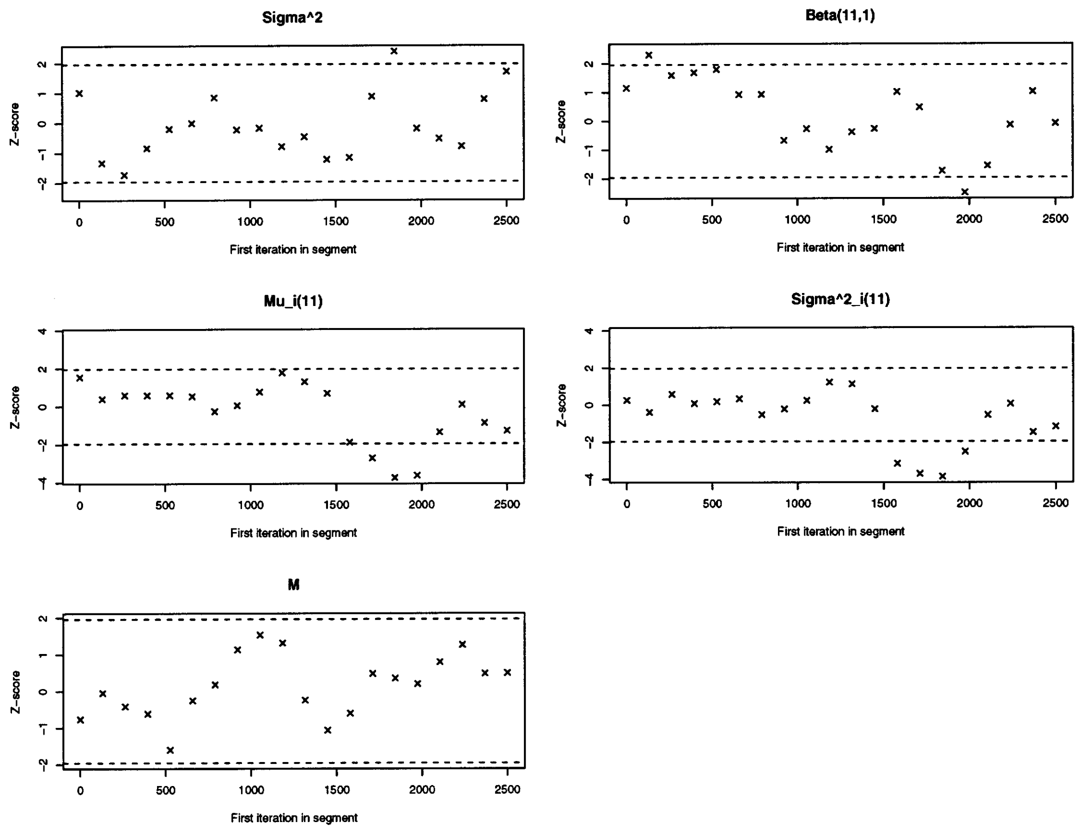

Figure 5-5: Geweke Z-scores for selected variables.

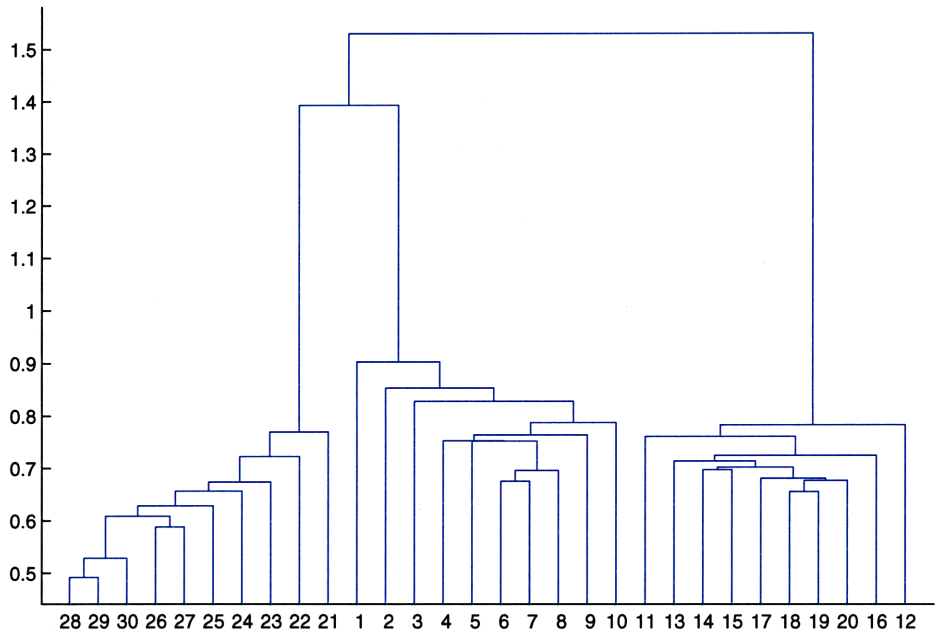

Figure 5-6: Dendrogram for synthetic data.

Data Analysis

This model was also applied to the same subset of the Spellman time-series data as the other models, and the results are shown in Figures 5-7 and 5-8.

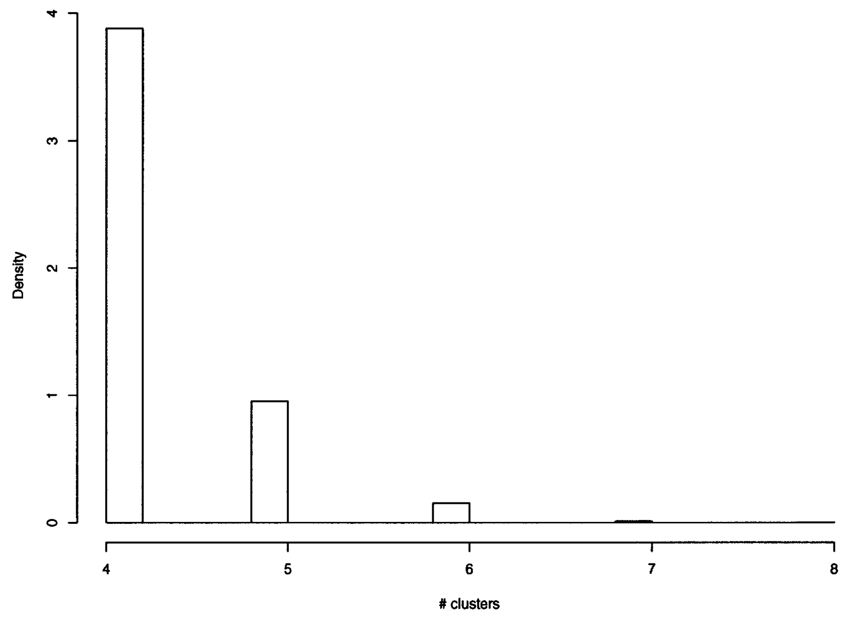

Figure 5-7: Histogram of number of clusters generated using the slope-based model.

A plot of the mean expression profiles for the top 5 clusters corresponding to this heatmap and the resulting dendrogram for the slope model is given in Figure 5-9. with a between-cluster separation value of 6.83, nearly as low as for the more general nested model.

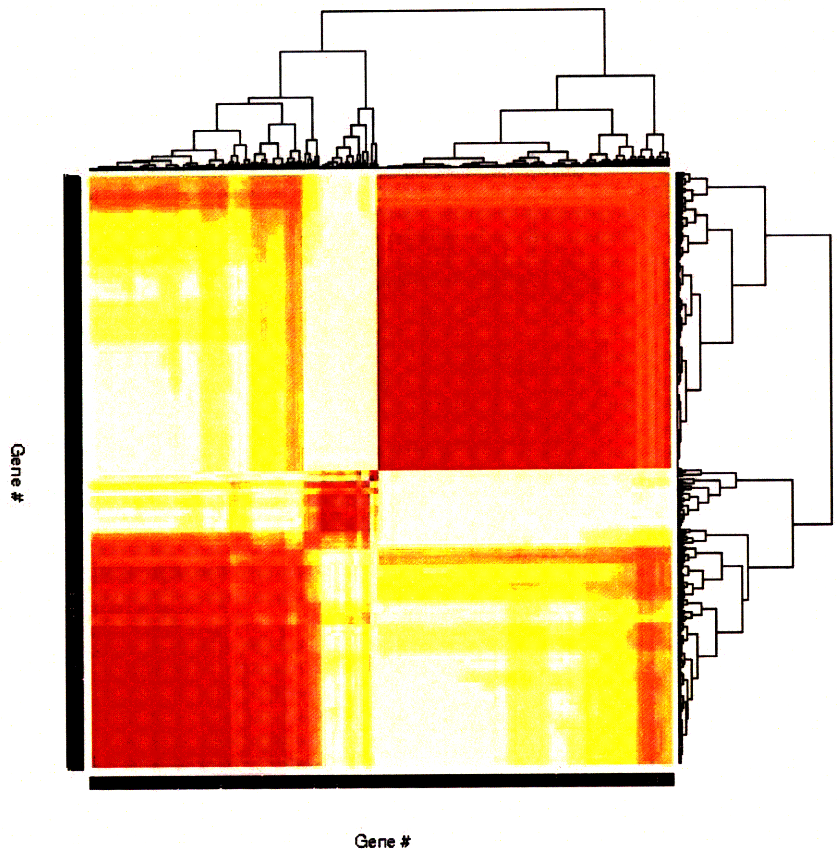

Figure 5-8: Heatmap of clusters of the 798 genes based on the average incidence matrix for the slope-based model. Higher intensity of red denotes more likelihood of clustering together.

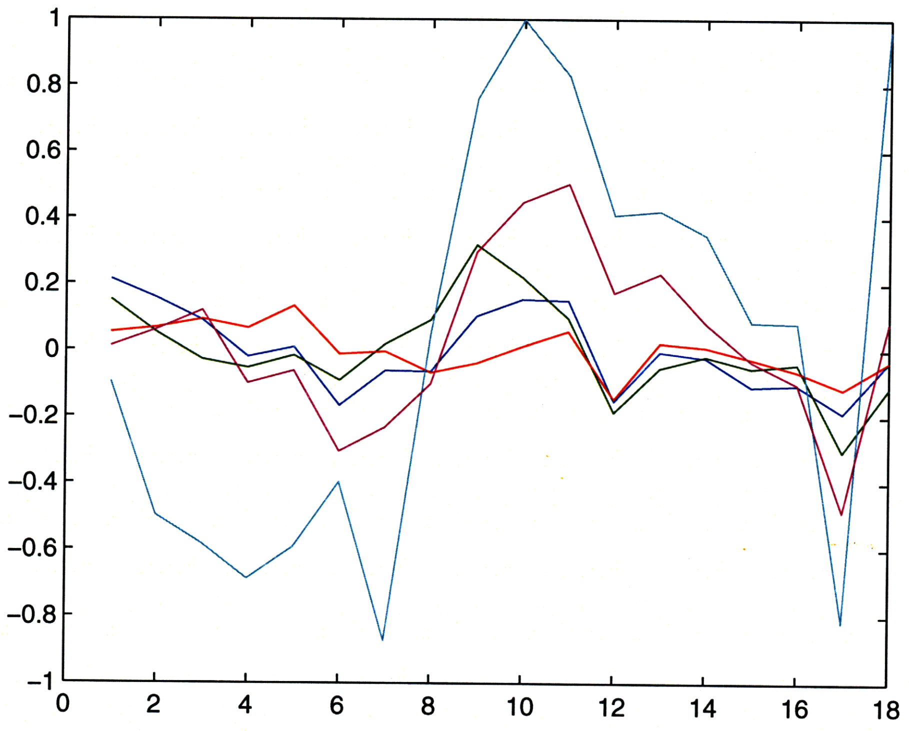

Figure 5-9: Mean curves of top 5 clusters found by the slope-based model.

Chapter 6

Growth Curve Models

In statistics, experiments where multiple measurements are obtained on the same unit are referred to as repeated-measures models. A typical situation is one where the same child is observed at different time-points to note the growth, say in weight or height, similar to what we do in measuring the expression values for the same gene at different times. Such measurements are clearly correlated, so that the observation vectors at say p different time-points can be considered as coming from a p-variate Gaussian. The children might be from different treatment groups and the goal might be compare the "mean" growth curves for the various groups. In our work, we treat gene expression data over different time-points of say, the cell-cycle, as being multivariate Gaussian, to which we fit growth curves of appropriate degree. After that we can establish through statistical tests whether the mean patterns for different groups/clusters are significantly different.

In treating time-series expression data, [3] for instance base their analysis on the coefficients of the fitted splines, as does the approach given in [26]. Here we propose to take a different approach, treating the observed vector on each gene as multivariate Gaussian, and fitting a mean-curve to each group. This so-called "growth curve" analysis, based on linear models for multivariate data, allows us to do proper statistical significance tests for hypotheses of interest.

Remark 1: The polynomial coefficients themselves, being in lower dimension, can be the basis for clustering the original data or as a representation for other purposes as in [3].

Remark 2: In this setup we are back to the situation where we have equal numbers of time points (even if of unequal lengths), although we allow for different numbers of genes for the different groups. Missing data in this setup can be handled by simple imputation or other methods.

Suppose that there are r different groups or clusters, and x denotes the real valued (growth) variable measured at p different time points: $t_1, t_2, ..., t_p$ for n_j individuals chosen at random from the j^{th} group, $(j = 1, \ldots, r)$. We specify the following polynomial regression model of degree $(q-1)$ for the growth x on the time variable t,

$$\begin{aligned} E(x_t) &= \psi_{j0}t^0 + \psi_{j1}t^1 + ... + \psi_{jq-1}t^{q-1}; \\ &\quad (t = t_1, ..., t_p; \quad p > q-1; \quad j = 1, 2, ..., r). \end{aligned} \tag{6.1}$$

Let

$$N = n_1 + n_2 + ... + n_r$$

denote the total number of observations in all the groups put together. Let

$$\psi_j' = [\psi_{j0}\psi_{j1}...\psi_{jq-1}] \tag{6.2}$$

denote the vector of growth curve coefficients for the j^{th} group. Since the observations $x_{t_1}, ..., x_{t_p}$ are on the same item and hence correlated, we denote their variance-covariance matrix by $\mathbf{\Sigma}$. For simplicity and convenience, we assume $\mathbf{\Sigma}$ to be the same for all the r groups.

Let $\mathbf{X}_j$ denote the $p \times n_j$ matrix of the observations for the j^{th} group with each column of dimension p representing one case. Let the $p \times N$ matrix

$$\mathbf{X} = [\mathbf{X}_1 \; \mathbf{X}_2 \; ... \; \mathbf{X}_r]. \tag{6.3}$$

denote the combined set of observations in all the r groups.

Then from Equation (6.1) we get,

$$\begin{aligned} E(\mathbf{X}_j) &= [\mathbf{B}\psi_j \mathbf{B}\psi_j ... \mathbf{B}\psi_j] \\ &= \mathbf{B}\psi_j \mathbf{E}_{1n_j} \quad (j = 1, 2, ..., r), \end{aligned} \tag{6.4}$$

where

$$\mathbf{B} = \begin{bmatrix} t_1^0 & t_1^1 & ... & t_1^{q-1} \\ t_2^0 & t_2^1 & ... & t_2^{q-1} \\ ... & ... & ... & ... \\ t_p^0 & t_p^1 & ... & t_p^{q-1} \end{bmatrix} \tag{6.5}$$

and $\mathbf{E}_{ab}$ denotes, a matrix of order $a \times b$ with all elements equal to 1. $\mathbf{B}_{p \times q}$ is called the "design matrix" and depends on the basis we use to represent the mean function. Let

$$\mathbf{A}_{r \times N} = \text{diag}[\mathbf{E}_{1n_1}, \mathbf{E}_{1n_2}, ..., \mathbf{E}_{1n_r}], \tag{6.6}$$

a block diagonal matrix with $\mathbf{E}_{1n_j} (j = 1, 2, ..., r)$ along the diagonal blocks and zeros elsewhere. From Equation (6.4), we get

$$\begin{aligned} E(\mathbf{X}) &= [\mathbf{B}\psi_1 \mathbf{E}_{1n_1} | \mathbf{B}\psi_2 \mathbf{E}_{1n_2} | ... | \mathbf{B}\psi_r \mathbf{E}_{1n_r}] \\ &= \mathbf{B\Psi A}, \end{aligned} \tag{6.7}$$

where

$$\mathbf{\Psi} = [\psi_1 \; ... \; \psi_r] \tag{6.8}$$

is the $q \times r$ matrix of the growth curve coefficients.

Let Vec$\mathbf{X}$, be defined as the column vector obtained by stacking the columns of $\mathbf{X}$ one below the other. Denoting Var(Vec$\mathbf{X}$) by Var($\mathbf{X}$) we see that,

$$\mathrm{Var}(\mathbf{X}) = \mathbf{I}_N \otimes \mathbf{\Sigma}, \tag{6.9}$$

where $\otimes$ denotes the Kronecker product of two matrices. Equation (6.7) together with Equation (6.9) is called the "growth curve model," and was introduced by [41]. See also [44], [29], or [40] for good expositions.

6.1 Analysis of growth curves

We briefly discuss how to fit growth curve models to time-series gene-expression data and how to test various hypotheses of interest. Two basic tests of interest are: (i) if the given model provides an adequate fit, and if so, (ii) based on this fit, if the data indicate significant differences between the r groups. The first test concerns the null hypothesis

H_0 : the degree $(q-1)$ of the growth curves is adequate,

while the second one as to whether different classes are significantly different can be formulated as the linear hypothesis

$$\begin{aligned} H_1 \;\; &: \;\; \psi_1 = \psi_2 = ... = \psi_r \\ &: \;\; \mathbf{\Psi M} = 0; \end{aligned} \tag{6.10}$$

where,

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 \; \dots & 0 \\ 0 & 1 & 0 \; \dots & 0 \\ . & . & . \; \dots & . \\ -1 & -1 & -1 \; \dots & -1 \end{bmatrix}.$$

We now discuss the details of the test procedures and apply them to gene expression data.

6.2 Fitting Growth Curves

To fit the growth curve model and to test various hypotheses of interest, we need to perform the following computations (see [29] for details). We first obtain a matrix $\mathbf{B}_2$ of order $p \times (p-q)$ such that

$$\mathbf{B}_2'\mathbf{B} = 0, \tag{6.11}$$

where $\mathbf{B}$ is as in Equation (6.5). This is accomplished for instance by choosing $(p-q)$ linearly independent columns of the matrix $(\mathbf{I}_p - \mathbf{B}(\mathbf{B}'\mathbf{B})^{-1}\mathbf{B}')$. Next we compute,

$$\mathbf{S} = \mathbf{Y}(\mathbf{I} - \mathbf{A}'(\mathbf{A}\mathbf{A})^{-1}\mathbf{A})\mathbf{Y}', \tag{6.12}$$

where $\mathbf{A}$ is defined in Equation (6.6). The growth curve coefficient matrix is then obtained as

$$\hat{\psi} = (\mathbf{B}'\mathbf{S}^{-1}\mathbf{B})^{-1}(\mathbf{B}'\mathbf{S}^{-1}\mathbf{Y})\mathbf{A}'(\mathbf{A}\mathbf{A})^{-1}, \tag{6.13}$$

which reduces to

$$(\mathbf{B}'\mathbf{S}^{-1}\mathbf{B})^{-1}(\mathbf{B}'\mathbf{S}^{-1}\mathbf{Y})$$

for the special matrix $\mathbf{A}$ defined in Equation (6.6). We now discuss two basic tests of interest, namely, if the given model provides an adequate fit, and if so, based on this model, if the data indicate significant differences between the r groups. To test the first of these hypotheses,

H_0 : the degree $(q-1)$ provides an adequate fit for the curves,

we need the following Multivariate Analysis of Variance (MANOVA) table (Table 6.1).

Source	d.f.	Dispersion, order $(p-q)$
H_0	r	$\mathbf{H}_0 = \mathbf{B}_2'\mathbf{Y}[\mathbf{A}'(\mathbf{A}(\mathbf{A}\mathbf{A}')^{-1}\mathbf{A}]\mathbf{Y}'\mathbf{B}_2$
Error	$N-r$	$\mathbf{E}_0 = \mathbf{B}_2'\mathbf{S}\mathbf{B}_2$
Total	N	$\mathbf{H}_0 + \mathbf{E}_0 = \mathbf{B}_2'\mathbf{Y}\mathbf{Y}'\mathbf{B}_2$

Table 6.1: MANOVA for Test of Specification

To test H_0 we find the Wilks' $\mathbf{\Lambda}$ statistic defined by

$$\Lambda_0 = \frac{\mid \mathbf{E}_0 \mid}{\mid \mathbf{E}_0 + \mathbf{H}_0 \mid}, \tag{6.14}$$

and calculate the test statistic given in [44]:

$$F_0 = \frac{1-\sqrt{\Lambda_0}}{\sqrt{\Lambda_0}} . \frac{ms - 2\lambda}{d_H d_m}, \tag{6.15}$$

which has an approximate distribution F_{df_1,df_2} with degrees of freedom $df_1 = d_m d_H$, and $df_2 = ms - 2\lambda$. Here $m = N - \frac{d_m + d_H + 1}{2}$, $s = (\frac{(d_m d_H)^2 - 4}{d_m^2 + d_H^2 - 5})^{\frac{1}{2}}$, and $\lambda = (d_m d_H - 2)/4$, with d_H equal to the hypothesis degrees of freedom as given in Table 6.1, and $d_m = p - q$.

Next, to test if the groups are significantly different from each other, i.e. the null hypothesis H_1, we construct the following MANOVA table (Table 6.2).

Source	d.f.	Dispersion, order $(p-q)$
H_1	m	$\mathbf{H}_1 = (\hat{\psi}\mathbf{M})(\mathbf{M}'\mathbf{R}_{11}\mathbf{M})^{-1}(\hat{\psi}\mathbf{M})'$
Error	$N-r-(p-q)$	$\mathbf{E}_1 = (\mathbf{B}'\mathbf{S}^{-1}\mathbf{B})^{-1}$
Total	$N-r-(p-q)+m$	$\mathbf{H}_1 + \mathbf{E}_1$

Table 6.2: MANOVA table for Testing H_1

$\mathbf{R}_{11}$ in Table 6.2 is defined by

$$\mathbf{R}_{11} = (\mathbf{A}\mathbf{A}')^{-1}[\mathbf{I} + \mathbf{A}\mathbf{Y}' \times \left\{\mathbf{S}^{-1}\mathbf{B}\left(\mathbf{B}\mathbf{S}^{-1}\mathbf{B}\right)^{-1}\right\}\mathbf{Y}\mathbf{A}'(\mathbf{A}\mathbf{A}')^{-1}]. \tag{6.16}$$

Based on the values in Table 6.2, we calculate the following Wilks' $\mathbf{\Lambda}$ statistic

$$\mathbf{\Lambda}_1 = \frac{|\,\mathbf{E}_1\,|}{|\,\mathbf{E}_1 + \mathbf{H}_1\,|}, \tag{6.17}$$

and the F-statistic

$$F_1 = \frac{1-\sqrt{\mathbf{\Lambda}_1}}{\sqrt{\mathbf{\Lambda}_1}}.\frac{(d_E - 1)}{d_H}. \tag{6.18}$$

Under the hypothesis H_1, this has an F distribution with $df = 2d_H, 2(d_E - 1)$ where d_H and d_E denote the hypothesis and error degrees of freedom as given in Table 6.2.

6.3 Growth Curve Analysis of Yeast Cell-Cycle Data

In this section we fit a growth curve model to a labeled portion of the yeast cell-cycle data due to [51]. The gene expression values are given at $p = 18$ values. In this data set, we have $r = 5$ groups with $n_1 = 113, n_2 = 299, n_3 = 71, n_4 = 121$, and $n_5 = 194$ for a grand total of $N = 798$ observations. Mean curves were tried with polynomials of varying degrees and the AIC and BIC criteria appear to indicate that a q value of 6 (corresponding to a 5th degree polynomial) provides a good fit (subject to the numerical limitations imposed by the data). The MATLAB code used for this exercise is provided in Appendix A. Figure 6-1 plots the model mean against the observed (sample) average at each time point for Group 5.

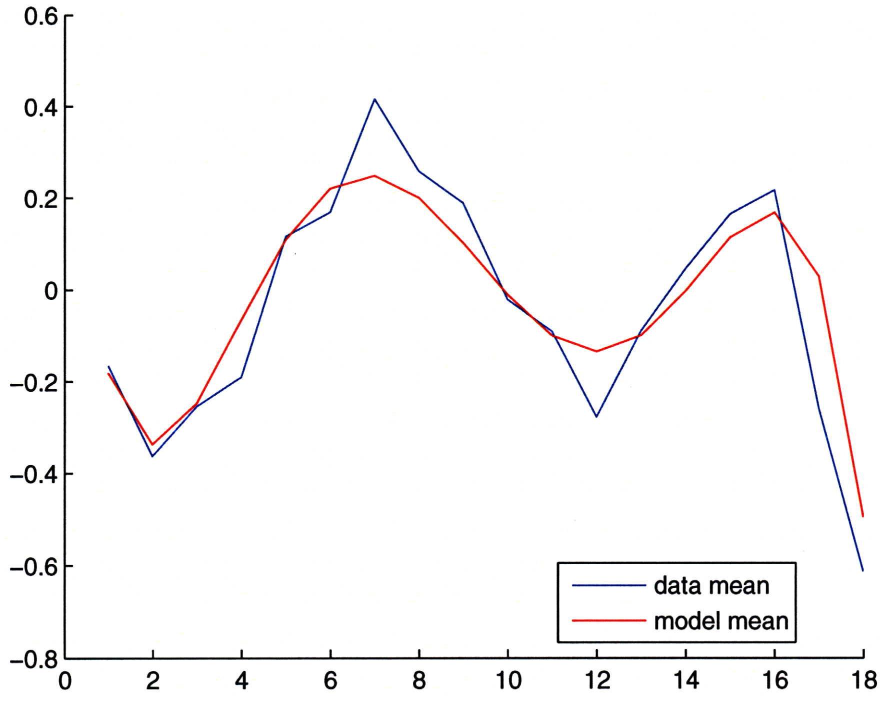

Figure 6-1: Observed Means and Fitted Means with $q = 6$

While this illustrates graphically that the fit is reasonable, we do the test for the model fit, i.e., the hypothesis H_0. This gives $\mathbf{\Lambda}_0 = 0.1518$ (see Equation (6.14)) and a p-value close to zero, signifying a good fit.

Figure 6-2 provides a graphical comparison of the mean curves for the 5 groups.

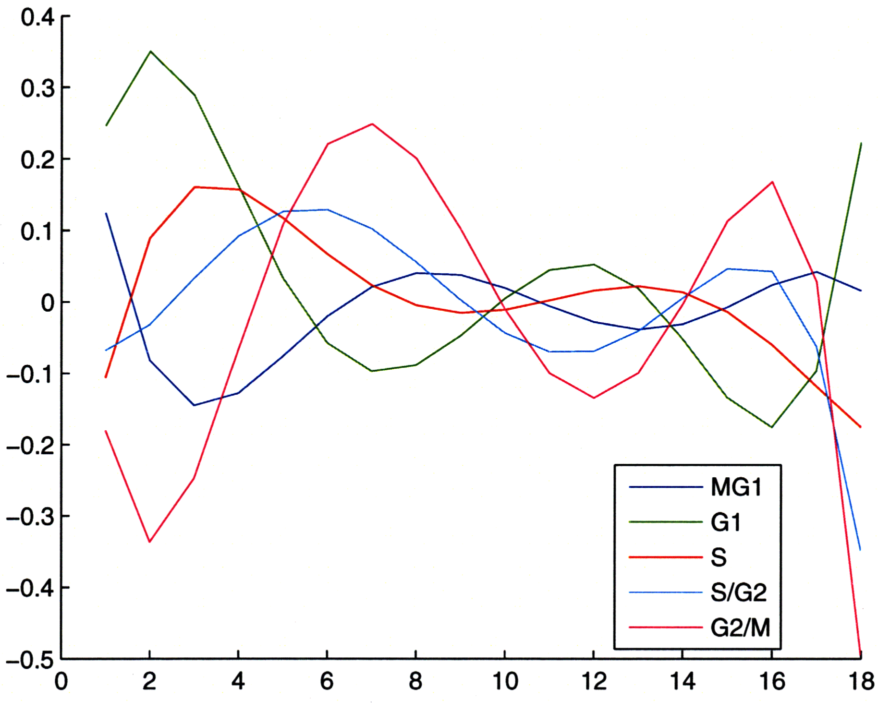

Figure 6-2: Model-based Mean curves for the 5 groups

The statistical test of hypothesis H_1 gives a $\mathbf{\Lambda}_1 = 0.5649$ (see Equation (6.17)) with a p-value again close to zero, indicating that the 5 groups do indeed have significantly different mean curves.

Remark 3: Although we assumed the simpler Rao's structure (see eg. [30]) for the covariance matrix, it is possible to estimate the common covariance matrix Σ of order $p \times p$ (with $\frac{p(p+1)}{2}$ unknowns) if there is a large amount of data. Otherwise, a reasonable alternative approach is to assume that this matrix is a function, say $\Sigma(\rho)$, of a small number of unknowns ρ as we did in Chapter 4, and estimate these. This may happen when, for example, the time-series is from an autoregressive model or the like.

Remark 4: Finally instead of the polynomial fits used in this work, one can employ a basis set of splines (such as cubic splines or B-splines) and it turns out that in this particular case, these approaches yield very similar results.

Chapter 7

Conclusions and Future Work

In this work, we have introduced and developed both the theory and the necessary computational methods for 5 different Bayesian procedures for clustering time-series data. These models were then tested, for illustrative purposes, on a classical time-series gene expression data set, due to [51]. The nonparametric nature of these approaches, and specifically the Dirichlet process used for the mixing distribution on the parameters, allows greater flexibility in model specification and does not suffer from the usual criticism made of Bayes methods, namely their strong dependence on the choice of prior. This property of robustness to choice of the prior, and more importantly the natural way in which the number of clusters is decided by borrowing information from genes with similar properties, i.e. by adapting to the data, are some of the main reasons we explored this approach and its variations in this thesis. In contrast, in typical model-based clustering, the number of clusters (or mixture components) have to be assessed from external inputs and criteria.

7.1 Summary

We now give a brief summary of the different models and see how they compare. The model introduced in Chapter 3 clusters the expression profiles based on their trajectories; that is, their graphs treated as functions of time. The trajectory function corresponding to each gene is assumed to have a random intercept and a random slope, with these slopes allowed to change at each of the measured time-points. To allow for the possibility that different genes (trajectories) may have different numbers and locations of changepoints, we assume that the slope changes have a distribution with point mass at zero. This very general formulation thus includes the range of possibility from having no change in slope at a given time point to having a different slope at every time point. The random parameter vector of the intercept, the slope, and the slope changes is then assumed to have a distribution with a Dirichlet prior.

In Chapter 4 it is assumed that the entire gene-expression profile, i.e. the data vector, has a multivariate normal distribution with a certain mean vector and a covariance matrix with a given structure. This extends past work where covariance structure has not been part of the model. The parameter vector involved is assumed then to come from a distribution on which we place a Dirichlet prior. Two models are considered here, one a slight generalization of the other in the sense of allowing different variance values for each gene.

In Chapter 5 we considered a very recent development in dependent Dirichlet processes, namely the nested Dirichlet process. This approach models the data for each gene with a nonparametric distribution, and clusters those genes which have similar distributional profiles, while at the same time looking for possible clustering of time points "nested" within such gene-clusters. In the later part of this chapter, we consider a specialization of this idea to clustering genes according to their slope distributions, which are assumed to be normal. We show more details here including an analysis of a synthetic data set and the specifics of MCMC burn-in diagnostics, as this is where the model development work began.

Chapter 6 takes our work in a different direction; namely how to statistically validate the distinctness of the clusters that we obtain by any one of our methods, or in general any time-series clustering method. We treat gene expression data over different time-points as being multivariate Gaussian, to which we fit curves of appropriate degree. After that, we provide statistical tests which check whether the mean curves for different clusters are significantly different, using methods based on growth curves.

7.2 Analysis and Discussion

As described in earlier chapters, these models have been individually validated by trying them on synthetic data sets appropriate to each model and verifying that they give expected results. Clearly these models are all built to identify clusters based on certain properties and features of the data, unlike a generic method such as k-means clustering; and of course there is considerable literature demonstrating the superiority of such clustering methods in a wide variety of contexts (e.g. [34], [53]).

There are several ways to see how the clusterings produced by these various models compare in relation to the particular data set we analyzed. Since the entries in the incidence matrix represent the proportion of times two genes are put into the same cluster by a particular model, one gross comparison is obtained by measuring how close two incidence matrices are, that is, by computing the Euclidean distances $\|Q - P\|^2 = \sum_{i,j}(q_{ij} - p_{ij})^2$ between the incidence matrices Q and P produced by any pair of models.

Given that there are 798×797, or nearly 6.4×10^5 entries ij over which this is summed (the diagonal entries always yielding a zero), the distances we computed for this were relatively small and indicate an overall agreement between the first three models. In particular we notice that the homoscedastic and heteroscedastic models are in close agreement with a distance of 0.845×10^4, suggesting that there is not much to be gained in this particular instance by allowing different variances for the different genes.

Another way to compare two clusterings is by looking at the so-called "confusion matrix," a contingency table which counts how the members of a cluster produced by the first clustering method are distributed across various clusters produced by the second method. Still another important class of criteria for comparing 2 clusterings **C** and **C**', is based on counting the pairs of points on which two clusterings agree or disagree (see for instance [19], [38]). If there are n objects being clustered, each of the $n(n-1)/2$ pairs of points will be counted under one of four cases:

1. N_{11}, the number of pairs that are in the same cluster under both **C** and **C**',
2. N_{00}, the number of pairs in different clusters under both **C** and **C**',
3. N_{10}, the number of pairs in the same cluster under **C** but not under **C**', and
4. N_{01}, the number of pairs in the same cluster under **C**' but not under **C**.

There are several metrics based on these numbers which measure how close these 2 clusterings **C** and **C**' are. Two of the prominent ones are the Rand metric ([43]) given by

$$\frac{N_{11}+N_{00}}{n(n-1)/2},$$

and the Jaccard metric given by

$$\frac{N_{11}}{N_{11}+N_{01}+N_{10}}$$

(see [4] for example). The results of calculating these metrics for a single run of our models and data are shown in Tables 7.1 and 7.2.

First, we note that a single run of each model is not sufficient to explore the self-consistency of each model with itself, and additional runs of these probabilistic clusterings for comparison purposes would provide more information. Given this, however, we notice from these tables of similarity metrics that both the Rand and Jaccard metrics point in the same general direction overall in terms of similarity of the various models discussed. Though this data is limited, we might still surmise that

	TM	Hom. PM	Het. PM	NM	SM
Trajectory Model	1.000	0.763	0.639	0.326	0.337
Homoscedastic Profile Model			0.664	0.389	0.492
Heteroscedastic Profile Model				0.439	0.564
Nested Model					0.559
Slope Model					

Table 7.1: Pairwise Rand scores for different models.

	TM	Hom. PM	Het. PM	NM	SM
Trajectory Model	1.000	0.760	0.634	0.316	0.326
Homoscedastic Profile Model			0.611	0.275	0.360
Heteroscedastic Profile Model				0.259	0.374
Nested Model					0.186
Slope Model					

Table 7.2: Pairwise Jaccard scores for different models.

the trajectory model, homoscedastic model, and heteroscedastic model give comparable clusters, implying that (i) there is not a significant difference in trying to cluster the genes according to the slope-changes between time-points, or clustering by their overall profiles for this data (although it should be emphasized that the trajectory model is more generally applicable, including in cases with unequal number of time points and missing data, whereas the other two are not), and (ii) the closeness of clustering provided by the homoscedastic and heteroscedastic models indicates again that introducing heteroscedasticity in the model does not provide additional benefit in clustering this particular data set. Additionally, one can find simulated critical values for each of these metrics under the hypothesis of randomness (as in [54]). In particular, for the Jaccard metric, the similarity score for comparing two random clusterings (of 798 elements into 5 clusters), is 0.11, suggesting that the entries in Table 7.2 are significant.

Finally, as a very rough measure of cluster separation, we also consider the distance achieved by each the models from the overall mean of the data $\bar{x}$, as measured by the sum across 5 clusters and 18 timepoints,

$$\sum_{i=1}^{5}\sum_{t=1}^{18}(x_{it} - \bar{x}_t)^2$$

While these results are not proof of anything, and this may not be the most desirable metric for measuring cluster separation, we can can consider the results of this (in Table 7.3) to be some further evidence that neither the nested model nor the slope model provide good clustering for this particular data, and their clusterings may be somewhat similar. On the other hand this evidence suggests that the model based on trajectories and the two based on profiles probably do a good job of providing clusters that are distinct. As a point of comparison, k-means clustering for 5 clusters provides a maximum distance from the mean of approximately 17 by this metric, or significantly less than our successful models. We also compared to the spline-based method of [3]. The mean curves generated for 5 clusters, with 5 restarts, on the Spellman set are shown in Figure 7-1, and the cluster distance from the mean was 9.7.

Model	Distance
Trajectory	54.85
Homoscedastic	40.13
Heteroscedastic	39.68
Nested	0.36
Slope	6.83

Table 7.3: Distance from mean for different models.

Although clustering gene expression profiles over time has been the main inspiration of our work, we have noted that these models are fully applicable to any time-series data in the appropriate context.

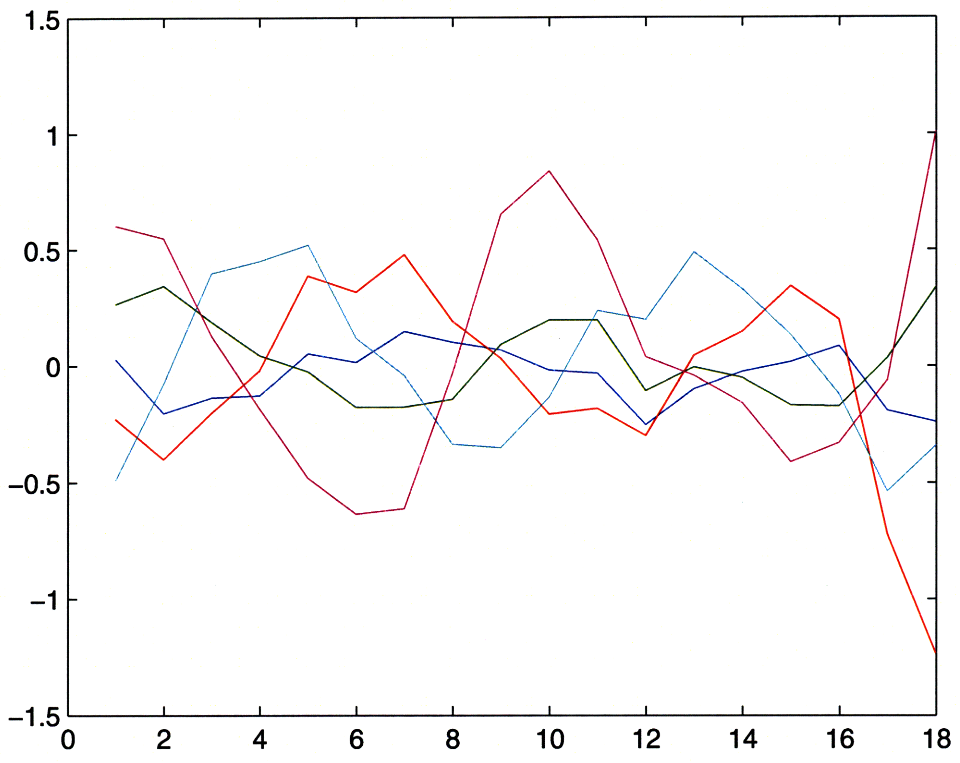

Figure 7-1: Mean curves of 5 clusters found by the spline model of Bar-Joseph, et al.

7.3 Further work and possible extensions

Our focus in this thesis has been to explore various nonparametric Bayes approaches to cluster time-series data, with gene expression data as a motivating example. Clearly this is only the starting point, especially when it comes to the analysis of gene expression experiments. There is significant remaining work to be done to place the models described here in the context of other approaches to the problem of time-series modeling and clustering in general, as well as additional verification work on synthetic and other data to show that all the models are recovering appropriate structure. Comparative work of this form could better elucidate why one might choose to use these models over the alternative options. Beyond that, further work and possible extensions include:

1. Incorporating covariate information. This has been discussed briefly in Chapter 5.3, and can be further developed, including the code to implement it. For example, one might try to take into account the mean expression level of each profile as a covariate to fold-change data, or a multinomial value associated with a chromatin marker. Such additional information could allow for more granular separation of genes into clusters.

2. Prediction of future values, for validation, or otherwise. The theory and code for doing this should be a natural extension of the existing models.

3. With the variable clusters that the Dirichlet process generates, there is need to evaluate the clustering methods we have outlined and compare the results for different data sets.

4. Identifying additional data sets in which clustering based on one of our models is most appropriate and relevant, based on the dynamics in that context.

Appendix A

Code

A.1 Chap. 3

A.1.1 Basic Trajectory Model

```
/*-------------included files----------------------------*/
#include<stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_sort_vector.h>
#include <gsl/gsl_sort_float.h>
#include <gsl/gsl_sort_vector_float.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_cdf.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_permute.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_integration.h>
#include <gsl/gsl_statistics.h>
#include "arms.h"

/*--------------constants---------------------------------*/
#define MATHLIB_STANDALONE 1 /*needed for GSL*/
#define NUM_ITER 20000 /*iterations after burn-in*/
#define NUM_BURN 10000 /*burn-in*/
#define NUM_THIN 1 /*thinning*/
#define PI 3.141592654
#define MAXPOINTS 10   /*max. number of observations per gene*/
#define MAXGENE 60     /*max. number of genes under study*/
#define LMAX 8 /*max number of changepoints allowed per trajectory*/
```

```
/*--------------macros-------------------*/
#define MIN(x, y) ((x) < (y) ? (x)  : (y))
#define MAX(x, y) ((x) > (y) ? (x)  : (y))
#define SPLINE(x) (MAX((x), 0))
#define SQR(x) ((x) * (x))

/*-------special data type declarations---------*/
typedef struct {
/*structure for each measurement -- notes the time
of measurement and the expression level*/
double timept; /*time*/
double express; /*expression*/
} Point;

typedef struct {
/*structure for all the observations on a particular
gene*/
int id; /*gene ID*/
int nobs; /*number of obs.*/
Point dat[MAXPOINTS]; /*timepoints and the corresponding expression levels*/
int config; /*configuration*/
} Indiv;

typedef struct {
/*structure for a unique random effect and its multiplicity*/
gsl_vector *delta; /*vector of intercept and slope*/
gsl_vector *slope; /*vector of changepoint slopes*/
gsl_vector *omega; /*vector of indicators for the changepoint slope mixture distribution*/
int count;
} Unique;

typedef struct {
/*structure for all parameters (including hyperparameters)*/
double prec; /*precision parameter*/
double sigma_sq_e; /*error variance*/
gsl_vector *cp; /*vector of changepoints*/
gsl_vector *sigma_sq_delta; /*vector of variances of the delta components*/
gsl_vector *sigma_sq_slope; /*vector of variances of the changepoint-slope components*/
gsl_vector *p; /*vector of probabilities of getting zero slopes in front of the changepoints*/
gsl_vector *u;
int num_unique; /*number of unique values in the DPP*/
Unique star[MAXGENE]; /*collection of the unique values*/

double prec_shape; /*shape of precision*/
double prec_scale; /*scale of precision*/
double sigma_shape; /*shape of error variance*/
double sigma_scale; /*scale of error variance*/

double delta_shape; /*shape of the variance of the delta's*/
double delta_scale; /*scale of the variances of the delta's*/
double slope_shape; /*shape of the variances of the slopes*/
double slope_scale; /*scale of the variances of the slopes*/
double rho;
gsl_vector *delta_mu;
gsl_vector *slope_mu;
gsl_vector *d0;
gsl_vector *b0;
gsl_vector *sigma_sq_d0;
gsl_vector *sigma_sq_b0;
} Param;


/*------------routine protoypes--------------*/
void read_data(Indiv *, int *);
void print_data(Indiv *, const int);
void get_prior_parameters(Param *);
void initialize(Param *, Indiv *, const int, gsl_rng *);
void update_prec(Param *,  const int, gsl_rng *);
double traj(int, double, gsl_vector *, gsl_vector *, gsl_vector *,  Indiv *);
void update_sigma_sq_e(Param *, int, Indiv *, gsl_rng *);
void update_delta_var(Param *, const int, Indiv *, gsl_rng *);
void update_slope_var(Param *, const int, Indiv *, gsl_rng *);
void update_p(Param *, const int, Indiv *, gsl_rng *);
double log_lik_theta(int, gsl_vector *, gsl_vector *, Param *, Indiv *);
void update_theta_neal(Param *, int, Indiv *, gsl_rng *);
void update_all(Param *, int, Indiv *, gsl_rng *);
double log_lik_indiv(int, Param *, Indiv *, int);
void burn(Param *, Indiv *, int, gsl_rng *);

/*------------routines-----------------------*/
void read_data(Indiv *gene, int *num_gene)
/*read in the data*/
{
FILE *fp;
int i;
int j;
double express;

/* fp=fopen("spelldata.prn", "r");*/
fp=fopen("cook.txt", "r");
*num_gene=MAXGENE;
for(i=0; i < *num_gene; i++)
{
gene[i].id=i;
gene[i].nobs=MAXPOINTS;
gene[i].config=i;
for(j = 0; j < gene[i].nobs; j++)
{
fscanf(fp, "%lf ", &express);
gene[i].dat[j].express=express;
gene[i].dat[j].timept=j;
}
}
fclose(fp);
}

void print_data(Indiv *gene, const int num_gene)
```

```
/* print out the read-in data*/
{
int i, j;

for(i = 0; i < num_gene; i++)
{
printf("id=%d ", gene[i].id);
for(j = 0; j < gene[i].nobs; j++)
printf("(t=%f,  y=%f) ", gene[i].dat[j].timept, gene[i].dat[j].express);
printf("\n\n");
}
}

void get_prior_parameters(Param *theta)
/*get the prior parameters*/
{
/* printf("getting prior parameters...");*/
/*mean is shape*scale, variance is shape*scale^2*/
theta->prec_shape = 1.0;
theta->prec_scale = 1.0;

/*prior mean of sigma_e is 1/(scale*(shape-1)), prior variance is 1/((shape-1)^2*(shape-2)*scale^
theta->sigma_shape = 1.0; /*2.1;*/
theta->sigma_scale = 1.0; /*1.0/1.1;*/

theta->delta_shape=100.0;
theta->delta_scale=0.1;

theta->slope_shape=100.0;
theta->slope_scale=0.1;

theta->rho=0.50; /*prior probability of no change at a candidate point*/

theta->d0=gsl_vector_calloc(2);
theta->sigma_sq_d0=gsl_vector_alloc(2);
gsl_vector_set_all(theta->sigma_sq_d0, 10.0);

theta->b0=gsl_vector_calloc(LMAX);
theta->sigma_sq_b0=gsl_vector_alloc(LMAX);
gsl_vector_set_all(theta->sigma_sq_b0, 10.0);

/* printf("done\n");*/
}


void initialize(Param *theta, Indiv *gene, const int num_gene, gsl_rng *r)
/*gets starting values of the parameters*/
{
int i, j, l;
int temp;
int start_type;

/* printf("initializing the chain...");*/
```

```
/*get precision*/
theta->prec = gsl_ran_gamma(r, theta->prec_shape, theta->prec_scale);

/*get sigma_sq_e*/
theta->sigma_sq_e = 1.0 / gsl_ran_gamma(r, theta->sigma_shape, theta->sigma_scale);

/*get sigma_sq_delta*/
theta->sigma_sq_delta=gsl_vector_alloc(2);
for(j=0; j<2; j++)
gsl_vector_set(theta->sigma_sq_delta, j, 1.0/gsl_ran_gamma(r, theta->delta_shape, theta->delta_sc

/*get sigma_sq_slope*/
theta->sigma_sq_slope=gsl_vector_alloc(LMAX);
for(j=0; j<LMAX; j++)
gsl_vector_set(theta->sigma_sq_slope, j, 1.0/gsl_ran_gamma(r, theta->slope_shape, theta->slope_sc

/*get p's*/
theta->p=gsl_vector_alloc(LMAX);
theta->u = gsl_vector_alloc(LMAX);
for(j=0; j<LMAX; j++)
{
temp=gsl_ran_bernoulli(r, theta->rho);
if(temp==1) gsl_vector_set(theta->p, j, 0.0); else gsl_vector_set(theta->p, j, gsl_rng_uniform(r)
gsl_vector_set(theta->u, j, temp);
}

/*get changepoint vector*/
theta->cp=gsl_vector_alloc(LMAX);
for(j=0; j<LMAX; j++)
gsl_vector_set(theta->cp, j, j+1);

/*get the baseline means of deltas*/
theta->delta_mu = gsl_vector_alloc(2);
for(j = 0; j < 2; j++)
{
gsl_vector_set(theta->delta_mu, j, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma_sq_d0, j)
}

/*get the baseline means of slopes*/
theta->slope_mu = gsl_vector_alloc(LMAX);
for(j = 0; j < LMAX; j++)
{
gsl_vector_set(theta->slope_mu, j, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma_sq_b0, j)
}


/*get the distinct values and their multiplicities*/
start_type=gsl_ran_bernoulli(r, 0.5); /*randomly choose 1 cluster or num_gene clusters*/
if(start_type == 0)
/*1 cluster*/
{
(theta->num_unique) = 1;
theta->star[0].count = num_gene;
theta->star[0].delta = gsl_vector_alloc(2);
```

```
theta->star[0].slope=gsl_vector_alloc(LMAX);
theta->star[0].omega=gsl_vector_alloc(LMAX);
for(l=0; l<2; l++)
gsl_vector_set(theta->star[0].delta, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma_sq_c
for(l=0; l<LMAX; l++)
{
temp=gsl_ran_bernoulli(r, gsl_vector_get(theta->p, l));
if(temp==1) gsl_vector_set(theta->star[0].slope, l, 0.0);
else gsl_vector_set(theta->star[0].slope, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma
gsl_vector_set(theta->star[0].omega, l, temp);
}
}
else
/*each observation is in its own cluster*/
{
(theta->num_unique) = num_gene;
for(j = 0; j < (theta->num_unique); j++)
{
theta->star[j].count = 1;
theta->star[j].delta = gsl_vector_alloc(2);
theta->star[j].slope=gsl_vector_alloc(LMAX);
theta->star[j].omega=gsl_vector_alloc(LMAX);
for(l=0; l<2; l++)
gsl_vector_set(theta->star[j].delta, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma_sq_c
for(l=0; l<LMAX; l++)
{
temp=gsl_ran_bernoulli(r, gsl_vector_get(theta->p, l));
if(temp==1) gsl_vector_set(theta->star[j].slope, l, 0.0);
else gsl_vector_set(theta->star[j].slope, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma
gsl_vector_set(theta->star[j].omega, l, temp);
}
}
}

for(i = 0; i < num_gene; i++)
{
/*get the configuration*/
if(start_type == 0) gene[i].config = 0; else gene[i].config = i;
}
/* printf("done\n");*/
}


void update_prec(Param *theta,  const int num_gene, gsl_rng *r)
/*updates the precision parameter*/
{
double temp;
double mix_prob;
unsigned int indicator;

/* printf("updating precision parameter...");*/
temp = gsl_ran_beta(r, theta->prec + 1, num_gene);
mix_prob = (theta->prec_shape + theta->num_unique - 1) / (num_gene * (1.0 / theta->prec_scale - l
mix_prob = mix_prob / (1 + mix_prob);
```

```
indicator = gsl_ran_bernoulli(r, mix_prob);
if(indicator == 1)
theta->prec = gsl_ran_gamma(r, theta->prec_shape + theta->num_unique, 1.0 / (1.0 / theta->prec_sc
else
theta->prec = gsl_ran_gamma(r, theta->prec_shape + theta->num_unique - 1, 1.0 / (1.0 / theta->pre
/* printf("done\n");*/
}


double traj(int i, double t, gsl_vector *cp, gsl_vector *delta, gsl_vector *slope,  Indiv *gene)
/*trajectory of the ith patient at time t, cp, delta, slope*/
{
gsl_vector *z2vec = gsl_vector_calloc(2);
gsl_vector *z3vec = gsl_vector_calloc(LMAX);
int j;
double temp,  temp2, temp3;

gsl_vector_set(z2vec, 0, 1.0);
gsl_vector_set(z2vec, 1, t);

for(j = 0; j < LMAX; j++)
{
temp = SPLINE(t - gsl_vector_get(cp, j));
gsl_vector_set(z3vec, j, temp);
}

gsl_blas_ddot(z2vec, delta, &temp2);
gsl_blas_ddot(z3vec, slope, &temp3);

if(isnan(temp2) || isnan(temp3))
{
printf("t=%f ", t);
printf("NAN in traj: temp2=%f, temp3=%f ", temp2, temp3);
printf("\n");
exit(1);
}

gsl_vector_free(z2vec);
gsl_vector_free(z3vec);

return(temp2 + temp3);
}


void update_sigma_sq_e(Param *theta, int num_gene, Indiv *gene, gsl_rng *r)
/*updates the error variance*/
{
int i, j,  ni;
double sum1 = 0.0, sum2 = 0.0, temp;
double new_shape, new_scale;

/* printf("updating the error variance...");*/
for(i = 0; i < num_gene; i++)
{
```

```
ni = gene[i].nobs;
sum1 += ni;
for(j = 0; j < ni; j++)
{
temp = gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, theta->cp,
theta->star[gene[i].config].delta, theta->star[gene[i].config].slope, gene);
sum2 += SQR(temp);
}
}
new_shape = theta->sigma_shape + sum1 / 2.0;
new_scale = 1.0 / (1.0 / theta->sigma_scale + sum2 / 2.0);
theta->sigma_sq_e = 1.0 / gsl_ran_gamma(r, new_shape, new_scale);
/* printf("done\n");*/
}

void update_delta_var(Param *theta, const int num_gene, Indiv *gene, gsl_rng *r)
/*updating the variances of the delta's*/
{
int i, l;
double sum1, sum2;

/* printf("updating sigma_sq_delta...");*/
sum1=theta->delta_shape + theta->num_unique/2.0;
for(l=0; l<2; l++)
{
sum2=0.0;
for(i=0; i < theta->num_unique; i++)
{
sum2 += SQR(gsl_vector_get(theta->star[i].delta, l) - gsl_vector_get(theta->delta_mu, l));
}
sum2 = 1.0/(1.0/theta->delta_scale + sum2/2.0);
gsl_vector_set(theta->sigma_sq_delta, l, 1.0/gsl_ran_gamma(r, sum1, sum2));
}
/* printf("done\n");*/
}

void update_slope_var(Param *theta, const int num_gene, Indiv *gene, gsl_rng *r)
/*updating the variances of the slopes*/
{
int i, l;
double sum1, sum2;

/* printf("updating sigma_sq_slope...");*/
for(l=0; l<LMAX; l++)
{
sum1=0.0;
sum2=0.0;
for(i=0; i < theta->num_unique; i++)
{
sum1 += (1.0 - gsl_vector_get(theta->star[i].omega, l));
sum2 += (1.0 - gsl_vector_get(theta->star[i].omega, l)) * SQR(gsl_vector_get(theta->star[i].slope
}
sum1 = theta->slope_shape + sum1/2.0;
sum2 = 1.0/(1.0/theta->slope_scale + sum2/2.0);
```

```
gsl_vector_set(theta->sigma_sq_slope, l, 1.0/gsl_ran_gamma(r, sum1, sum2));
}
/* printf("done\n");*/
}

void update_p(Param *theta, const int num_gene, Indiv *gene, gsl_rng *r)
/*updating the p's*/
{
int i, l;
double sum;

/* printf("updating p's...");*/
for(l=0; l<LMAX; l++)
{
if(gsl_vector_get(theta->u, l) == 1) gsl_vector_set(theta->p, l, 0.0);
else
{
sum=0.0;
for(i=0; i < theta->num_unique; i++)
sum += gsl_vector_get(theta->star[i].omega, l);
gsl_vector_set(theta->p, l, gsl_ran_beta(r, sum + 1, theta->num_unique - sum +1));
}
}
/* printf("done\n");*/
}

void update_u(Param *theta, int num_gene, Indiv *gene, gsl_rng *r)
/*update u's*/
{
int l;

/*printf("updating u's...");*/
for(l=0; l<LMAX; l++)
{
if(gsl_vector_get(theta->p, l) == 0)
{
gsl_vector_set(theta->u, l, gsl_ran_bernoulli(r, theta->rho));
}
else gsl_vector_set(theta->u, l, 0.0);
}
/*printf("done\n");*/
}


double log_lik_theta(int i, gsl_vector *delta, gsl_vector *slope, Param *theta, Indiv *gene)
/*log-likelihood contribution of ith individual at theta=(delta, slope)*/
{
int j;
int ni = gene[i].nobs;
double temp;
double temp3=0.0;

for(j=0; j<ni; j++)
{
```

```
temp=gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, theta->cp, delta, slope, gene);
temp3 +=SQR(temp);
}
temp3 /= (-2.0*theta->sigma_sq_e);

return(temp3);
}


void update_theta_neal(Param *theta, int num_gene, Indiv *gene, gsl_rng *r)
/*Neal's algorithm 8 to update the configuration vector and the distinct parameter values for  DF
{
int i;
int k;
int ci;
int k_minus;
int h;
int extra=5;
gsl_vector *tempslope=gsl_vector_alloc(LMAX);
gsl_vector *tempdelta=gsl_vector_alloc(2);
gsl_vector *tempomega=gsl_vector_alloc(LMAX);
int tempcount;
int j;
int l;
double probvec[num_gene + extra];
gsl_ran_discrete_t *aa;
int idx;
int c;
int temp;
double prob;
double maxprob;
double sum1, sum2;


/*printf("updating the configuration..");*/
for(i = 0; i<num_gene; i++)
{
/*get the current configuration*/
ci = gene[i].config;
k = theta->num_unique;

(theta->star[ci].count)--;

if(theta->star[ci].count == 0)
/*occurs only once, hence relabel*/
{
/*begin relabeling*/
k_minus = k - 1;
h = k_minus + extra;
gsl_vector_memcpy(tempslope, theta->star[ci].slope);
gsl_vector_memcpy(tempdelta, theta->star[ci].delta);
gsl_vector_memcpy(tempomega, theta->star[ci].omega);
tempcount = theta->star[ci].count;
for(j = ci; j < k_minus; j++)
```

```
{
gsl_vector_swap(theta->star[j].slope, theta->star[j + 1].slope);
gsl_vector_swap(theta->star[j].delta, theta->star[j + 1].delta);
gsl_vector_swap(theta->star[j].omega, theta->star[j + 1].omega);
theta->star[j].count = theta->star[j + 1].count;
}
gsl_vector_swap(theta->star[k_minus].slope, tempslope);
gsl_vector_swap(theta->star[k_minus].delta, tempdelta);
gsl_vector_swap(theta->star[k_minus].omega, tempomega);
theta->star[k_minus].count = tempcount;

for(j = 0; j < num_gene; j++)
{
if(gene[j].config > ci) gene[j].config -= 1;
}
gene[i].config = k - 1;
/*end relabeling*/

/*augment the set of distinct quantities by m additional values*/
for(j = k; j < h; j++)
{
theta->star[j].slope = gsl_vector_alloc(LMAX);
theta->star[j].omega = gsl_vector_alloc(LMAX);
theta->star[j].delta = gsl_vector_alloc(2);
for(l=0; l<2; l++)
gsl_vector_set(theta->star[j].delta, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma_sq_c
for(l=0; l<LMAX; l++)
{
temp=gsl_ran_bernoulli(r, gsl_vector_get(theta->p, l));
if(temp==1) gsl_vector_set(theta->star[j].slope, l, 0.0);
else gsl_vector_set(theta->star[j].slope, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma
gsl_vector_set(theta->star[j].omega, l, temp);
}
}

/*draw a new value of ci for the ith element*/
for(j = 0; j < k_minus; j++)
{
probvec[j] = log_lik_theta(i, theta->star[j].delta, theta->star[j].slope, theta, gene) + log(thet
}
for(j = k_minus; j < h; j++)
{
probvec[j] = log_lik_theta(i, theta->star[j].delta, theta->star[j].slope, theta, gene) + log(thet
}

maxprob = probvec[0];
for(j = 1; j < h; j++)
maxprob = MAX(maxprob, probvec[j]);
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j] - maxprob);
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
```

```
gsl_ran_discrete_free(aa);

if(idx >= (k - 1)) /*a new value is drawn*/
{
gsl_vector_memcpy(theta->star[k - 1].slope, theta->star[idx].slope);
gsl_vector_memcpy(theta->star[k - 1].delta, theta->star[idx].delta);
gsl_vector_memcpy(theta->star[k - 1].omega, theta->star[idx].omega);
gene[i].config = k - 1;
theta->star[gene[i].config].count = 1;
for(j = k; j < h; j++)
{
gsl_vector_free(theta->star[j].slope);
gsl_vector_free(theta->star[j].delta);
gsl_vector_free(theta->star[j].omega);
}
}
else /*existing value is drawn*/
{
gene[i].config = idx;
theta->star[idx].count++;
for(j = (k - 1); j < h; j++)
{
gsl_vector_free(theta->star[j].slope);
gsl_vector_free(theta->star[j].delta);
gsl_vector_free(theta->star[j].omega);
}
theta->num_unique--;
}
}
else if(theta->star[ci].count > 0)/*do not relabel*/
{
k_minus = k;
h = k_minus + extra;
for(j = k; j < h; j++)
{
theta->star[j].slope = gsl_vector_alloc(LMAX);
theta->star[j].omega = gsl_vector_alloc(LMAX);
theta->star[j].delta = gsl_vector_alloc(2);
for(l=0; l<2; l++)
gsl_vector_set(theta->star[j].delta, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma_sq_c
for(l=0; l<LMAX; l++)
{
temp=gsl_ran_bernoulli(r, gsl_vector_get(theta->p, l));
if(temp==1) gsl_vector_set(theta->star[j].slope, l, 0.0);
else gsl_vector_set(theta->star[j].slope, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(theta->sigma
gsl_vector_set(theta->star[j].omega, l, temp);
}
}

for(j = 0; j < k; j++)
{
probvec[j] = log_lik_theta(i, theta->star[j].delta, theta->star[j].slope, theta, gene) + log(thet
}
for(j = k; j < h; j++)
```

```
{
probvec[j] = log_lik_theta(i, theta->star[j].delta, theta->star[j].slope, theta, gene) + log(thet
}

maxprob = probvec[0];
for(j = 1; j < h; j++)
maxprob = MAX(maxprob, probvec[j]);
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j] -maxprob);
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

if(idx > (k - 1))
{
gsl_vector_memcpy(theta->star[k].delta, theta->star[idx].delta);
gsl_vector_memcpy(theta->star[k].slope, theta->star[idx].slope);
gsl_vector_memcpy(theta->star[k].omega, theta->star[idx].omega);
gene[i].config = k;
theta->star[gene[i].config].count = 1;
(theta->num_unique)++;
for(j = (k + 1); j < h; j++)
{
gsl_vector_free(theta->star[j].slope);
gsl_vector_free(theta->star[j].delta);
gsl_vector_free(theta->star[j].omega);
}
}
else
{
gene[i].config = idx;
theta->star[idx].count++;
for(j = k; j < h; j++)
{
gsl_vector_free(theta->star[j].slope);
gsl_vector_free(theta->star[j].delta);
gsl_vector_free(theta->star[j].omega);
}
}
}
}
gsl_vector_free(tempslope);
gsl_vector_free(tempdelta);
gsl_vector_free(tempomega);
/*printf("done\n");*/

/*printf("updating the distinct values...");*/


for(c=0; c<(theta->num_unique); c++)
/*update the cth distinct set of slope and delta*/
{
```

```
for(l=0; l< 2; l++)
/*update the lth component of delta*/
{
sum1=0.0;
sum2=0.0;

for(i=0; i<num_gene; i++)
{
if(gene[i].config==c)
{
for(j=0; j<gene[i].nobs; j++)
{
if(l==1)
{
sum1 += gene[i].dat[j].timept *
(gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, theta->cp,  theta->star[c].delta, theta-
gsl_vector_get(theta->star[c].delta, l) * gene[i].dat[j].timept);
sum2 += SQR(gene[i].dat[j].timept);
}
else
{
sum1 += gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, theta->cp,  theta->star[c].delta,
gsl_vector_get(theta->star[c].delta, 0);
sum2 += 1.0;
}
}
}
}
sum1 = sum1 / theta->sigma_sq_e + gsl_vector_get(theta->delta_mu, l) / gsl_vector_get(theta->sigm
sum2 = sum2 / theta->sigma_sq_e + 1 / gsl_vector_get(theta->sigma_sq_delta, l);
gsl_vector_set(theta->star[c].delta, l, gsl_ran_gaussian(r, sqrt(1/sum2)) + sum1/sum2);
}

for(l=0; l < LMAX; l++)
/*update the lth component of slope*/
{
if(gsl_vector_get(theta->star[c].omega, l)==1) gsl_vector_set(theta->star[c].slope, l, 0.0);
else
{
sum1=0.0;
sum2=0.0;
for(i=0; i<num_gene; i++)
{
if(gene[i].config==c)
{
for(j=0; j<gene[i].nobs; j++)
{
sum1 += SPLINE(gene[i].dat[j].timept - gsl_vector_get(theta->cp, l)) *
(gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, theta->cp,  theta->star[c].delta, theta-
gsl_vector_get(theta->star[c].slope, l) * (SPLINE(gene[i].dat[j].timept - gsl_vector_get(theta->
sum2 += SQR(SPLINE(gene[i].dat[j].timept - gsl_vector_get(theta->cp, l)));
}
}
}
```

```
sum1 = sum1 / theta->sigma_sq_e + gsl_vector_get(theta->slope_mu, l)/gsl_vector_get(theta->sigma_
sum2 = sum2 / theta->sigma_sq_e + 1 / gsl_vector_get(theta->sigma_sq_slope, l);
gsl_vector_set(theta->star[c].slope, l, gsl_ran_gaussian(r, sqrt(1/sum2)) + sum1/sum2);
}

/*update the corresponding omega*/
if(gsl_vector_get(theta->star[c].slope, l) != 0) gsl_vector_set(theta->star[c].omega, l, 0.0);
else
{
prob = gsl_vector_get(theta->p, l);
prob = prob / (prob + (1 - prob) * exp(-0.5 / gsl_vector_get(theta->sigma_sq_slope, l) * SQR(gsl_
gsl_vector_set(theta->star[c].omega, l, gsl_ran_bernoulli(r, prob));
}
}
}
/* printf("done\n");*/
}



void update_delta_mu(Param *theta, int num_gene, Indiv *gene, gsl_rng *r)
{
int i;
int l;
double new_mean, new_var, temp;

for(l = 0; l < 2; l++)
{
new_mean=0.0;
for(i = 0; i< theta->num_unique; i++)
{
new_mean += gsl_vector_get(theta->star[i].delta, l);
}
new_mean = new_mean / gsl_vector_get(theta->sigma_sq_delta, l) + gsl_vector_get(theta->d0, l) / g

new_var = 1.0 / (theta->num_unique/gsl_vector_get(theta->sigma_sq_delta, l) + 1.0 / gsl_vector_ge
new_mean = new_mean * new_var;

temp = gsl_ran_gaussian(r, sqrt(new_var)) + new_mean;
gsl_vector_set(theta->delta_mu, l, temp);
}
}

void update_slope_mu(Param *theta, int num_gene, Indiv *gene, gsl_rng *r)
{
int i;
int l;
double new_mean, new_var, temp;

for(l = 0; l < LMAX; l++)
{
new_mean = 0.0;
new_var = 0.0;
for(i = 0; i < theta->num_unique; i++)
```

```
{
new_mean += gsl_vector_get(theta->star[i].slope, l) * (1 - gsl_vector_get(theta->star[i].omega, l
new_var += (1 - gsl_vector_get(theta->star[i].omega, l));
}
new_mean = new_mean / gsl_vector_get(theta->sigma_sq_slope, l) + gsl_vector_get(theta->b0, l) / g
new_var = 1.0 / (new_var/gsl_vector_get(theta->sigma_sq_slope, l) + 1.0 / gsl_vector_get(theta->s
new_mean = new_mean * new_var;
temp = gsl_ran_gaussian(r, sqrt(new_var)) + new_mean;
gsl_vector_set(theta->slope_mu, l, temp);
}
}



void update_all(Param *theta, int num_gene, Indiv *gene, gsl_rng *r)
/*updates all the parameters once*/
{
update_prec(theta,  num_gene, r);
update_sigma_sq_e(theta, num_gene, gene,  r);
update_delta_var(theta, num_gene, gene, r);
update_slope_var(theta, num_gene, gene, r);
update_p(theta, num_gene, gene, r);
update_u(theta, num_gene, gene, r);
update_theta_neal(theta, num_gene, gene, r);
update_delta_mu(theta, num_gene, gene, r);
update_slope_mu(theta, num_gene, gene, r);
}

double log_lik_indiv(int i, Param *theta, Indiv *gene, int num_gene)
/*log-likelihood of ith gene*/
{
double sum=0.0;
int j;

for(j=0; j<gene[i].nobs; j++)
{
sum -= SQR(gene[i].dat[j].express-traj(i, gene[i].dat[j].timept, theta->cp,  theta->star[gene[i].
}
sum /= (2*theta->sigma_sq_e);
sum -= gene[i].nobs/2 * log(theta->sigma_sq_e);

return(sum);
}

void burn(Param *theta, Indiv *gene, int num_gene, gsl_rng *r)
/*burns the sampler and prints out parameters of interest (or their functons)*/
{
int l;
int burn;
double log_lik;
int i;

for(burn=0; burn<NUM_BURN; burn++)
/*burn the Gibbs sampler, while monitoring certain quantities of interest*/
```

```
{
if((burn % NUM_THIN) == 0)
{

printf("%d ", theta->num_unique);
printf("%f ", theta->prec);
printf("%f ", theta->sigma_sq_e);

for(l = 0; l < 2; l++)
printf("%.4f ", gsl_vector_get(theta->sigma_sq_delta, l));
for(l = 0; l < LMAX; l++)
printf("%.4f ", gsl_vector_get(theta->sigma_sq_slope, l));

for(l = 0; l < 2; l++)
printf("%.4f ", gsl_vector_get(theta->delta_mu, l));

for(l = 0; l < LMAX; l++)
printf("%.4f ", gsl_vector_get(theta->slope_mu, l));
for(l = 0; l < LMAX; l++)
printf("%.4f ", gsl_vector_get(theta->p, l));

log_lik=0.0;
for(i=0; i<num_gene; i++)
{
log_lik +=  log_lik_indiv(i, theta, gene, num_gene);
}
printf("%.4f\n", log_lik);
}
update_all(theta, num_gene, gene, r);
}
}


/*-----------------main program------------------*/
int main(void)
{
int num_gene;
Indiv gene[MAXGENE];
Param theta;
const gsl_rng_type *T;
gsl_rng *r;
int i, l, count, iter;
double incid_mat[MAXGENE][MAXGENE];
double  log_lik;

printf("NUM_ITER=%d, NUM_BURN=%d, NUM_THIN=%d, LMAX=%d\n", NUM_ITER, NUM_BURN, NUM_THIN, LMAX);
read_data(gene, &num_gene);
/* print_data(gene, num_gene);*/
get_prior_parameters(&theta);

/*start the random number generator*/
gsl_rng_env_setup();
T = gsl_rng_default;
r = gsl_rng_alloc(T);
```

```
/*initialize all the parameters*/
initialize(&theta, gene, num_gene, r);

burn(&theta, gene, num_gene, r);

/* burn the sampler for some time, until satisfied with its convergence*/
/* do
{
burn(&theta, gene, num_gene, r);
printf("burn more? (y/n)");
} while ((c=getchar())!='n');
*/

/*initialize the incidence matrix*/
for(i=0; i<num_gene; i++)
{
for(l=(i+1); l<num_gene; l++)
{
incid_mat[i][l]=0.0;
}
}

/*generate more iterations from the sampler after it has converged.
use them to get the incidence matrix*/
count=0;
for(iter=0; iter<NUM_ITER; iter++)
{
update_all(&theta, num_gene, gene, r);
if(iter % NUM_THIN == 0)
{

printf("%d ", theta.num_unique);
printf("%f ", theta.prec);
printf("%f ", theta.sigma_sq_e);

for(l = 0; l < 2; l++)
{
printf("%.4f ", gsl_vector_get(theta.sigma_sq_delta, l));
}
for(l = 0; l < LMAX; l++)
{
printf("%.4f ", gsl_vector_get(theta.sigma_sq_slope, l));
}

for(l = 0; l < 2; l++)
{
printf("%.4f ", gsl_vector_get(theta.delta_mu, l));
}

for(l = 0; l < LMAX; l++)
{
printf("%.4f ", gsl_vector_get(theta.slope_mu, l));
}
for(l = 0; l < LMAX; l++)
{
printf("%.4f ", gsl_vector_get(theta.p, l));
}


log_lik=0.0;
for(i=0; i<num_gene; i++)
{
log_lik +=  log_lik_indiv(i, &theta,  gene, num_gene);
}
printf("%.4f\n", log_lik);

count++;
for(i=0; i<num_gene; i++)
{
for(l=(i+1); l<num_gene; l++)
{
if(gene[i].config==gene[l].config)
incid_mat[i][l]++;
}
}
}
}
gsl_rng_free(r);
printf("\n");

/*print out the relative clustering frequencies to generate the heatmap*/
printf("heatmap data\n");
for(i=0; i<num_gene; i++)
{
for(l=0; l<num_gene; l++)
{
if(l>i)
printf("%.4f ", incid_mat[i][l]/count);
else
{
if(l<i)
{
printf("%.4f ", incid_mat[l][i]/count);
}
else
{
printf("%.4f ", 1.00);
}
}
}
printf("\n");
}
return 0;
}
```

A.1.2 General Trajectory Model

```
/*---------------included files------------------*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_sort_vector.h>
#include <gsl/gsl_sort_float.h>
#include <gsl/gsl_sort_vector_float.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_cdf.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_permute.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_statistics.h>
#include <gsl/gsl_sf_gamma.h>
#include "arms.h"

/*-------------constants----------------*/
#define MATHLIB_STANDALONE 1
#define NUM_ITER 20000 /*iterations to sample after burn-in*/
#define NUM_BURN 10000 /*burn-in until convergence*/
#define NUM_THIN 1 /*thinning*/
#define PI 3.141592654
#define MAXPOINTS 10   /*max. number of observations per subject*/
#define MAXGENE 60     /*max. number of genes under study*/
#define LMAX 8 /*max number of changepoints allowed per trajectory*/
#define MAXDF 50 /*maximum df of the inv-chi^2 distribution*/


/*--------------macros-------------------*/
#define MIN(x, y) ((x) < (y) ? (x)  : (y))
#define MAX(x, y) ((x) > (y) ? (x)  : (y))
#define SPLINE(x) (MAX((x), 0))
#define SQR(x) ((x) * (x))

/*-------special data type declarations---------*/

typedef struct {
/*structure for each measurement -- notes the time
of measurement and the expression level*/
double timept; /*time*/
double express; /*expression*/
} Point;


typedef struct {
/*structure for all the observations on a particular
individual*/
```

```
int id; /*gene ID*/
int nobs; /*number of obs.*/
Point dat[MAXPOINTS]; /*timepoints and the corresponding measurements*/
int config_theta; /*configuration vector of thetas*/
int config_eta;         /*configuration vector of etas*/
} Indiv;


typedef struct {
/*structure for a unique random effect and its multiplicity*/
gsl_vector *delta; /*vector of intercept and slope*/
gsl_vector *slope; /*vector of slope changes*/
gsl_vector *omega; /*vector of indicators for the changepoint slope mixture distribution*/
int count;
} Unique_theta;


typedef struct{
/*structure for a unique value of eta and its multiplicity*/
double eta; /*eta value*/
int count;  /*multiplicity*/
} Unique_eta;


typedef struct {
/*structure for all parameters (including hyperparameters)*/
double prec_theta; /*precision parameter for random effects*/
double prec_eta; /*precision parameter for the eta values*/
double sigma_sq_e; /*error variance*/
gsl_vector *cp; /*vector of changepoints*/
gsl_vector *sigma_sq_delta; /*vector of variances of the delta components*/
gsl_vector *sigma_sq_slope; /*vector of variances of the changepoint-slope components*/
gsl_vector *p; /*vector of probabilities of getting zero slopes in front of the changepoints*/
gsl_vector *u; /*vector of auxillary variables for p's*/
int num_unique_theta; /*number of unique thet values in the DPP*/
int num_unique_eta; /*number of unique eta values*/
Unique_theta star1[MAXGENE]; /*collection of the unique theta values*/
Unique_eta star2[MAXGENE]; /*collection of unique eta values*/
double prec_theta_shape, prec_theta_scale; /*shape and scale of precision of random effects*/
double prec_eta_shape, prec_eta_scale; /*shape and scale of the precision of etas*/
double sigma_shape; /*shape of error variance*/
double sigma_scale; /*scale of error variance*/
double delta_shape; /*shape of the variance of the delta's*/
double delta_scale; /*scale of the variances of the delta's*/
double slope_shape; /*shape of the variances of the slopes*/
double slope_scale; /*scale of the variances of the slopes*/
double rho; /*mixture probability*/
gsl_vector *delta_mu; /*baseline mean of delta*/
gsl_vector *slope_mu; /*baseline mean of slope*/
gsl_vector *d0; /*mean of delta_mu*/
gsl_vector *b0; /*mean of slope_mu*/
gsl_vector *sigma_sq_d0; /*variance of delta_mu*/
gsl_vector *sigma_sq_b0; /*variance of slope_mu*/
int df; /*df of the chi^2 distribution*/
```

```
} Param;

/*--------------routine prototypes---------------*/
void read_data(Indiv *, int *);
void print_data(Indiv *, const int);
void get_prior_parameters(Param *);
void initialize(Param *, Indiv *,  const int, gsl_rng *);
void update_prec_theta(Param *,  const int, gsl_rng *);
void update_prec_eta(Param *,  const int, gsl_rng *);
double traj(int, double, gsl_vector *, gsl_vector *, gsl_vector *,  Indiv *);
void update_sigma_sq_e(Param *, int, Indiv *, gsl_rng *);
void update_delta_var(Param *, const int, Indiv *, gsl_rng *);
void update_slope_var(Param *, const int, Indiv *, gsl_rng *);
void update_p(Param *, const int, Indiv *, gsl_rng *);
void update_u(Param *, int, Indiv *, gsl_rng *);
double log_lik_theta(int, gsl_vector *, gsl_vector *, Param *, Indiv *);
double log_lik_eta(int, double, Param *, Indiv *);
void update_theta_neal(Param *, int , Indiv *, gsl_rng *);
void update_eta_neal(Param *, int, Indiv *, gsl_rng *);
void update_delta_mu(Param *, int, Indiv *, gsl_rng *);
void update_slope_mu(Param *, int, Indiv *, gsl_rng *);
double log_df_dens(int, Param *, int, Indiv *);
void update_df(Param *, int, Indiv *, gsl_rng *);
void update_all(Param *, int, Indiv *,  gsl_rng *);
double log_lik_indiv(int, Param *,  Indiv *, int);
void burn(Param *, Indiv *, int, gsl_rng *);


/*-----------------routines-------------------*/

void read_data(Indiv *gene, int *num_gene)
/*read in the data*/
{
FILE *fp;
int i;
int j;
double express;

fp=fopen("cook.txt", "r");
*num_gene=MAXGENE;
for(i=0; i < *num_gene; i++)
{
gene[i].id=i;
gene[i].nobs=MAXPOINTS;
for(j = 0; j < gene[i].nobs; j++)
{
fscanf(fp, "%lf ", &express);
gene[i].dat[j].express=express;
gene[i].dat[j].timept=j;
}
}
fclose(fp);
}
```

```
void print_data(Indiv *gene, const int num_gene)
/* print out the read-in data*/
{
int i, j;

for(i = 0; i < num_gene; i++)
{
printf("id=%d ", gene[i].id);
for(j = 0; j < gene[i].nobs; j++)
printf("(t=%f,  y=%f) ", gene[i].dat[j].timept, gene[i].dat[j].express);
printf("\n\n");
}
}


void get_prior_parameters(Param *all)
/*get the prior parameters*/
{
/* printf("getting prior parameters...");*/
/*mean is shape*scale, variance is shape*scale^2*/
all->prec_theta_shape = 1.0;
all->prec_theta_scale = 1.0;

all->prec_eta_shape = 1.0;
all->prec_eta_scale = 1.0;

/*prior mean of sigma_e is 1/(scale*(shape-1)), prior variance is 1/((shape-1)^2*(shape-2)*scale^
all->sigma_shape = 1.0;/*2.1;*/
all->sigma_scale = 1.0;/*1.0/1.1;*/

all->delta_shape = 100.0;
all->delta_scale = 0.1;

all->slope_shape = 100.0;
all->slope_scale = 0.1;

all->rho = 0.50; /*prior probability of no change at a candidate point*/

/*set the mean of the baseline mean of delta*/
all->d0 = gsl_vector_calloc(2);

/*set the variance of the baseline mean of delta*/
all->sigma_sq_d0 = gsl_vector_alloc(2);
gsl_vector_set_all(all->sigma_sq_d0, 10.0);

/*set the mean of the baseline mean of slope*/
all->b0 = gsl_vector_calloc(LMAX);

/*set the variance of the baseline mean of slope*/
all->sigma_sq_b0 = gsl_vector_alloc(LMAX);
gsl_vector_set_all(all->sigma_sq_b0, 10.0);

/* printf("done\n");*/
}
```

```
void initialize(Param *all, Indiv *gene, const int num_gene, gsl_rng *r)
/*gets starting values of the parameters*/
{
int i, j, l;
int temp;

/* printf("initializing the chain...");*/

/*get precision*/
all->prec_theta = gsl_ran_gamma(r, all->prec_theta_shape, all->prec_theta_scale);
all->prec_eta = gsl_ran_gamma(r, all->prec_eta_shape, all->prec_eta_scale);

/*get sigma_sq_e*/
all->sigma_sq_e = 1.0 / gsl_ran_gamma(r, all->sigma_shape, all->sigma_scale);

/*get sigma_sq_delta*/
all->sigma_sq_delta = gsl_vector_alloc(2);
for(j = 0; j < 2; j++)
{
gsl_vector_set(all->sigma_sq_delta, j, 1.0 / gsl_ran_gamma(r, all->delta_shape, all->delta_scale)
}

/*get sigma_sq_slope*/
all->sigma_sq_slope = gsl_vector_alloc(LMAX);
for(j = 0; j < LMAX; j++)
{
gsl_vector_set(all->sigma_sq_slope, j, 1.0 / gsl_ran_gamma(r, all->slope_shape, all->slope_scale)
}

/*get p's and u's*/
all->p = gsl_vector_alloc(LMAX);
all->u = gsl_vector_alloc(LMAX);
for(j = 0; j < LMAX; j++)
{
temp = gsl_ran_bernoulli(r, all->rho);
gsl_vector_set(all->u, j, temp);
if(temp == 1)
{
gsl_vector_set(all->p, j, 0.0);
}
else
{
gsl_vector_set(all->p, j, gsl_rng_uniform(r));
}
}

/*get changepoint vector*/
all->cp=gsl_vector_alloc(LMAX);
for(j=0; j<LMAX; j++)
gsl_vector_set(all->cp, j, j+1);

/*get the chi^2 df*/
```

```
all->df = MAXDF/2.0;

/*get the number of unique eta's, their cluster structure and their values*/
all->num_unique_eta = 1;
all->star2[0].count = num_gene;
all->star2[0].eta = 1.0;
for(i = 0; i < num_gene; i++)
{
gene[i].config_eta = 0;
}

/*get the baseline means of deltas*/
all->delta_mu = gsl_vector_alloc(2);
for(j = 0; j < 2; j++)
{
gsl_vector_set(all->delta_mu, j, gsl_ran_gaussian(r, sqrt(gsl_vector_get(all->sigma_sq_d0, j)))
}

/*get the baseline means of slopes*/
all->slope_mu = gsl_vector_alloc(LMAX);
for(j = 0; j < LMAX; j++)
{
gsl_vector_set(all->slope_mu, j, gsl_ran_gaussian(r, sqrt(gsl_vector_get(all->sigma_sq_b0, j)))
}

all->num_unique_theta = 1;
all->star1[0].count = num_gene;
all->star1[0].delta = gsl_vector_alloc(2);
all->star1[0].slope = gsl_vector_alloc(LMAX);
all->star1[0].omega = gsl_vector_alloc(LMAX);
for(l = 0; l<2; l++)
{
gsl_vector_set(all->star1[0].delta, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(all->sigma_sq_delt
}
for(l = 0; l < LMAX; l++)
{
temp = gsl_ran_bernoulli(r, gsl_vector_get(all->p, l));
if(temp==1) gsl_vector_set(all->star1[0].slope, l, 0.0);
else gsl_vector_set(all->star1[0].slope, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(all->sigma_sc
gsl_vector_set(all->star1[0].omega, l, temp);
}

for(i = 0; i < num_gene; i++)
{
/*get the configuration*/
gene[i].config_theta = 0;
}
/* printf("done\n");*/
}


void update_prec_theta(Param *all,  const int num_gene, gsl_rng *r)
/*updates the precision parameter*/
{
```

```
double temp;
double mix_prob;
unsigned int indicator;

/* printf("updating precision parameter of theta's...");*/
temp = gsl_ran_beta(r, all->prec_theta + 1, num_gene);
mix_prob = (all->prec_theta_shape + all->num_unique_theta - 1) / (num_gene * (1.0 / all->prec_the
mix_prob = mix_prob / (1 + mix_prob);
indicator = gsl_ran_bernoulli(r, mix_prob);
if(indicator == 1)
all->prec_theta = gsl_ran_gamma(r, all->prec_theta_shape + all->num_unique_theta, 1.0 / (1.0 / al
else
all->prec_theta = gsl_ran_gamma(r, all->prec_theta_shape + all->num_unique_theta - 1, 1.0 / (1.0
/* printf("done\n");*/
}


void update_prec_eta(Param *all,  const int num_gene, gsl_rng *r)
/*updates the precision parameter of eta's*/
{
double temp;
double mix_prob;
unsigned int indicator;

/* printf("updating precision parameter of eta's...");*/
temp = gsl_ran_beta(r, all->prec_eta + 1, num_gene);
mix_prob = (all->prec_eta_shape + all->num_unique_eta - 1) / (num_gene * (1.0 / all->prec_eta_sca
mix_prob = mix_prob / (1 + mix_prob);
indicator = gsl_ran_bernoulli(r, mix_prob);
if(indicator == 1)
all->prec_eta = gsl_ran_gamma(r, all->prec_eta_shape + all->num_unique_eta, 1.0 / (1.0 / all->pre
else
all->prec_eta = gsl_ran_gamma(r, all->prec_eta_shape + all->num_unique_eta - 1, 1.0 / (1.0 / all-
/* printf("done\n");*/
}


double traj(int i, double t, gsl_vector *cp,  gsl_vector *delta, gsl_vector *slope,  Indiv *gene)
/*trajectory of the ith gene at time t, cp, alpha, delta slope*/
{
gsl_vector *z2vec = gsl_vector_calloc(2);
gsl_vector *z3vec = gsl_vector_calloc(LMAX);
int j;
double temp, temp2, temp3;

gsl_vector_set(z2vec, 0, 1.0);
gsl_vector_set(z2vec, 1, t);

for(j = 0; j < LMAX; j++)
{
temp = SPLINE(t - gsl_vector_get(cp, j));
gsl_vector_set(z3vec, j, temp);
}
```

```
gsl_blas_ddot(z2vec, delta, &temp2);
gsl_blas_ddot(z3vec, slope, &temp3);

if(isnan(temp2) || isnan(temp3))
{
printf("NAN in traj: temp2=%f, temp3=%f ", temp2, temp3);
printf("\n");
exit(1);
}
gsl_vector_free(z2vec);
gsl_vector_free(z3vec);

return(temp2 + temp3);
}

void update_sigma_sq_e(Param *all, int num_gene, Indiv *gene, gsl_rng *r)
/*updates the error variance*/
{
int i, j,  ni;
double sum1 = 0.0, sum2 = 0.0, temp;
double term;
double new_shape, new_scale;

/* printf("updating the error variance...");*/
for(i = 0; i < num_gene; i++)
{
ni = gene[i].nobs;
sum1 += ni;
term = 0.0;
for(j = 0; j < ni; j++)
{
temp = gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, all->cp,
 all->star1[gene[i].config_theta].delta, all->star1[gene[i].config_theta].slope, gene);
term += SQR(temp);
}
sum2 += all->star2[gene[i].config_eta].eta * term;
}
new_shape = all->sigma_shape + sum1 / 2.0;
new_scale = 1.0 / (1.0 / all->sigma_scale + 0.5 * sum2);
all->sigma_sq_e = 1.0 / gsl_ran_gamma(r, new_shape, new_scale);
/* printf("done\n");*/
}

void update_delta_var(Param *all, const int num_gene, Indiv *gene,  gsl_rng *r)
/*updating the variances of the delta's*/
{
int i, l;
double sum1, sum2;

/* printf("updating sigma_sq_delta...");*/
sum1 = all->delta_shape +  all->num_unique_theta / 2.0;
for(l=0; l<2; l++)
{
sum2 = 0.0;
```

```
for(i = 0; i < all->num_unique_theta; i++)
{
sum2 += SQR(gsl_vector_get(all->star1[i].delta, l) - gsl_vector_get(all->delta_mu, l));
}
sum2 = 1.0 / (1.0 / all->delta_scale + sum2/2.0);
gsl_vector_set(all->sigma_sq_delta, l, 1.0 / gsl_ran_gamma(r, sum1, sum2));
}
/* printf("done\n");*/
}

void update_slope_var(Param *all, const int num_gene, Indiv *gene, gsl_rng *r)
/*updating the variances of the slopes*/
{
int i, l;
double sum1, sum2;

/* printf("updating sigma_sq_slope...");*/
for(l = 0; l < LMAX; l++)
{
sum1 = 0.0;
sum2 = 0.0;
for(i = 0; i < all->num_unique_theta; i++)
{
sum1 += (1.0 - gsl_vector_get(all->star1[i].omega, l));
sum2 += (1.0 - gsl_vector_get(all->star1[i].omega, l)) * SQR(gsl_vector_get(all->star1[i].slope,
}
sum1 = all->slope_shape + sum1/2.0;
sum2 = 1.0 / (1.0 / all->slope_scale + sum2 / 2.0);
gsl_vector_set(all->sigma_sq_slope, l, 1.0 / gsl_ran_gamma(r, sum1, sum2));
}
/* printf("done\n");*/
}


void update_p(Param *all, const int num_gene, Indiv *gene, gsl_rng *r)
/*updating the p's*/
{
int i, l;
double sum;

/* printf("updating p's...");*/
for(l = 0; l < LMAX; l++)
{
if(gsl_vector_get(all->u, l) == 1) gsl_vector_set(all->p, l, 0.0);
else
{
sum = 0.0;
for(i = 0; i <  all->num_unique_theta; i++)
{
sum += gsl_vector_get(all->star1[i].omega, l);
}
gsl_vector_set(all->p, l, gsl_ran_beta(r, sum + 1, all->num_unique_theta - sum + 1));
}
}
```

```
/* printf("done\n");*/
}

void update_u(Param *all, int num_gene, Indiv *gene, gsl_rng *r)
/*update u's*/
{
int l;

/*printf("updating u's...");*/
for(l=0; l<LMAX; l++)
{
if(gsl_vector_get(all->p, l) == 0)
{
gsl_vector_set(all->u, l, gsl_ran_bernoulli(r, all->rho));
}
else gsl_vector_set(all->u, l, 0.0);
}
/*printf("done\n");*/
}


double log_lik_theta(int i, gsl_vector *delta, gsl_vector *slope, Param *all, Indiv *gene)
/*log-likelihood contribution of ith individual at theta=(delta, slope)*/
{
int j;
int ni = gene[i].nobs;
double temp;
double temp3 = 0.0;

for(j = 0; j < ni; j++)
{
temp = gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, all->cp, delta, slope, gene);
if(isnan(temp))
{
printf("NAN in |y - psi| for i=%d, j=%d\n", i, j);
exit(1);
}
temp3 -= SQR(temp);
}
temp3 *= all->star2[gene[i].config_eta].eta / (2.0 * all->sigma_sq_e);

return(temp3);
}

double log_lik_eta(int i, double eta, Param *all, Indiv *gene)
/*log-likelihood contribution of the ith gene at eta*/
{
int j;
double temp;
double sum = 0.0;

for(j = 0; j < gene[i].nobs; j++)
{
temp = gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, all->cp, all->star1[gene[i].config
sum -= SQR(temp);
}
sum *= eta / (2 * all->sigma_sq_e);
sum += gene[i].nobs / 2.0 * log(eta);

return(sum);
}

void update_theta_neal(Param *all, int num_gene, Indiv *gene,  gsl_rng *r)
/*Neal's algorithm 8 to update the configuration vector and the distinct parameter values for  DI
{
int i;
int k;
int ci;
int k_minus;
int h;
int extra = 5;
gsl_vector *tempslope=gsl_vector_alloc(LMAX);
gsl_vector *tempdelta=gsl_vector_alloc(2);
gsl_vector *tempomega=gsl_vector_alloc(LMAX);
int tempcount;
int j;
int l;
double probvec[num_gene + extra];
gsl_ran_discrete_t *aa;
int idx;
int c;
int temp;
double prob;
double maxprob;
double sum1, sum2, temp1, temp2;


/* printf("updating the configuration..");*/
for(i = 0; i < num_gene; i++)
{
/*get the current configuration*/
ci = gene[i].config_theta;
k = all->num_unique_theta;

(all->star1[ci].count)--;

if(all->star1[ci].count == 0)
/*occurs only once, hence relabel*/
{
/*begin relabeling*/
k_minus = k - 1;
h = k_minus + extra;
gsl_vector_memcpy(tempslope, all->star1[ci].slope);
gsl_vector_memcpy(tempdelta, all->star1[ci].delta);
gsl_vector_memcpy(tempomega, all->star1[ci].omega);
tempcount = all->star1[ci].count;

for(j = ci; j < k_minus; j++)
```

```
{
gsl_vector_swap(all->star1[j].slope, all->star1[j + 1].slope);
gsl_vector_swap(all->star1[j].delta, all->star1[j + 1].delta);
gsl_vector_swap(all->star1[j].omega, all->star1[j + 1].omega);
all->star1[j].count = all->star1[j + 1].count;
}
gsl_vector_swap(all->star1[k_minus].slope, tempslope);
gsl_vector_swap(all->star1[k_minus].delta, tempdelta);
gsl_vector_swap(all->star1[k_minus].omega, tempomega);
all->star1[k_minus].count = tempcount;

for(j = 0; j < num_gene; j++)
{
if(gene[j].config_theta > ci) gene[j].config_theta -= 1;
}
gene[i].config_theta = k - 1;
/*end relabeling*/

/*augment the set of distinct quantities by m additional values*/
for(j = k; j < h; j++)
{
all->star1[j].slope = gsl_vector_alloc(LMAX);
all->star1[j].omega = gsl_vector_alloc(LMAX);
all->star1[j].delta = gsl_vector_alloc(2);
for(l = 0; l < 2; l++)
gsl_vector_set(all->star1[j].delta, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(all->sigma_sq_delt
for(l = 0; l < LMAX; l++)
{
temp = gsl_ran_bernoulli(r, gsl_vector_get(all->p, l));
if(temp == 1) gsl_vector_set(all->star1[j].slope, l, 0.0);
else gsl_vector_set(all->star1[j].slope, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(all->sigma_sc
gsl_vector_set(all->star1[j].omega, l, temp);
}
}

/*draw a new value of ci for the ith element*/
for(j = 0; j < k_minus; j++)
{
probvec[j] = log_lik_theta(i, all->star1[j].delta, all->star1[j].slope, all, gene) + log(all->sta
}

for(j = k_minus; j < h; j++)
{
probvec[j] = log_lik_theta(i, all->star1[j].delta, all->star1[j].slope, all, gene) + log(all->pre
}
maxprob = probvec[0];
for(j = 1; j < h; j++)
maxprob = MAX(maxprob, probvec[j]);
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j] - maxprob);
}

aa = gsl_ran_discrete_preproc(h, probvec);
```

```
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

if(idx >= (k - 1)) /*a new value is drawn*/
{
gsl_vector_memcpy(all->star1[k - 1].slope, all->star1[idx].slope);
gsl_vector_memcpy(all->star1[k - 1].delta, all->star1[idx].delta);
gsl_vector_memcpy(all->star1[k - 1].omega, all->star1[idx].omega);
gene[i].config_theta = k - 1;
all->star1[gene[i].config_theta].count = 1;
for(j = k; j < h; j++)
{
gsl_vector_free(all->star1[j].slope);
gsl_vector_free(all->star1[j].delta);
gsl_vector_free(all->star1[j].omega);
}
}
else /*existing value is drawn*/
{
gene[i].config_theta = idx;
all->star1[idx].count++;
for(j = (k - 1); j < h; j++)
{
gsl_vector_free(all->star1[j].slope);
gsl_vector_free(all->star1[j].delta);
gsl_vector_free(all->star1[j].omega);
}
all->num_unique_theta--;
}
}
else if(all->star1[ci].count > 0)/*do not relabel*/
{
k_minus = k;
h = k_minus + extra;
for(j = k; j < h; j++)
{
all->star1[j].slope = gsl_vector_alloc(LMAX);
all->star1[j].omega = gsl_vector_alloc(LMAX);
all->star1[j].delta = gsl_vector_alloc(2);
for(l = 0; l < 2; l++)
gsl_vector_set(all->star1[j].delta, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(all->sigma_sq_delt
for(l = 0; l < LMAX; l++)
{
temp = gsl_ran_bernoulli(r, gsl_vector_get(all->p, l));
if(temp == 1) gsl_vector_set(all->star1[j].slope, l, 0.0);
else gsl_vector_set(all->star1[j].slope, l, gsl_ran_gaussian(r, sqrt(gsl_vector_get(all->sigma_sc
gsl_vector_set(all->star1[j].omega, l, temp);
}
}

for(j = 0; j < k; j++)
{
probvec[j] = log_lik_theta(i, all->star1[j].delta, all->star1[j].slope, all, gene) + log(all->sta
}
for(j = k; j < h; j++)
{
probvec[j] = log_lik_theta(i, all->star1[j].delta, all->star1[j].slope, all, gene) + log(all->pre
}


maxprob = probvec[0];
for(j = 1; j < h; j++)
maxprob = MAX(maxprob, probvec[j]);
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j] - maxprob);
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

if(idx > (k - 1))
{
gsl_vector_memcpy(all->star1[k].delta, all->star1[idx].delta);
gsl_vector_memcpy(all->star1[k].slope, all->star1[idx].slope);
gsl_vector_memcpy(all->star1[k].omega, all->star1[idx].omega);
gene[i].config_theta = k;
all->star1[gene[i].config_theta].count = 1;
(all->num_unique_theta)++;
for(j = (k + 1); j < h; j++)
{
gsl_vector_free(all->star1[j].slope);
gsl_vector_free(all->star1[j].delta);
gsl_vector_free(all->star1[j].omega);
}
}
else
{
gene[i].config_theta = idx;
all->star1[idx].count++;
for(j = k; j < h; j++)
{
gsl_vector_free(all->star1[j].slope);
gsl_vector_free(all->star1[j].delta);
gsl_vector_free(all->star1[j].omega);
}
}
}
}
gsl_vector_free(tempslope);
gsl_vector_free(tempdelta);
gsl_vector_free(tempomega);
/* printf("done\n");*/

/* printf("updating the distinct values...");*/
```

```
/****************here*****************/
for(c = 0; c < (all->num_unique_theta); c++)
/*update the cth distinct set of slope and delta*/
{
for(l = 0; l < 2; l++)
/*update the lth component of delta*/
{
sum1 = 0.0;
sum2 = 0.0;

for(i=0; i<num_gene; i++)
{
if(gene[i].config_theta==c)
{
temp1=0.0;
temp2=0.0;
for(j=0; j<gene[i].nobs; j++)
{
if(l==1)
{
temp1 += gene[i].dat[j].timept *
(gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, all->cp,  all->star1[c].delta, all->star
gsl_vector_get(all->star1[c].delta, l) * gene[i].dat[j].timept);
temp2 += SQR(gene[i].dat[j].timept);
}
else
{
temp1 += gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, all->cp,  all->star1[c].delta, a
gsl_vector_get(all->star1[c].delta, 0);
temp2 += 1.0;
}
}
sum1 += temp1 * all->star2[gene[i].config_eta].eta;
sum2 += temp2 * all->star2[gene[i].config_eta].eta;
}
}
sum1 = sum1 / all->sigma_sq_e + gsl_vector_get(all->delta_mu, l)/gsl_vector_get(all->sigma_sq_del
sum2 = sum2/all->sigma_sq_e + 1/gsl_vector_get(all->sigma_sq_delta, l);
gsl_vector_set(all->star1[c].delta, l, gsl_ran_gaussian(r, sqrt(1/sum2)) + sum1/sum2);
}

for(l = 0; l < LMAX; l++)
/*update the lth component of slope*/
{
if(gsl_vector_get(all->star1[c].omega, l) == 1) gsl_vector_set(all->star1[c].slope, l, 0.0);
else
{
sum1=0.0;
sum2=0.0;
for(i=0; i<num_gene; i++)
{
if(gene[i].config_theta==c)
{
temp1=0.0;
```

```
temp2=0.0;
for(j=0; j<gene[i].nobs; j++)
{
temp1 += SPLINE(gene[i].dat[j].timept - gsl_vector_get(all->cp, 1)) *
(gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, all->cp,  all->star1[c].delta, all->sta
gsl_vector_get(all->star1[c].slope, 1) * (SPLINE(gene[i].dat[j].timept - gsl_vector_get(all->cp,
temp2 += SQR(SPLINE(gene[i].dat[j].timept - gsl_vector_get(all->cp, 1)));
}
sum1 += temp1 * all->star2[gene[i].config_eta].eta;
sum2 += temp2 * all->star2[gene[i].config_eta].eta;
}
}
sum1 = sum1 / all->sigma_sq_e + gsl_vector_get(all->slope_mu, 1)/gsl_vector_get(all->sigma_sq_sl
sum2 = sum2/all->sigma_sq_e + 1/gsl_vector_get(all->sigma_sq_slope, 1);
gsl_vector_set(all->star1[c].slope, 1, gsl_ran_gaussian(r, sqrt(1/sum2)) + sum1/sum2);
}

/*update the corresponding omega*/
if(gsl_vector_get(all->star1[c].slope, 1) != 0) gsl_vector_set(all->star1[c].omega, 1, 0.0);
else
{
prob = gsl_vector_get(all->p, 1);
prob = prob / (prob + (1 - prob) * exp(-0.5 / gsl_vector_get(all->sigma_sq_slope, 1) * SQR(gsl_v
gsl_vector_set(all->star1[c].omega, 1, gsl_ran_bernoulli(r, prob));
}
}
}
/* printf("done\n");*/
}


void update_eta_neal(Param *all, int num_gene, Indiv *gene, gsl_rng *r)
{
int i;
int k;
int h;
int extra = 10;
int ci;
int j;
double probvec[num_gene + extra];
double maxprob;
gsl_ran_discrete_t *aa;
int idx;
double temp_eta;
int prevclus;
int c;
double sum1, sum2, temp;

for(i = 0; i < num_gene; i++)
{
k = all->num_unique_eta;
h = k + extra;
ci = gene[i].config_eta;
for(j = k; j < h; j++)
```

```
{
all->star2[j].eta = gsl_ran_gamma(r, all->df / 2.0, 2.0 / all->df);
}
all->star2[ci].count--;
if(all->star2[ci].count > 0)
{
for(j = 0; j < k; j++)
{
probvec[j] = log_lik_eta(i, all->star2[j].eta, all, gene) + log(all->star2[j].count);
}
for(j = k; j < h; j++)
{
probvec[j] = log_lik_eta(i, all->star2[j].eta, all, gene) + log(all->prec_eta) - log(extra);
}
maxprob = probvec[0];
for(j = 1; j < h; j++)
maxprob = MAX(maxprob, probvec[j]);

for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j] - maxprob);
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

gene[i].config_eta = idx;
if(gene[i].config_eta >= k)
{
temp_eta = all->star2[k].eta;
all->star2[k].eta = all->star2[gene[i].config_eta].eta;
all->star2[gene[i].config_eta].eta = temp_eta;
gene[i].config_eta = k;
k++;
}
all->star2[gene[i].config_eta].count++;
}
else if(all->star2[gene[i].config_eta].count == 0)
{
k--;
for(j = gene[i].config_eta; j < k; j++)
{
temp_eta = all->star2[j].eta;
all->star2[j].eta = all->star2[j + 1].eta;
all->star2[j + 1].eta = temp_eta;
all->star2[j].count = all->star2[j + 1].count;
}
all->star2[k].count = 0;
prevclus = gene[i].config_eta;
for(j = 0; j < num_gene; j++)
{
if(gene[j].config_eta > prevclus) gene[j].config_eta -= 1;
}
for(j = 0; j < k; j++)
{
probvec[j] = log_lik_eta(i, all->star2[j].eta, all, gene) + log(all->star2[j].count);
}
for(j = k; j < h; j++)
{
probvec[j] = log_lik_eta(i, all->star2[j].eta, all, gene) + log(all->prec_eta) - log(extra);
}
maxprob = probvec[0];
for(j = 1; j < h; j++)
maxprob = MAX(maxprob, probvec[j]);
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j] - maxprob);
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

gene[i].config_eta = idx;
if(gene[i].config_eta >= k)
{
temp_eta = all->star2[k].eta;
all->star2[k].eta = all->star2[gene[i].config_eta].eta;
all->star2[gene[i].config_eta].eta = temp_eta;
gene[i].config_eta = k;
k++;
}
all->star2[gene[i].config_eta].count++;
}
all->num_unique_eta = k;
}

for(c = 0; c < all->num_unique_eta; c++)
{
sum1 = 0.0;
sum2 = 0.0;
for(i = 0; i < num_gene; i++)
{
if(gene[i].config_eta == c)
{
sum1 += gene[i].nobs;
for(j = 0; j < gene[i].nobs; j++)
{
temp = gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, all->cp, all->star1[gene[i].config
sum2 += SQR(temp);
}
}
}
sum1 = (all->df + sum1) / 2.0;
sum2 = 1.0 / (0.5 * all->df + sum2 / (2 * all->sigma_sq_e));
all->star2[c].eta = gsl_ran_gamma(r, sum1, sum2);
}
}
```

```
void update_delta_mu(Param *all, int num_gene, Indiv *gene, gsl_rng *r)
{
int i;
int l;
double new_mean, new_var, temp;

for(l = 0; l < 2; l++)
{
new_mean=0.0;
for(i = 0; i< all->num_unique_theta; i++)
{
new_mean += gsl_vector_get(all->star1[i].delta, l);
}
new_mean = new_mean / gsl_vector_get(all->sigma_sq_delta, l) + gsl_vector_get(all->d0, l) / gsl_

new_var = 1.0 / (all->num_unique_theta/gsl_vector_get(all->sigma_sq_delta, l) + 1.0 / gsl_vector_
new_mean = new_mean * new_var;

temp = gsl_ran_gaussian(r, sqrt(new_var)) + new_mean;
gsl_vector_set(all->delta_mu, l, temp);
}
}

void update_slope_mu(Param *all, int num_gene, Indiv *gene, gsl_rng *r)
{
int i;
int l;
double new_mean, new_var, temp;

for(l = 0; l < LMAX; l++)
{
new_mean = 0.0;
new_var = 0.0;
for(i = 0; i < all->num_unique_theta; i++)
{
new_mean += gsl_vector_get(all->star1[i].slope, l) * (1 - gsl_vector_get(all->star1[i].omega, l))
new_var += (1 - gsl_vector_get(all->star1[i].omega, l));
}
new_mean = new_mean / gsl_vector_get(all->sigma_sq_slope, l) + gsl_vector_get(all->b0, l) / gsl_
new_var = 1.0 / (new_var/gsl_vector_get(all->sigma_sq_slope, l) + 1.0 / gsl_vector_get(all->sigma
new_mean = new_mean * new_var;
temp = gsl_ran_gaussian(r, sqrt(new_var)) + new_mean;
gsl_vector_set(all->slope_mu, l, temp);
}
}


double log_df_dens(int x, Param *all, int num_gene, Indiv *gene)
{
int j;
double term1, term2 = 0.0, term3 = 0.0, term;


for(j = 0; j < all->num_unique_eta; j++)
```

```
{
term3 += log(all->star2[j].eta);
term2 += all->star2[j].eta;
}

term1 = all->num_unique_eta * ( 0.5 * x * (log(x) - log(2)) - gsl_sf_lngamma(0.5 * x));
term2 *= (-0.5 * x);
term3 *= (0.5 * x);
if(isnan(term1) || isnan(term2) || isnan(term3))
{
printf("NAN in log-df-dens, term1=%f, term2=%f, term3=%f\n", term1, term2, term3);
exit(1);
}
term = term1 + term2 + term3;
return(term);
}


void update_df(Param *all, int num_gene, Indiv *gene, gsl_rng *r)
/*updates the df of the inverse-chi^2*/
{
int i;
double probvec[MAXDF], maxprob;
gsl_ran_discrete_t *aa;
int idx;

/* printf("updating df ...");*/
for(i = 0; i < MAXDF; i++)
{
probvec[i] = log_df_dens(i + 1, all, num_gene, gene);
}
maxprob = probvec[0];
for(i = 1; i < MAXDF; i++)
{
maxprob = MAX(maxprob, probvec[i]);
}

for(i = 0; i < MAXDF; i++)
{
probvec[i] = exp(probvec[i] - maxprob);
}
aa = gsl_ran_discrete_preproc(MAXDF, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

all->df = idx + 1;
/* printf("done\n");*/
}


void update_all(Param *all, int num_gene, Indiv *gene, gsl_rng *r)
{
update_prec_theta(all,  num_gene, r); /*printf("two\n");*/
update_prec_eta(all, num_gene, r); /*printf("three\n");*/
```

```
update_sigma_sq_e(all, num_gene, gene, r); /*printf("four\n");*/
update_delta_var(all, num_gene, gene, r); /*printf("five\n");*/
update_slope_var(all, num_gene, gene, r); /*printf("six\n");*/
update_p(all, num_gene, gene, r); /*printf("seven\n");*/
update_u(all, num_gene, gene, r); /*printf("eight\n");*/
update_theta_neal(all, num_gene, gene, r);  /*printf("thirteen\n");*/
update_eta_neal(all, num_gene, gene, r); /*printf("fourteen\n");*/
update_delta_mu(all, num_gene, gene, r); /*printf("fifteen\n");*/
update_slope_mu(all, num_gene, gene, r); /*printf("sixteen\n");*/
update_df(all, num_gene, gene, r); /*printf("seventeen\n");*/
}


double log_lik_indiv(int i, Param *all,  Indiv *gene,  int num_gene)
/*log-likelihood of ith gene*/
{
double sum = 0.0;
int j;

for(j = 0; j < gene[i].nobs; j++)
{
sum -= SQR(gene[i].dat[j].express - traj(i, gene[i].dat[j].timept, all->cp,  all->star1[gene[i].c
}
sum /= (2 * all->sigma_sq_e);
sum -= gene[i].nobs / 2 * log(all->sigma_sq_e);
sum += gene[i].nobs / 2 * log(all->star2[gene[i].config_eta].eta);

return(sum);
}

void burn(Param *all, Indiv *gene, int num_gene, gsl_rng *r)
/*burns the sampler and prints out parameters of interest (or their functions)*/
{
int l;
int burn;
double log_lik;
int i;

for(burn = 0; burn < NUM_BURN; burn++)
/*burn the Gibbs sampler, while monitoring certain quantities of interest*/
{
if(burn % NUM_THIN == 0)
{
printf("%d ", all->df);
printf("%d ", all->num_unique_eta);
printf("%.4f ", all->prec_eta);
printf("%d ", all->num_unique_theta);
printf("%.4f ", all->prec_theta);
printf("%.4f ", all->sigma_sq_e);
for(l = 0; l < 2; l++)
printf("%.4f ", gsl_vector_get(all->sigma_sq_delta, l));
for(l = 0; l < LMAX; l++)
printf("%.4f ", gsl_vector_get(all->sigma_sq_slope, l));
```

```
for(l = 0; l < 2; l++)
printf("%.4f ", gsl_vector_get(all->delta_mu, l));

for(l = 0; l < LMAX; l++)
printf("%.4f ", gsl_vector_get(all->slope_mu, l));
for(l = 0; l < LMAX; l++)
printf("%.4f ", gsl_vector_get(all->p, l));

log_lik = 0.0;
for(i = 0; i < num_gene; i++)
{
log_lik +=  log_lik_indiv(i, all,  gene,  num_gene);
}
printf("%.4f\n", log_lik);
}
update_all(all, num_gene, gene,  r);
}
}


int main(void)
/*---------------------main program------------*/
{
int num_gene;
Indiv gene[MAXGENE];
Param all;
const gsl_rng_type *T;
gsl_rng *r;
int i, l;
double incid_mat[MAXGENE][MAXGENE];
int iter;
int count;
double log_lik;


printf("num_gene=%d, NUM_ITER=%d, NUM_BURN=%d, NUM_THIN=%d, LMAX=%d\n", num_gene, NUM_ITER, NUM_
read_data(gene, &num_gene);
/* print_data(gene, num_gene);*/
get_prior_parameters(&all);

/*start the random number generator*/
gsl_rng_env_setup();
T = gsl_rng_default;
r = gsl_rng_alloc(T);

/*initialize all the parameters*/
initialize(&all,  gene,   num_gene, r);

burn(&all,  gene, num_gene, r);
/* burn the sampler for some time, until satisfied with its convergence*/

/*initialize the incidence matrix*/
for(i = 0; i < num_gene; i++)
{
```

```
for(l = (i + 1); l < num_gene; l++)
{
incid_mat[i][l] = 0.0;
}
}

/*generate more iterations from the sampler after it has converged.
use them to get the incidence matrix*/

count = 0;
for(iter = 0; iter < NUM_ITER; iter++)
{
update_all(&all, num_gene, gene,  r);
if(iter % NUM_THIN == 0)
{
printf( "%d ", all.df);
printf( "%d ", all.num_unique_eta);
printf( "%.4f ", all.prec_eta);
printf( "%d ", all.num_unique_theta);
printf( "%.4f ", all.prec_theta);
printf( "%.4f ", all.sigma_sq_e);
for(l = 0; l < 2; l++)
{
printf("%.4f ", gsl_vector_get(all.sigma_sq_delta, l));
}
for(l = 0; l < LMAX; l++)
{
printf("%.4f ", gsl_vector_get(all.sigma_sq_slope, l));
}

for(l = 0; l < 2; l++)
{
printf("%.4f ", gsl_vector_get(all.delta_mu, l));
}

for(l = 0; l < LMAX; l++)
{
printf("%.4f ", gsl_vector_get(all.slope_mu, l));
}
for(l = 0; l < LMAX; l++)
{
printf("%.4f ", gsl_vector_get(all.p, l));
}

log_lik = 0.0;

for(i = 0; i < num_gene; i++)
{
log_lik +=  log_lik_indiv(i, &all, gene,num_gene);
}
printf("%.4f\n", log_lik);

count++;
for(i = 0; i < num_gene; i++)
{
for(l = (i + 1); l < num_gene; l++)
{
if(gene[i].config_theta == gene[l].config_theta)
{
incid_mat[i][l]++;
}
}

}
}
}
gsl_rng_free(r);
printf("\n");


/*print out the relative clustering frequencies to generate the heatmap of clusters*/
printf("heatmap data\n");
for(i = 0; i < num_gene; i++)
{
for(l = 0; l < num_gene; l++)
{
if(l > i)
{
printf("%.4f ", incid_mat[i][l]/count);
}
else
{
if(l < i)
{
printf("%.4f ", incid_mat[l][i]/count);
}
else
{
printf("%.4f ", 1.00);
}
}
}
printf("\n");
}
return 0;
}
```

A.2 Chap. 4

A.2.1 Homoscedastic model

```
/*---------------included files--------------------*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_sort_vector.h>
#include <gsl/gsl_sort_float.h>
#include <gsl/gsl_sort_vector_float.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_cdf.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_permute.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_integration.h>
#include <gsl/gsl_statistics.h>
#include "arms.h"

/*-------------constants----------------*/
#define MATHLIB_STANDALONE 1
#define NUM_ITER 5000 /*iterations after burn-in*/
#define NUM_BURN 5000 /*burn-in*/
#define NUM_THIN 10 /*thinning*/
#define NGENE 798 /*number of genes in the file*/
#define NOBS 18 /*number of measurements on each gene*/
#define NEXTRA 5 /*number of extra quantities to draw in Algorithm 8. Should be a positive intege

/*--------------macros-------------------*/
#define MIN(x, y) ((x) < (y) ? (x)  : (y))
#define MAX(x, y) ((x) > (y) ? (x)  : (y))
#define SQR(x) ((x) * (x))

/*-------special data type declarations---------*/
typedef struct {
/*structure for all the information on a particular gene*/
int id; /*gene ID*/
gsl_vector *value; /*vector of expression levels*/
int config; /*configuration, saying who it is clustered with*/
} Indiv;

typedef struct {
/*structure for information on a particular cluster*/
gsl_vector *mu; /*vector of mean expression levels*/
double rho; /*correlation coefficient*/
int count; /*number of occurences*/
} Unique;

typedef struct {
/*structure for all parameters (including hyperparameters) for the model*/
double prec; /*precision parameter for the DPP*/
double sigma; /*(common) variance of the expression values*/

gsl_vector *base_mu; /*baseline mean vector*/
gsl_vector *base_sigma; /*baseline variance vector (assuming independence)*/

int num_unique; /*number of clusters in the DPP*/
Unique star[NGENE + NEXTRA]; /*collection of the unique values, one for each cluster*/

double prec_shape; /*shape of gamma prior on precision parameter*/
double prec_scale; /*scale of gamma prior on precision parameter*/
double sigma_shape; /*shape of IG prior on common variance*/
double sigma_scale; /*scale of IG prior common variance*/
double rho_shape1; /*shape 1 of the beta prior on correlations*/
double rho_shape2; /*shape 2 of the beta prior on correlations*/
double base_sig_shape; /*shape of IG prior on baseline variances*/
double base_sig_scale; /*scale of IG prior on baseline variances*/
} Param;

typedef struct {
Indiv *gene; /*data on all the genes*/
Param *theta; /*data on all the parameters*/
int c; /*which distinct value of rho*/
} rho_data;

/*--------------routine prototypes---------------*/
double log_det(gsl_matrix *, int);
void mat_inv(gsl_matrix *, int, gsl_matrix *);
void mvn(gsl_vector *, gsl_matrix *, int, gsl_vector *, gsl_rng *);
double quad_form(gsl_vector *, gsl_matrix *, int);
void read_data(Indiv *);
void print_data(Indiv *);
void get_prior_parameters(Param *);
void initialize(Param *, Indiv *, gsl_rng *);
void update_prec(Param *, gsl_rng *);
void get_corr_mat(double, gsl_matrix *, int);
void update_sigma_sq(Param *, Indiv *, gsl_rng *);
void update_baseline_variances(Param *, Indiv *, gsl_rng *);
double log_rho_dens(double, void *);
double log_dens(int, double, gsl_vector *, Indiv *, Param *);
void alg8(Param *, Indiv *, gsl_rng *);
void update_all(Param *, Indiv *, gsl_rng *);

/*-----------------routines-------------------*/
double log_det(gsl_matrix *mat, int dim)
/*gets the logarithm of determinant of the matrix "mat" of dimension "dim"*/
{
gsl_matrix *temp_mat = gsl_matrix_alloc(dim, dim);
int i;
double sum = 0.0;

gsl_matrix_memcpy(temp_mat, mat);
gsl_linalg_cholesky_decomp(temp_mat);
for(i = 0; i < dim; i++)
sum += log(gsl_matrix_get(temp_mat, i, i));
sum *= 2.0;
gsl_matrix_free(temp_mat);
return(sum);
}

void mat_inv(gsl_matrix *mat, int dim, gsl_matrix *mat_inv)
```

```
/*inverts matrix mat of dimension "dim" and puts it in mat_inv*/
{
int signum;
gsl_permutation *p = gsl_permutation_alloc(dim);
gsl_matrix *g = gsl_matrix_alloc(dim, dim);

gsl_matrix_memcpy(g, mat);
gsl_linalg_LU_decomp(g, p, &signum);
gsl_linalg_LU_invert(g, p, mat_inv);
gsl_matrix_free(g);
gsl_permutation_free(p);
}

void mvn(gsl_vector *mu, gsl_matrix *Sigma, int dim, gsl_vector *x, gsl_rng *r)
/*generates a random vector "x" from multivariate normal with mean "mu" and covariance "Sigma"*/
{
int i;
gsl_vector *z = gsl_vector_alloc(dim);
gsl_matrix *sigma = gsl_matrix_alloc(dim, dim);

gsl_matrix_memcpy(sigma, Sigma);
for(i = 0; i < dim; i++)
gsl_vector_set(z, i, gsl_ran_gaussian(r, 1));
gsl_linalg_cholesky_decomp(sigma);

gsl_blas_dtrmv(CblasLower, CblasNoTrans, CblasNonUnit, sigma, z);

for(i = 0; i < dim; i++)
gsl_vector_set(x, i, gsl_vector_get(z, i) + gsl_vector_get(mu, i));
gsl_vector_free(z);
gsl_matrix_free(sigma);
}

double quad_form(gsl_vector *x, gsl_matrix *A, int dim)
/*calculates the quadratic form x'Ax*/
{
int i, j;
double temp = 0.0;

for(i = 0; i < dim; i++)
{
for(j = 0; j < dim; j++)
{
temp += gsl_vector_get(x, i) * gsl_vector_get(x, j) * gsl_matrix_get(A, i, j);
}
}
return(temp);
}

void read_data(Indiv *gene)
/*reading the input data. make sure to have equal number of observations on each gene*/
{
FILE *fp;
int i, j;
```

```
double temp;

printf("reading data ...");
fp = fopen("spelldata.prn", "r");
for(i = 0; i < NGENE; i++)
{
gene[i].id = i;
gene[i].value = gsl_vector_alloc(NOBS);
for(j = 0; j < NOBS; j++)
{
fscanf(fp, "%lf ", &temp);
gsl_vector_set(gene[i].value, j, temp);
}
}
fclose(fp);
printf("done\n");
}

void print_data(Indiv *gene)
/* print out the read-in data*/
{
int i, j;

for(i = 0; i < NGENE; i++)
{
printf("id=%d ", gene[i].id);
for(j = 0; j < NOBS; j++)
printf("%f ", gsl_vector_get(gene[i].value, j));
printf("\n");
}
}

void get_prior_parameters(Param *theta)
/*get the prior parameters*/
{
/* printf("getting prior parameters...");*/
/*mean is shape*scale, variance is shape*scale^2*/
theta->prec_shape = 1.0;
theta->prec_scale = 1.0;

/*prior mean of sigma^2 is 1/(scale*(shape-1)), prior variance is 1/((shape-1)^2*(shape-2)*scale^
theta->sigma_shape = 2.1;
theta->sigma_scale = 1.0/1.1;

/*parameters for the beta distribution on rho*/
theta->rho_shape1 = 1.0;
theta->rho_shape2 = 1.0;

/*parameters for the inverse gamma prior for sigma_j^2's*/
theta->base_sig_shape = 2.1;
theta->base_sig_scale = 1.0/1.1;
/* printf("done\n");*/
}
```

```
void initialize(Param *theta, Indiv *gene, gsl_rng *r)
/*gets starting values of the parameters*/
{
int i, j;
int start_type;
gsl_matrix *Sigma = gsl_matrix_calloc(NOBS, NOBS);


/* printf("initializing the chain...");*/
/*get precision*/
theta->prec = gsl_ran_gamma(r, theta->prec_shape, theta->prec_scale);

/*get sigma^2*/
theta->sigma = 1.0 / gsl_ran_gamma(r, theta->sigma_shape, theta->sigma_scale);

/*get the mean of the baseline and set it to zero*/
theta->base_mu = gsl_vector_calloc(NOBS);

/*get the variances in the baseline*/
theta->base_sigma = gsl_vector_alloc(NOBS);
for(i = 0; i < NOBS; i++)
gsl_vector_set(theta->base_sigma, i, 1.0 / gsl_ran_gamma(r, theta->base_sig_shape, theta->base_si
for(i = 0; i < NOBS; i++)
gsl_matrix_set(Sigma, i, i, gsl_vector_get(theta->base_sigma, i));

/*get the distinct values and their multiplicities*/
start_type = gsl_ran_bernoulli(r, 0.5); /*randomly choose 1 cluster or NGENE clusters to start*/
if(start_type == 0)
/*1 cluster*/
{
(theta->num_unique) = 1;
theta->star[0].count = NGENE;
theta->star[0].mu = gsl_vector_alloc(NOBS);
mvn(theta->base_mu, Sigma, NOBS, theta->star[0].mu, r);
theta->star[0].rho = gsl_ran_beta(r, theta->rho_shape1, theta->rho_shape2);
}
else
/*each observation is in its own cluster*/
{
(theta->num_unique) = NGENE;
for(j = 0; j < (theta->num_unique); j++)
{
theta->star[j].mu = gsl_vector_alloc(NOBS);
mvn(theta->base_mu, Sigma, NOBS, theta->star[j].mu, r);
theta->star[j].rho = gsl_ran_beta(r, theta->rho_shape1, theta->rho_shape2);
theta->star[j].count = 1;
}
}

for(i = 0; i < NGENE; i++)
{
/*get the configuration*/
if(start_type == 0) gene[i].config = 0; else gene[i].config = i;
}
```

```
/* printf("done\n");*/

}

void update_prec(Param *theta, gsl_rng *r)
/*updates the precision parameter*/
{
double eta;
double mix_prob;
unsigned int indicator;

/* printf("updating precision parameter...");*/
eta = gsl_ran_beta(r, theta->prec + 1, NGENE);
mix_prob = (theta->prec_shape + theta->num_unique - 1) / (NGENE * (1.0 / theta->prec_scale - log(
mix_prob = mix_prob / (1 + mix_prob);
indicator = gsl_ran_bernoulli(r, mix_prob);
if(indicator == 1)
theta->prec = gsl_ran_gamma(r, theta->prec_shape + theta->num_unique, 1.0 / (1.0 / theta->prec_sc
else
theta->prec = gsl_ran_gamma(r, theta->prec_shape + theta->num_unique - 1, 1.0 / (1.0 / theta->pre
/* printf("done\n");*/
}

void get_corr_mat(double rho, gsl_matrix *corr_mat, int dim)
/*creates correlation matrix "mat" of dimension "dim" as a function of rho*/
{
int i, j;

for(i = 0; i < dim; i++)
{
for(j = 0; j < dim; j++)
{
/* gsl_matrix_set(corr_mat, i, j, pow(rho, abs(i - j)));*/ /*use this option if the correlations
gsl_matrix_set(corr_mat, i, j, (i==j ? 1.0 : rho)); /*use this option if all the correlations are
}
}
}

void update_sigma_sq(Param *theta, Indiv *gene, gsl_rng *r)
/*updates the variance*/
{
int i;
gsl_matrix *Rho_mat = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *Rho_mat_inv = gsl_matrix_alloc(NOBS, NOBS);
double sum = 0.0;
double new_shape, new_scale;
gsl_vector *yvec = gsl_vector_alloc(NOBS);

/* printf("updating the error variance...");*/
for(i = 0; i < NGENE; i++)
{
gsl_matrix_set_zero(Rho_mat);
get_corr_mat(theta->star[gene[i].config].rho, Rho_mat, NOBS);
mat_inv(Rho_mat, NOBS, Rho_mat_inv);
```

```
gsl_vector_memcpy(yvec, gene[i].value);
gsl_vector_sub(yvec, theta->star[gene[i].config].mu);
sum += quad_form(yvec, Rho_mat_inv, NOBS);
}

new_shape = theta->sigma_shape + NGENE * NOBS / 2.0;
new_scale = 1.0 / theta->sigma_scale + sum / 2.0;
new_scale = 1.0 / new_scale;
theta->sigma = 1.0 / gsl_ran_gamma(r, new_shape, new_scale);
gsl_matrix_free(Rho_mat);
gsl_matrix_free(Rho_mat_inv);
gsl_vector_free(yvec);
/* printf("done\n");*/
}

void update_baseline_variances(Param *theta, Indiv *gene, gsl_rng *r)
/*updates the baseline variances*/
{
int i, j;
double new_shape, new_scale;

/* printf("updating baseline variances ...");*/
for(j = 0; j < NOBS; j++)
{
new_shape = theta->base_sig_shape + NGENE / 2.0;
new_scale = 0.0;
for(i = 0; i < NGENE; i++)
{
new_scale += SQR(gsl_vector_get(theta->star[gene[i].config].mu, j) - gsl_vector_get(theta->base_m
}

new_scale = 1.0 / theta->base_sig_scale + new_scale / 2.0;
new_scale = 1.0 / new_scale;
gsl_vector_set(theta->base_sigma, j, 1.0 / gsl_ran_gamma(r, new_shape, new_scale));
}
/* printf("done\n");*/
}

double log_rho_dens(double x, void *mydata)
/*log of the posterior density of the correlation (upto a constant)*/
{
rho_data *d;
Indiv *gene;
int i;
int c;
gsl_vector *mu = gsl_vector_alloc(NOBS);
gsl_matrix *Rho_mat = gsl_matrix_calloc(NOBS, NOBS);
gsl_matrix *Rho_mat_inv = gsl_matrix_alloc(NOBS, NOBS);
gsl_vector *tempy = gsl_vector_alloc(NOBS);
double temp = 0.0;
double sigma_sq;

if(isnan(x))
{
```

```
printf("NAN in logrhodens, x=%f, quitting\n", x);
exit(1);
}
d = mydata;
gene = d->gene;
c = d->c;
gsl_vector_memcpy(mu, d->theta->star[c].mu);
sigma_sq = d->theta->sigma;

get_corr_mat(x, Rho_mat, NOBS);
mat_inv(Rho_mat, NOBS, Rho_mat_inv);

for(i = 0; i < NGENE; i++)
{
if(gene[i].config == c)
{
gsl_vector_memcpy(tempy, gene[i].value);
gsl_vector_sub(tempy, mu);
temp -= quad_form(tempy, Rho_mat_inv, NOBS);
}
}

temp /= (2.0 * sigma_sq);
temp =- 0.5 * d->theta->star[c].count * exp(log_det(Rho_mat, NOBS));

temp += (d->theta->rho_shape1 - 1) * log(x) + (d->theta->rho_shape2 - 1) * log(1 - x);

gsl_vector_free(mu);
gsl_matrix_free(Rho_mat);
gsl_matrix_free(Rho_mat_inv);
gsl_vector_free(tempy);
return(temp);
}


double log_dens(int i, double rho, gsl_vector *mu, Indiv *gene, Param *theta)
/*gets the log-likeihood of the ith gene when its mean level is mu and correlation is rho*/
{

gsl_vector *tempy = gsl_vector_alloc(NOBS);
gsl_matrix *Rho = gsl_matrix_calloc(NOBS, NOBS);
gsl_matrix *Rho_inv = gsl_matrix_calloc(NOBS, NOBS);
double temp;


gsl_vector_memcpy(tempy, gene[i].value);
gsl_vector_sub(tempy, mu);

get_corr_mat(rho, Rho, NOBS);
mat_inv(Rho, NOBS, Rho_inv);


temp = quad_form(tempy, Rho_inv, NOBS);
temp /= (- 2.0 * theta->sigma);
temp -= 0.5 * log_det(Rho, NOBS);

gsl_vector_free(tempy);
gsl_matrix_free(Rho);
gsl_matrix_free(Rho_inv);
return(temp);
}


void alg8(Param *theta, Indiv *gene, gsl_rng *r)
/*Neal's algorithm 8 to update the configuration vector and the distinct parameter values for  DI
{
int i;
int ci;
int k;
int k_minus;
int h;
double temp_rho;
int temp_count;
int j;

double probvec[NGENE + NEXTRA];
gsl_ran_discrete_t *aa;
int idx;

rho_data mydata_rho;
double xprev_rho;
int ninit_rho = 4;
double xl_rho = 0.0;
double xr_rho = 1.0;
int dometrop_rho=1;
double xsamp_rho[1];
int c;
int err_rho;
gsl_vector *temp_mu = gsl_vector_alloc(NOBS);
gsl_vector *tempy = gsl_vector_alloc(NOBS);
gsl_matrix *Sigma = gsl_matrix_calloc(NOBS, NOBS);
gsl_matrix *Sigma_inv = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *new_Sigma = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *Rho = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *Rho_inv = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *Amat = gsl_matrix_alloc(NOBS, NOBS);
gsl_vector *eta = gsl_vector_calloc(NOBS);
gsl_vector *new_mu = gsl_vector_alloc(NOBS);


/* printf("updating the configuration..");*/

/*get the Sigma matrix*/
for(i=0; i < NOBS; i++)
{
gsl_matrix_set(Sigma, i, i, gsl_vector_get(theta->base_sigma, i));
}
```

```
for(i = 0; i<NGENE; i++)
{
/*get the current configuration*/
ci = gene[i].config;
k = theta->num_unique;

(theta->star[ci].count)--;

if(theta->star[ci].count == 0)
/*occurs only once, hence relabel*/
{
/*begin relabeling*/
k_minus = k - 1;
h = k_minus + NEXTRA;
gsl_vector_memcpy(temp_mu, theta->star[ci].mu);
temp_rho = theta->star[ci].rho;
temp_count = theta->star[ci].count;
for(j = ci; j < k_minus; j++)
{
gsl_vector_swap(theta->star[j].mu, theta->star[j + 1].mu);
theta->star[j].rho = theta->star[j + 1].rho;
theta->star[j].count = theta->star[j + 1].count;
}
gsl_vector_swap(theta->star[k_minus].mu, temp_mu);
theta->star[k_minus].rho = temp_rho;
theta->star[k_minus].count = temp_count;

for(j = 0; j < NGENE; j++)
{
if(gene[j].config > ci) gene[j].config -= 1;
}
gene[i].config = k - 1;
/*end relabeling*/

/*augment the set of distinct quantities by m additional values*/
for(j = k; j < h; j++)
{
theta->star[j].mu = gsl_vector_alloc(NOBS);
mvn(theta->base_mu, Sigma, NOBS, theta->star[j].mu, r);
theta->star[j].rho = gsl_ran_beta(r, theta->rho_shape1, theta->rho_shape2);
}

/*draw a new value of ci for the ith element*/
for(j = 0; j < k_minus; j++)
{
probvec[j] = log_dens(i, theta->star[j].rho, theta->star[j].mu, gene, theta) + log(theta->star[j]
}
for(j = k_minus; j < h; j++)
{
probvec[j] = log_dens(i, theta->star[j].rho, theta->star[j].mu, gene, theta) + log(theta->prec) -
}
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j]);
```

```
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

if(idx >= (k - 1)) /*a new value is drawn*/
{
gsl_vector_memcpy(theta->star[k - 1].mu, theta->star[idx].mu);
theta->star[k - 1].rho = theta->star[idx].rho;
gene[i].config = k - 1;
theta->star[gene[i].config].count = 1;
for(j = k; j < h; j++)
{
gsl_vector_free(theta->star[j].mu);
}
}
else /*existing value is drawn*/
{
gene[i].config = idx;
theta->star[idx].count++;
for(j = (k - 1); j < h; j++)
{
gsl_vector_free(theta->star[j].mu);
}
theta->num_unique--;
}
}
else if(theta->star[ci].count > 0)/*do not relabel*/
{
k_minus = k;
h = k_minus + NEXTRA;
for(j = k; j < h; j++)
{
theta->star[j].mu = gsl_vector_alloc(NOBS);
mvn(theta->base_mu, Sigma, NOBS, theta->star[j].mu, r);
theta->star[j].rho = gsl_ran_beta(r, theta->rho_shape1, theta->rho_shape2);
}

for(j = 0; j < k; j++)
{
probvec[j] = log_dens(i, theta->star[j].rho, theta->star[j].mu, gene, theta) + log(theta->star[j]
}
for(j = k; j < h; j++)
{
probvec[j] = log_dens(i, theta->star[j].rho, theta->star[j].mu, gene, theta) + log(theta->prec) -
}
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j]);
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);
```

```
if(idx > (k - 1))
{
gsl_vector_memcpy(theta->star[k].mu, theta->star[idx].mu);
theta->star[k].rho = theta->star[idx].rho;
gene[i].config = k;
theta->star[gene[i].config].count = 1;
(theta->num_unique)++;
for(j = (k + 1); j < h; j++)
{
gsl_vector_free(theta->star[j].mu);
}
}
else
{
gene[i].config = idx;
theta->star[idx].count++;
for(j = k; j < h; j++)
{
gsl_vector_free(theta->star[j].mu);
}
}
}
}
gsl_vector_free(temp_mu);
/* printf("done\n");*/


/* printf("updating the distinct values...");*/

mydata_rho.gene = gene;
mydata_rho.theta = theta;

for(c = 0; c < (theta->num_unique); c++)
/*update the cth distinct set of mu and rho*/
{
/*get the rho updated here*/
mydata_rho.c = c;
xprev_rho = theta->star[c].rho;

err_rho = arms_simple(ninit_rho, &xl_rho, &xr_rho, log_rho_dens, &mydata_rho, dometrop_rho, &xpre
if(err_rho > 0)
{
printf("error code in rho arms=%d\n", err_rho);
exit(1);
}
theta->star[c].rho = xsamp_rho[0];


/*get the mu updated here*/
mat_inv(Sigma, NOBS, Sigma_inv);
gsl_matrix_memcpy(new_Sigma, Sigma_inv);

get_corr_mat(theta->star[c].rho, Rho, NOBS);
gsl_matrix_scale(Rho, theta->sigma/theta->star[c].count);
```

```
mat_inv(Rho, NOBS, Rho_inv);
gsl_matrix_memcpy(Amat, Sigma_inv);
gsl_matrix_add(Amat, Rho_inv);
gsl_vector_set_zero(tempy);
for(i = 0; i < NGENE; i++)
{
if(gene[i].config == c)
{
gsl_vector_add(tempy, gene[i].value);
}
}
gsl_vector_scale(tempy, 1.0 / theta->star[c].count);

gsl_blas_dtrmv(CblasUpper, CblasNoTrans, CblasNonUnit, Rho_inv, tempy);
gsl_blas_dgemv(CblasNoTrans, 1.0, Sigma_inv, theta->base_mu, 1.0, tempy);
mat_inv(Amat, NOBS, new_Sigma);
gsl_blas_dtrmv(CblasUpper, CblasNoTrans, CblasNonUnit, new_Sigma, tempy);
mvn(tempy, new_Sigma, NOBS, theta->star[c].mu, r);
}
gsl_vector_free(eta);
gsl_vector_free(tempy);
gsl_matrix_free(Sigma);
gsl_matrix_free(Sigma_inv);
gsl_matrix_free(new_Sigma);
gsl_matrix_free(Rho);
gsl_matrix_free(Rho_inv);
gsl_matrix_free(Amat);
gsl_vector_free(new_mu);
/* printf("done\n");*/
}


void update_all(Param *theta, Indiv *gene, gsl_rng *r)
/*updates all the parameters*/
{
update_prec(theta, r);
update_sigma_sq(theta, gene, r);
update_baseline_variances(theta, gene, r);
alg8(theta, gene, r);
}


void burn(Param *theta, Indiv *gene, gsl_rng *r)
/*burns the sampler and prints out parameters of interest (or their functons)*/
{
int l;
int burn;


for(burn=0; burn<NUM_BURN; burn++)
/*burn the Gibbs sampler, while monitoring certain quantities of interest*/
{
if((burn % NUM_THIN) == 0)
{
```

```
printf("%d ", theta->num_unique);
printf("%f ", theta->prec);
printf("%f ", theta->sigma);

/*print out the mu and rho for the first gene*/
printf("%f ", theta->star[gene[0].config].rho);
for(l=0; l<NOBS; l++)
{
printf("%f ", gsl_vector_get(theta->star[gene[0].config].mu, l));
}
printf("\n");
}
update_all(theta, gene, r);
}
}


int main(void)
/*---------------------main program------------*/
{
Indiv gene[NGENE];
Param theta;
const gsl_rng_type *T;
gsl_rng *r;
int i, j;
double incid_mat[NGENE][NGENE];
int count;
int iter;


printf("NUM_ITER=%d, NUM_BURN=%d, NUM_THIN=%d\n", NUM_ITER, NUM_BURN, NUM_THIN);
read_data(gene);

/* print_data(gene);*/
get_prior_parameters(&theta);

/*start the random number generator*/
gsl_rng_env_setup();
T = gsl_rng_default;
r = gsl_rng_alloc(T);

/*initialize all the parameters*/
initialize(&theta, gene, r);

/*burn the sampler*/
burn(&theta, gene, r);

/*initialize the incidence matrix*/
for(i = 0; i < NGENE; i++)
{
for(j = (i + 1); j < NGENE; j++)
{
incid_mat[i][j] = 0.0;
```

```
}
}

/*get the incidence matrix*/
count=0;
for(iter = 0; iter < NUM_ITER; iter++)
{
update_all(&theta, gene, r);
if(iter % NUM_THIN ==0)
{
count++;
for(i = 0; i < NGENE; i++)
{
for(j = (i + 1); j < NGENE; j++)
{
if(gene[i].config == gene[j].config)
incid_mat[i][j]++;
}
}
}
}
gsl_rng_free(r);

/*print out the relative clustering frequencies, to be used to generate the heatmap*/
printf("heatmap data:\n");
for(i = 0; i < NGENE; i++)
{
for(j = 0; j < NGENE; j++)
{
if(j > i)
printf("%f ", incid_mat[i][j]/count);
else
{
if(j < i)
printf("%f ", incid_mat[j][i]/count);
else printf("%f ", 1.00);
}
}
printf("\n");
}

return 0;
}
```

A.2.2 Heteroscedastic model

```
/*---------------included files-------------------*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_sort.h>
```

```
#include <gsl/gsl_sort_vector.h>
#include <gsl/gsl_sort_float.h>
#include <gsl/gsl_sort_vector_float.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_cdf.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_permute.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_integration.h>
#include <gsl/gsl_statistics.h>
#include "arms.h"

/*-------------constants----------------*/
#define MATHLIB_STANDALONE 1
#define NUM_ITER 5000 /*iterations after burn-in*/
#define NUM_BURN 5000 /*burn-in*/
#define NUM_THIN 10 /*thinning*/
#define NGENE 798 /*number of genes in the file*/
#define NOBS 18 /*number of measurements on each gene*/
#define NEXTRA 5 /*number of extra quantities to draw in Algorithm 8. Should be a positive intege

/*--------------macros-------------------*/
#define MIN(x, y) ((x) < (y) ? (x)  : (y))
#define MAX(x, y) ((x) > (y) ? (x)  : (y))
#define SQR(x) ((x) * (x))

/*-------special data type declarations---------*/
typedef struct {
/*structure for all the information on a particular gene*/
int id; /*gene ID*/
gsl_vector *value; /*vector of expression levels*/
int config; /*configuration, indicating who it is clustered with*/
} Indiv;

typedef struct {
/*structure for information on a particular cluster*/
gsl_vector *mu; /*vector of mean expression levels*/
double rho; /*correlation coefficient*/
double sigma; /*standard deviation*/
int count; /*number of occurences*/
} Unique;

typedef struct {
/*structure for all parameters (including hyperparameters) for the model*/
double prec; /*precision parameter for the DPP*/

gsl_vector *base_mu; /*baseline mean vector*/
gsl_vector *base_sigma; /*baseline variance vector (assuming independence)*/

int num_unique; /*number of clusters in the DPP*/
Unique star[NGENE + NEXTRA]; /*collection of the unique values, one for each cluster*/

double prec_shape; /*shape of gamma prior on precision parameter*/
double prec_scale; /*scale of gamma prior on precision parameter*/

double sigma_shape; /*shape of IG prior on variance*/
double sigma_scale; /*scale of IG prior on variance*/

double rho_shape1; /*shape 1 of the beta prior on correlations*/
double rho_shape2; /*shape 2 of the beta prior on correlations*/

double base_sig_shape; /*shape of IG prior on baseline variances*/
double base_sig_scale; /*scale of IG prior on baseline variances*/
} Param;

typedef struct {
Indiv *gene; /*data on all the genes*/
Param *theta; /*data on all the parameters*/
int c; /*which distinct value of rho*/
} rho_data;

/*--------------routine prototypes---------------*/
double log_det(gsl_matrix *, int);
void mat_inv(gsl_matrix *, int, gsl_matrix *);
void mvn(gsl_vector *, gsl_matrix *, int, gsl_vector *, gsl_rng *);
double quad_form(gsl_vector *, gsl_matrix *, int);
void read_data(Indiv *);
void print_data(Indiv *);
void get_prior_parameters(Param *);
void initialize(Param *, Indiv *, gsl_rng *);
void update_prec(Param *, gsl_rng *);
void get_corr_mat(double, gsl_matrix *, int);
void update_baseline_variances(Param *, Indiv *, gsl_rng *);
double log_rho_dens(double, void *);
double log_dens(int, double, double, gsl_vector *, Indiv *, Param *);
void alg8(Param *, Indiv *, gsl_rng *);
void update_all(Param *, Indiv *, gsl_rng *);

/*-----------------routines-------------------*/
double log_det(gsl_matrix *mat, int dim)
/*gets the logarithm of determinant of the matrix "mat" of dimension "dim"*/
{
gsl_matrix *temp_mat = gsl_matrix_alloc(dim, dim);
int i;
double sum = 0.0;

gsl_matrix_memcpy(temp_mat, mat);
gsl_linalg_cholesky_decomp(temp_mat);
for(i = 0; i < dim; i++)
sum += log(gsl_matrix_get(temp_mat, i, i));
sum *= 2.0;
gsl_matrix_free(temp_mat);
return(sum);
}

void mat_inv(gsl_matrix *mat, int dim, gsl_matrix *mat_inv)
/*inverts matrix mat of dimension "dim" and puts it in mat_inv*/
```

```
{
int signum;
gsl_permutation *p = gsl_permutation_alloc(dim);
gsl_matrix *g = gsl_matrix_alloc(dim, dim);

gsl_matrix_memcpy(g, mat);
gsl_linalg_LU_decomp(g, p, &signum);
gsl_linalg_LU_invert(g, p, mat_inv);
gsl_matrix_free(g);
gsl_permutation_free(p);
}

void mvn(gsl_vector *mu, gsl_matrix *Sigma, int dim, gsl_vector *x, gsl_rng *r)
/*generates a random vector "x" from multivariate normal with mean "mu" and covariance "Sigma"*/
{
int i;
gsl_vector *z = gsl_vector_alloc(dim);
gsl_matrix *sigma = gsl_matrix_alloc(dim, dim);

gsl_matrix_memcpy(sigma, Sigma);
for(i = 0; i < dim; i++)
gsl_vector_set(z, i, gsl_ran_gaussian(r, 1));
gsl_linalg_cholesky_decomp(sigma);

gsl_blas_dtrmv(CblasLower, CblasNoTrans, CblasNonUnit, sigma, z);

for(i = 0; i < dim; i++)
gsl_vector_set(x, i, gsl_vector_get(z, i) + gsl_vector_get(mu, i));
gsl_vector_free(z);
gsl_matrix_free(sigma);
}

double quad_form(gsl_vector *x, gsl_matrix *A, int dim)
/*calculates the quadratic form x'Ax*/
{
int i, j;
double temp = 0.0;

for(i = 0; i < dim; i++)
{
for(j = 0; j < dim; j++)
{
temp += gsl_vector_get(x, i) * gsl_vector_get(x, j) * gsl_matrix_get(A, i, j);
}
}
return(temp);
}

void read_data(Indiv *gene)
/*reading the input data. make sure to have equal number of observations on each gene*/
{
FILE *fp;
int i, j;
double temp;
```

```
printf("reading data ...");
fp = fopen("spelldata.prn", "r");
for(i = 0; i < NGENE; i++)
{
gene[i].id = i;
gene[i].value = gsl_vector_alloc(NOBS);
for(j = 0; j < NOBS; j++)
{
fscanf(fp, "%lf ", &temp);
gsl_vector_set(gene[i].value, j, temp);
}
}
fclose(fp);
printf("done\n");
}

void print_data(Indiv *gene)
/* print out the read-in data*/
{
int i, j;

for(i = 0; i < NGENE; i++)
{
printf("id=%d ", gene[i].id);
for(j = 0; j < NOBS; j++)
printf("%f ", gsl_vector_get(gene[i].value, j));
printf("\n");
}
}

void get_prior_parameters(Param *theta)
/*get the prior parameters*/
{
/* printf("getting prior parameters...");*/
/*mean is shape*scale, variance is shape*scale^2*/
theta->prec_shape = 1.0;
theta->prec_scale = 1.0;

/*prior mean of sigma^2 is 1/(scale*(shape-1)), prior variance is 1/((shape-1)^2*(shape-2)*scale^
theta->sigma_shape = 2.1;
theta->sigma_scale = 1.0/1.1;

/*parameters for the beta distribution on rho*/
theta->rho_shape1 = 1.0;
theta->rho_shape2 = 1.0;

/*parameters for the inverse gamma prior for sigma_j^2's*/
theta->base_sig_shape = 2.1;
theta->base_sig_scale = 1.0/1.1;
/* printf("done\n");*/
}

void initialize(Param *theta, Indiv *gene, gsl_rng *r)
```

```
/*gets starting values of the parameters*/
{
int i, j;
int start_type;
gsl_matrix *Sigma = gsl_matrix_calloc(NOBS, NOBS);


/* printf("initializing the chain...");*/
/*get precision*/
theta->prec = gsl_ran_gamma(r, theta->prec_shape, theta->prec_scale);

/*get the mean of the baseline and set it to zero*/
theta->base_mu = gsl_vector_calloc(NOBS);

/*get the variances in the baseline*/
theta->base_sigma = gsl_vector_alloc(NOBS);
for(i = 0; i < NOBS; i++)
gsl_vector_set(theta->base_sigma, i, 1.0 / gsl_ran_gamma(r, theta->base_sig_shape, theta->base_si
for(i = 0; i < NOBS; i++)
gsl_matrix_set(Sigma, i, i, gsl_vector_get(theta->base_sigma, i));

/*get the distinct values and their multiplicities*/
start_type = gsl_ran_bernoulli(r, 0.5); /*randomly choose 1 cluster or NGENE clusters to start*/
if(start_type == 0)
/*1 cluster*/
{
(theta->num_unique) = 1;
theta->star[0].count = NGENE;
theta->star[0].mu = gsl_vector_alloc(NOBS);
mvn(theta->base_mu, Sigma, NOBS, theta->star[0].mu, r);
theta->star[0].rho = gsl_ran_beta(r, theta->rho_shape1, theta->rho_shape2);
theta->star[0].sigma = 1.0 / gsl_ran_gamma(r, theta->sigma_shape, theta->sigma_scale);
}
else
/*each observation is in its own cluster*/
{
(theta->num_unique) = NGENE;
for(j = 0; j < (theta->num_unique); j++)
{
theta->star[j].mu = gsl_vector_alloc(NOBS);
mvn(theta->base_mu, Sigma, NOBS, theta->star[j].mu, r);
theta->star[j].rho = gsl_ran_beta(r, theta->rho_shape1, theta->rho_shape2);
theta->star[j].count = 1;
theta->star[j].sigma = 1.0 / gsl_ran_gamma(r, theta->sigma_shape, theta->sigma_scale);
}
}

for(i = 0; i < NGENE; i++)
{
/*get the configuration*/
if(start_type == 0) gene[i].config = 0; else gene[i].config = i;
}
/* printf("done\n");*/
```

```
}

void update_prec(Param *theta, gsl_rng *r)
/*updates the precision parameter*/
{
double eta;
double mix_prob;
unsigned int indicator;

/* printf("updating precision parameter...");*/
eta = gsl_ran_beta(r, theta->prec + 1, NGENE);
mix_prob = (theta->prec_shape + theta->num_unique - 1) / (NGENE * (1.0 / theta->prec_scale - log(
mix_prob = mix_prob / (1 + mix_prob);
indicator = gsl_ran_bernoulli(r, mix_prob);
if(indicator == 1)
theta->prec = gsl_ran_gamma(r, theta->prec_shape + theta->num_unique, 1.0 / (1.0 / theta->prec_sc
else
theta->prec = gsl_ran_gamma(r, theta->prec_shape + theta->num_unique - 1, 1.0 / (1.0 / theta->pre
/* printf("done\n");*/
}

void get_corr_mat(double rho, gsl_matrix *corr_mat, int dim)
/*creates correlation matrix "mat" of dimension "dim" as a function of rho*/
{
int i, j;

for(i = 0; i < dim; i++)
{
for(j = 0; j < dim; j++)
{
gsl_matrix_set(corr_mat, i, j, pow(rho, abs(i - j))); /*use this option if the correlations decre
/* gsl_matrix_set(corr_mat, i, j, (i==j ? 1.0 : rho));*/ /*use this option if all the correlatior
}
}
}


void update_baseline_variances(Param *theta, Indiv *gene, gsl_rng *r)
/*updates the baseline variances*/
{
int i, j;
double new_shape, new_scale;

/* printf("updating baseline variances...");*/
for(j = 0; j < NOBS; j++)
{
new_shape = theta->base_sig_shape + NGENE / 2.0;
new_scale = 0.0;
for(i = 0; i < NGENE; i++)
{
new_scale += SQR(gsl_vector_get(theta->star[gene[i].config].mu, j) - gsl_vector_get(theta->base_m
}

new_scale = 1.0 / theta->base_sig_scale + new_scale / 2.0;
new_scale = 1.0 / new_scale;
gsl_vector_set(theta->base_sigma, j, 1.0 / gsl_ran_gamma(r, new_shape, new_scale));
}
/* printf("done\n");*/
}

/*
void update_baseline_variances_alternate(Param *theta, Indiv *gene, gsl_rng *r)
{
int i, j;
double new_shape, new_scale, temp;

new_shape = theta->base_sig_shape + NGENE *NOBS / 2.0;
new_scale=0.0;
for(j = 0; j < NOBS; j++)
{
for(i = 0; i < NGENE; i++)
{
new_scale += SQR(gsl_vector_get(theta->star[gene[i].config].mu, j) - gsl_vector_get(theta->base_m
}
}
new_scale = 1.0 / theta->base_sig_scale + new_scale / 2.0;
new_scale = 1.0 / new_scale;
temp = 1.0 / gsl_ran_gamma(r, new_shape, new_scale);
for(j=0; j<NOBS; j++)
gsl_vector_set(theta->base_sigma, j, temp);
}
*/


double log_rho_dens(double x, void *mydata)
/*log of the posterior density of a distinct correlation (upto a constant)*/
{
rho_data *d;
Indiv *gene;
int i;
int c;
gsl_vector *mu = gsl_vector_alloc(NOBS);
gsl_matrix *Rho_mat = gsl_matrix_calloc(NOBS, NOBS);
gsl_matrix *Rho_mat_inv = gsl_matrix_alloc(NOBS, NOBS);
gsl_vector *tempy = gsl_vector_alloc(NOBS);
double temp = 0.0;

if(isnan(x))
{
printf("NAN in logrhodens, x=%f, quitting\n", x);
exit(1);
}
d = mydata;
gene = d->gene;
c = d->c;
gsl_vector_memcpy(mu, d->theta->star[c].mu);

get_corr_mat(x, Rho_mat, NOBS);
mat_inv(Rho_mat, NOBS, Rho_mat_inv);

for(i = 0; i < NGENE; i++)
{
if(gene[i].config == c)
{
gsl_vector_memcpy(tempy, gene[i].value);
gsl_vector_sub(tempy, mu);
temp -= quad_form(tempy, Rho_mat_inv, NOBS);
}
}

temp /= (2.0 * d->theta->star[c].sigma);
temp =- 0.5 * d->theta->star[c].count * exp(log_det(Rho_mat, NOBS));

temp += (d->theta->rho_shape1 - 1) * log(x) + (d->theta->rho_shape2 - 1) * log(1 - x);

gsl_vector_free(mu);
gsl_matrix_free(Rho_mat);
gsl_matrix_free(Rho_mat_inv);
gsl_vector_free(tempy);
return(temp);
}


double log_dens(int i, double sigma, double rho, gsl_vector *mu, Indiv *gene, Param *theta)
/*gets the log-likeihood of the ith gene when its mean level is mu, variance is sigma  and correl
{

gsl_vector *tempy = gsl_vector_alloc(NOBS);
gsl_matrix *Rho = gsl_matrix_calloc(NOBS, NOBS);
gsl_matrix *Rho_inv = gsl_matrix_calloc(NOBS, NOBS);
double temp;


gsl_vector_memcpy(tempy, gene[i].value);
gsl_vector_sub(tempy, mu);

get_corr_mat(rho, Rho, NOBS);
mat_inv(Rho, NOBS, Rho_inv);


temp = quad_form(tempy, Rho_inv, NOBS);
temp /= (- 2.0 * sigma);
temp -= 0.5 * log_det(Rho, NOBS) + NOBS / 2.0 * log(sigma);

gsl_vector_free(tempy);
gsl_matrix_free(Rho);
gsl_matrix_free(Rho_inv);
return(temp);
}


void alg8(Param *theta, Indiv *gene, gsl_rng *r)
```

```
/*Neal's algorithm 8 to update the configuration vector and the distinct parameter values for  DF
{
int i;
int ci;
int k;
int k_minus;
int h;
double temp_rho;
int temp_count;
int j;

double probvec[NGENE + NEXTRA];
gsl_ran_discrete_t *aa;
int idx;

rho_data mydata_rho;
double xprev_rho;
int ninit_rho = 4;
double xl_rho = 0.0;
double xr_rho = 1.0;
int dometrop_rho=1;
double xsamp_rho[1];
int c;
int err_rho;
gsl_vector *temp_mu = gsl_vector_alloc(NOBS);
gsl_vector *tempy = gsl_vector_alloc(NOBS);
gsl_matrix *Sigma = gsl_matrix_calloc(NOBS, NOBS);
gsl_matrix *Sigma_inv = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *new_Sigma = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *Rho = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *Rho_inv = gsl_matrix_alloc(NOBS, NOBS);
gsl_matrix *Amat = gsl_matrix_alloc(NOBS, NOBS);
gsl_vector *eta = gsl_vector_calloc(NOBS);
gsl_vector *new_mu = gsl_vector_alloc(NOBS);
double temp_sigma;
double new_shape, new_scale;


/* printf("updating the configuration..");*/

/*get the Sigma matrix*/
for(i=0; i < NOBS; i++)
{
gsl_matrix_set(Sigma, i, i, gsl_vector_get(theta->base_sigma, i));
}

for(i = 0; i<NGENE; i++)
{
/*get the current configuration*/
ci = gene[i].config;
k = theta->num_unique;

(theta->star[ci].count)--;
```

```
if(theta->star[ci].count == 0)
/*occurs only once, hence relabel*/
{
/*begin relabeling*/
k_minus = k - 1;
h = k_minus + NEXTRA;
gsl_vector_memcpy(temp_mu, theta->star[ci].mu);
temp_rho = theta->star[ci].rho;
temp_sigma = theta->star[ci].sigma;
temp_count = theta->star[ci].count;
for(j = ci; j < k_minus; j++)
{
gsl_vector_swap(theta->star[j].mu, theta->star[j + 1].mu);
theta->star[j].rho = theta->star[j + 1].rho;
theta->star[j].sigma = theta->star[j+1].sigma;
theta->star[j].count = theta->star[j + 1].count;
}
gsl_vector_swap(theta->star[k_minus].mu, temp_mu);
theta->star[k_minus].rho = temp_rho;
theta->star[k_minus].sigma = temp_sigma;
theta->star[k_minus].count = temp_count;

for(j = 0; j < NGENE; j++)
{
if(gene[j].config > ci) gene[j].config -= 1;
}
gene[i].config = k - 1;
/*end relabeling*/

/*augment the set of distinct quantities by m additional values*/
for(j = k; j < h; j++)
{
theta->star[j].mu = gsl_vector_alloc(NOBS);
mvn(theta->base_mu, Sigma, NOBS, theta->star[j].mu, r);
theta->star[j].rho = gsl_ran_beta(r, theta->rho_shape1, theta->rho_shape2);
theta->star[j].sigma = 1.0 / gsl_ran_gamma(r, theta->sigma_shape, theta->sigma_scale);
}

/*draw a new value of ci for the ith element*/
for(j = 0; j < k_minus; j++)
{
probvec[j] = log_dens(i, theta->star[j].sigma, theta->star[j].rho, theta->star[j].mu, gene, theta
}
for(j = k_minus; j < h; j++)
{
probvec[j] = log_dens(i, theta->star[j].sigma, theta->star[j].rho, theta->star[j].mu, gene, theta
}
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j]);
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);
```

```
if(idx >= (k - 1)) /*a new value is drawn*/
{
gsl_vector_memcpy(theta->star[k - 1].mu, theta->star[idx].mu);
theta->star[k - 1].rho = theta->star[idx].rho;
theta->star[k-1].sigma = theta->star[idx].sigma;
gene[i].config = k - 1;
theta->star[gene[i].config].count = 1;
for(j = k; j < h; j++)
{
gsl_vector_free(theta->star[j].mu);
}
}
else /*existing value is drawn*/
{
gene[i].config = idx;
theta->star[idx].count++;
for(j = (k - 1); j < h; j++)
{
gsl_vector_free(theta->star[j].mu);
}
theta->num_unique--;
}
}
else if(theta->star[ci].count > 0)/*do not relabel*/
{
k_minus = k;
h = k_minus + NEXTRA;
for(j = k; j < h; j++)
{
theta->star[j].mu = gsl_vector_alloc(NOBS);
mvn(theta->base_mu, Sigma, NOBS, theta->star[j].mu, r);
theta->star[j].rho = gsl_ran_beta(r, theta->rho_shape1, theta->rho_shape2);
theta->star[j].sigma = 1.0 / gsl_ran_gamma(r, theta->sigma_shape, theta->sigma_scale);
}

for(j = 0; j < k; j++)
{
probvec[j] = log_dens(i, theta->star[j].sigma, theta->star[j].rho, theta->star[j].mu, gene, theta
}
for(j = k; j < h; j++)
{
probvec[j] = log_dens(i, theta->star[j].sigma, theta->star[j].rho, theta->star[j].mu, gene, theta
}
for(j = 0; j < h; j++)
{
probvec[j] = exp(probvec[j]);
}
aa = gsl_ran_discrete_preproc(h, probvec);
idx = gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

if(idx > (k - 1))
{
```

```
gsl_vector_memcpy(theta->star[k].mu, theta->star[idx].mu);
theta->star[k].rho = theta->star[idx].rho;
theta->star[k].sigma = theta->star[idx].sigma;
gene[i].config = k;
theta->star[gene[i].config].count = 1;
(theta->num_unique)++;
for(j = (k + 1); j < h; j++)
{
gsl_vector_free(theta->star[j].mu);
}
}
else
{
gene[i].config = idx;
theta->star[idx].count++;
for(j = k; j < h; j++)
{
gsl_vector_free(theta->star[j].mu);
}
}
}
}
gsl_vector_free(temp_mu);
/* printf("done\n");*/


/* printf("updating the distinct values...");*/

mydata_rho.gene = gene;
mydata_rho.theta = theta;

for(c = 0; c < (theta->num_unique); c++)
/*update the cth distinct set of mu and rho and sigma*/
{
/*get the rho updated here*/
mydata_rho.c = c;
xprev_rho = theta->star[c].rho;

err_rho = arms_simple(ninit_rho, &xl_rho, &xr_rho, log_rho_dens, &mydata_rho, dometrop_rho, &xpre
if(err_rho > 0)
{
printf("error code in rho arms=%d\n", err_rho);
exit(1);
}
theta->star[c].rho = xsamp_rho[0];

/*get the sigma updated here*/
get_corr_mat(theta->star[c].rho, Rho, NOBS);
mat_inv(Rho, NOBS, Rho_inv);
new_shape = theta->sigma_shape + NOBS * theta->star[c].count / 2.0;
new_scale = 0.0;
for(i = 0; i < NGENE; i++)
{
if(gene[i].config == c)
```

```
{
gsl_vector_memcpy(tempy, gene[i].value);
gsl_vector_sub(tempy, theta->star[c].mu);
new_scale += quad_form(tempy, Rho_inv, NOBS);
}
}
new_scale = 1.0 / (1.0 / theta->sigma_scale + new_scale / 2.0);
theta->star[c].sigma = 1.0 / gsl_ran_gamma(r, new_shape, new_scale);




/*get the mu updated here*/
mat_inv(Sigma, NOBS, Sigma_inv);
gsl_matrix_memcpy(new_Sigma, Sigma_inv);

get_corr_mat(theta->star[c].rho, Rho, NOBS);
gsl_matrix_scale(Rho, theta->star[c].sigma / theta->star[c].count);
mat_inv(Rho, NOBS, Rho_inv);
gsl_matrix_memcpy(Amat, Sigma_inv);
gsl_matrix_add(Amat, Rho_inv);
gsl_vector_set_zero(tempy);
for(i = 0; i < NGENE; i++)
{
if(gene[i].config == c)
{
gsl_vector_add(tempy, gene[i].value);
}
}
gsl_vector_scale(tempy, 1.0 / theta->star[c].count);

gsl_blas_dtrmv(CblasUpper, CblasNoTrans, CblasNonUnit, Rho_inv, tempy);
gsl_blas_dgemv(CblasNoTrans, 1.0, Sigma_inv, theta->base_mu, 1.0, tempy);
mat_inv(Amat, NOBS, new_Sigma);
gsl_blas_dtrmv(CblasUpper, CblasNoTrans, CblasNonUnit, new_Sigma, tempy);
mvn(tempy, new_Sigma, NOBS, theta->star[c].mu, r);
}
gsl_vector_free(eta);
gsl_vector_free(tempy);
gsl_matrix_free(Sigma);
gsl_matrix_free(Sigma_inv);
gsl_matrix_free(new_Sigma);
gsl_matrix_free(Rho);
gsl_matrix_free(Rho_inv);
gsl_matrix_free(Amat);
gsl_vector_free(new_mu);
/* printf("done\n");*/
}


void update_all(Param *theta, Indiv *gene, gsl_rng *r)
/*updates all the parameters*/
{
update_prec(theta, r);
```

```
update_baseline_variances(theta, gene, r);
alg8(theta, gene, r);
}


void burn(Param *theta, Indiv *gene, gsl_rng *r)
/*burns the sampler and prints out parameters of interest (or their functons)*/
{
int l;
int burn;


for(burn=0; burn<NUM_BURN; burn++)
/*burn the Gibbs sampler, while monitoring certain quantities of interest*/
{
if((burn % NUM_THIN) == 0)
{

printf("%d ", theta->num_unique);
printf("%f ", theta->prec);

/*print out the sigma, rho and mu for the first gene*/
printf("%f %f ", theta->star[gene[0].config].sigma, theta->star[gene[0].config].rho);
for(l=0; l<NOBS; l++)
{
printf("%f ", gsl_vector_get(theta->star[gene[0].config].mu, l));
}
printf("\n");
}
update_all(theta, gene, r);
}
}


int main(void)
/*---------------------main program------------*/
{
Indiv gene[NGENE];
Param theta;
const gsl_rng_type *T;
gsl_rng *r;
int i, j;
double incid_mat[NGENE][NGENE];
int count;
int iter;


printf("NUM_ITER=%d, NUM_BURN=%d, NUM_THIN=%d\n", NUM_ITER, NUM_BURN, NUM_THIN);
read_data(gene);

/* print_data(gene);*/
get_prior_parameters(&theta);
```

```
/*start the random number generator*/
gsl_rng_env_setup();
T = gsl_rng_default;
r = gsl_rng_alloc(T);

/*initialize all the parameters*/
initialize(&theta, gene, r);

/*burn the sampler*/
burn(&theta, gene, r);

/*initialize the incidence matrix*/
for(i = 0; i < NGENE; i++)
{
for(j = (i + 1); j < NGENE; j++)
{
incid_mat[i][j] = 0.0;
}
}

/*get the incidence matrix*/
count=0;
for(iter = 0; iter < NUM_ITER; iter++)
{
update_all(&theta, gene, r);
if(iter % NUM_THIN ==0)
{
count++;
for(i = 0; i < NGENE; i++)
{
for(j = (i + 1); j < NGENE; j++)
{
if(gene[i].config == gene[j].config)
incid_mat[i][j]++;
}
}
}
}
gsl_rng_free(r);

/*print out the relative clustering frequencies, to be used to generate the heatmap*/
printf("heatmap data:\n");
for(i = 0; i < NGENE; i++)
{
for(j = 0; j < NGENE; j++)
{
if(j > i)
printf("%f ", incid_mat[i][j]/count);
else
{
if(j < i)
printf("%f ", incid_mat[j][i]/count);
else printf("%f ", 1.00);
}
}
printf("\n");
}

return 0;
}
```

A.3 Chap. 5

A.3.1 NDP Model

```
model {


# describe data

for ( i in 1: N) {
d[i,1] ~ dnorm(mu[i,1],sigmasq)
mu[i,1] <- d0[i]+beta[i,1]

for ( j in 2:M) {

d[i,j] ~ dnorm(mu[i,j],sigmasq)
mu[i,j] <- d[i,j-1]+beta[i,j]
}
d0[i] ~ dnorm(d[i,1],0.1)
}


#draw v's

for ( k in 1:L) {
v[k] ~ dbeta(1,alpha)
}


#stick breaking for pi using v

pi[1] <- v[1]
tempsum[1] <- log(1-v[1])

for ( k in 2:(L-1)) {
log(pi[k]) <- log(v[k])+ tempsum[k-1]
tempsum[k] <- tempsum[k-1] + log(1-v[k])
}
pi[L] <- 1-sum(pi[1:(L-1)])


#draw u's

for ( k in 1:L) {

for ( m in 1:L2) {
```

```
u[k,m] ~ dbeta(1,alpha2)
}
}

#stick breaking for w using u

for ( k in 1:L) {

w[k,1] <- u[k,1]
tempsum2[k,1] <- log(1-u[k,1])

for ( m in 2:(L2-1)) {
log(w[k,m]) <- log(u[k,m])+ tempsum2[k,m-1]
tempsum2[k,m] <- tempsum2[k,m-1] + log(1-u[k,m])
}
w[k,L2] <- 1-sum(w[k,1:(L2-1)])
}


#get latent, latent2, beta, theta

for ( i in 1:N) {
latent[i] ~ dcat(pi[1:L])

for ( j in 1:M) {
beta[i,j] <- theta[latent[i],latent2[latent[i],j]]
}
}

for ( k in 1:L) {

for ( j in 1:M) {
latent2[k,j] ~ dcat(w[k,1:L2])
}

for ( m in 1:L2) {
theta[k,m] ~ dnorm(eta, tau)
}
}


#hyperparameters

sigmasq ~ dgamma(5,1)
alpha ~ dgamma(2.01,1.01)
alpha2 ~ dgamma(2.01,1.01)
eta ~ dnorm(0,1)
tau ~ dnorm(2.01,1.01)


#count clusters

for ( i in 1:N) {
```

```
for ( k in 1:L) {
Memb[i,k] <- equals(latent[i],k)
}
}

for ( k in 1:L) {
Tmemb[k] <- sum(Memb[,k])
Fmemb[k] <- step(Tmemb[k]-1)
}
Tcluster <- sum(Fmemb[])

}#end
```

A.3.2 Slope model

```
/*----------------------included files and headers-------------------------------------*/
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>
#include<gsl/gsl_rng.h>
#include<gsl/gsl_randist.h>
#include<gsl/gsl_sf_gamma.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>

/*-----------------------numerical constants-------------------------*/
#define NUM_ITER 10000/*number of Gibbs iterations*/
#define MAX_GENE 20000 /*maximum number of genes*/
#define NUM_BURN 5000 /*number of burn-in*/
#define NUM_THIN 20 /*thinning interval*/
#define MAXPOINTS 20 /*maximum number of observations per gene*/

/*------------------------macros-----------------------------*/
#define SQR(x) ((x)*(x))

/*--------------------data definitions--------------------------------*/
/*structure for the observation at each time point*/
typedef struct {
double timept;
double obs;
double slope;
} Point;

/*structure for all the observations on a gene*/
typedef struct {
int nobs;
Point dat[MAXPOINTS];
} Gene;

/*structure for prior parameters*/
typedef struct {
double prec_shape;
double prec_scale;
double sigma_shape;
double sigma_scale;
double nu;
double sigma0_sq;
double kappa;
double mu0;
} Prior;

/*structure for a unique mean-variance pair and its multiplicity*/
typedef struct {
double mu;
double sigma_sq;
int count;
} Unique;

/*structure for all parameters (except the slopes)*/
typedef struct {
double prec;
double sigma_sq;
Unique tau_star[MAX_GENE];
} Param;

/*------------------------routine prototypes---------------------*/
void read_data(Gene *, int);
void print_data(Gene *, int);
void get_prior_parameters(Prior *);
void get_starting_values(Gene *, int, Prior, Param *, int *, int *, gsl_rng *);
void update_sigma_sq(Gene *, Prior, Param *, int, gsl_rng *);
void update_prec(Gene *, Prior , Param *, int , int , gsl_rng *);
void update_slopes(Gene *, Param, int *, int, gsl_rng *);
void update_config(Gene *, Prior, Param *,  int *, int , int *, gsl_rng *);
void update_distinct(Gene *, int, Prior, Param *, int, int *, gsl_rng *);
void update_all(Gene *, int, int *, int *, Prior, Param *, gsl_rng *);
void get_optimal_cluster(Gene *, int , int, int *, Prior , Param *, gsl_rng *);
void get_dist(int , int *, gsl_matrix *);

/*-------------------------routines--------------------------------*/
void read_data(Gene *gene, int num_gene)
/*read in the information for each gene*/
{
FILE *fp;
int i, j;
int nobs;
float t, y;

printf("reading data...");
fp=fopen("gene_data.txt", "r");
for(i=0; i<num_gene; i++)
{
fscanf(fp, "%d", &nobs);
gene[i].nobs=nobs;
for(j=0; j<nobs; j++)
{
fscanf(fp, "%f %f", &t, &y);
```

```
gene[i].dat[j].timept=t;
gene[i].dat[j].obs=y;
}
}
fclose(fp);
printf("done\n");
}

void print_data(Gene *gene, int num_gene)
/*print out the data on time series profiles*/
{
int i, j;

printf("printing data...");
for(i=0;i<num_gene; i++)
{
printf("i=%d, ni=%d, ", i, gene[i].nobs);
for(j=0; j<gene[i].nobs; j++)
printf("(%3.0f, %5.3f) ", gene[i].dat[j].timept, gene[i].dat[j].obs);
printf("\n");
}
printf("done\n");
}


void get_prior_parameters(Prior *theta)
/*get the prior parameters*/
{
printf("getting prior parameters...");
theta->prec_shape=1.0;
theta->prec_scale=1.0; /*E(M) = shape * scale, V(M) = shape * scale^2*/

theta->sigma_shape=2.1;
theta->sigma_scale=10.0/11.0; /*E(sigma^2) = 1 /(scale(shape-1)), V(sigma^2) = 1/(scale^2(shape-1

theta->nu=1.5; /*need nu > 2 to have a finite expected prior variance, but nu < 2 implies a flat
theta->sigma0_sq=1.0;/*make this bigger to accomodate bigger prior variance of slopes*/
theta->kappa=1.0; /*keep this at 1 always*/
theta->mu0=0.0; /*prior mean of the slopes*/
printf("done\n");
}

/*get starting values for all parameters*/
void get_starting_values(Gene *gene, int num_gene, Prior theta, Param *gamma, int
*config, int *num_unique, gsl_rng *r)
{
int i, j;

printf("getting starting values...");
gamma->prec=gsl_ran_gamma(r, theta.prec_shape, theta.prec_scale);
gamma->sigma_sq=1.0/gsl_ran_gamma(r, theta.sigma_shape, theta.sigma_scale);
for(i=0; i<num_gene; i++)
{
gamma->tau_star[i].sigma_sq=theta.nu*theta.sigma0_sq/gsl_ran_chisq(r, theta.nu);
```

```
gamma->tau_star[i].mu=gsl_ran_gaussian(r, sqrt(gamma->tau_star[i].sigma_sq/theta.kappa))+theta.mu
gamma->tau_star[i].count=1;
for(j=0; j<(gene[i].nobs-1); j++)
gene[i].dat[j].slope=gsl_ran_gaussian(r, sqrt(gamma->tau_star[i].sigma_sq))+gamma->tau_star[i].mu
config[i]=i;
}
(*num_unique)=num_gene;
printf("done\n");
}

void update_prec(Gene *gene, Prior theta, Param *gamma, int num_gene, int num_unique, gsl_rng *r)
{
double eta;
double mix_prob;
unsigned int indicator;

/* printf("updating the precision parameter..");*/
eta = gsl_ran_beta(r, gamma->prec + 1, num_gene);
mix_prob = (theta.prec_shape + num_unique -1)/(num_gene * (1.0/theta.prec_scale - log(eta)));
mix_prob = mix_prob/(1+mix_prob);
indicator = gsl_ran_bernoulli(r, mix_prob);
if(indicator==1)
gamma->prec = gsl_ran_gamma(r, theta.prec_shape + num_unique, 1.0/(1.0/theta.prec_scale -log(eta)
else
gamma->prec = gsl_ran_gamma(r, theta.prec_shape + num_unique -1, 1.0/(1.0/theta.prec_scale -log(e
/* printf("done\n");*/
}

void update_sigma_sq(Gene *gene, Prior theta, Param *gamma, int num_gene, gsl_rng *r)
{
double new_shape=0.0;
double new_scale=0.0;
double new_sigma_sq;
int i, j;

/* printf("updating sigma_sq...");*/
for(i=0; i<num_gene; i++)
{
new_shape+=gene[i].nobs;
for(j=0; j<(gene[i].nobs-1); j++)
new_scale+=SQR((gene[i].dat[j+1].obs-gene[i].dat[j].obs-gene[i].dat[j].slope*(gene[i].dat[j+1].ti
}
new_shape=theta.sigma_shape+new_shape/2.0-num_gene/2.0;
new_scale=1.0/theta.sigma_scale+new_scale/2.0;
new_scale=1.0/new_scale;
new_sigma_sq=gsl_ran_gamma(r, new_shape, new_scale);
gamma->sigma_sq=1.0/new_sigma_sq;
/* printf("done\n");*/
}

void update_slopes(Gene *gene, Param gamma, int *config, int num_gene, gsl_rng *r)
{
int i, j;
double new_var;
double new_mean;
double new_slope;

/* printf("updating slopes...");*/
for(i=0; i<num_gene; i++)
{
new_var=1.0/gamma.sigma_sq + 1.0/gamma.tau_star[config[i]].sigma_sq;
new_var=1.0/new_var;
for(j=0; j<(gene[i].nobs-1); j++)
{
new_mean=(gene[i].dat[j+1].obs-gene[i].dat[j].obs)/SQR(gene[i].dat[j+1].timept-gene[i].dat[j].tim
new_mean*=new_var;
new_slope=gsl_ran_gaussian(r, sqrt(new_var))+new_mean;
gene[i].dat[j].slope=new_slope;
}
}
/* printf("done\n");*/
}

void update_config(Gene *gene, Prior theta, Param *gamma, int *config, int num_gene, int *num_uni
{
int i, j;
double probvec[num_gene];
int ci;
int k;
int l;
double sum1, sum2, sum;
gsl_ran_discrete_t *aa;
int idx;
double temp1, temp2, temp3;
double nu_star, sum3, sum4, sigma0_star_sq, mu0_star, kappa_star, new_sigma_sq, new_mu;

/* printf("updating configuration...");*/
k=(*num_unique);
for(i=0; i<num_gene; i++)
{
ci=config[i];
if(gamma->tau_star[ci].count==1)
{
/*need to relabel the distinct values and the configuration*/
temp1=gamma->tau_star[ci].mu;
temp2=gamma->tau_star[ci].sigma_sq;
temp3=gamma->tau_star[ci].count;
for(j=ci; j<(k-1); j++)
{
gamma->tau_star[j].mu=gamma->tau_star[j+1].mu;
gamma->tau_star[j].sigma_sq=gamma->tau_star[j+1].sigma_sq;
gamma->tau_star[j].count=gamma->tau_star[j+1].count;
}
gamma->tau_star[k-1].mu=temp1;
gamma->tau_star[k-1].sigma_sq=temp2;
gamma->tau_star[k-1].count=temp3;

for(j=0; j<num_gene; j++)
```

```
if(config[j]>ci) config[j]--;
config[i]=k-1;

k--;
}
gamma->tau_star[config[i]].count--;

/*get the probabilities of c_i taking various values*/
for(j=0; j<k; j++)
{
probvec[j]=0.0;
for(l=0; l<(gene[i].nobs-1); l++)
probvec[j]+=SQR(gene[i].dat[l].slope-gamma->tau_star[j].mu);
probvec[j]/=(2.0*gamma->tau_star[j].sigma_sq);
probvec[j]=-probvec[j]-(gene[i].nobs-1)/2.0*log(gamma->tau_star[j].sigma_sq);
probvec[j]+=log(gamma->tau_star[j].count);
}

sum1=0.0;
sum2=0.0;
for(j=0; j<(gene[i].nobs-1); j++)
{
sum1+=gene[i].dat[j].slope;
sum2+=SQR(gene[i].dat[j].slope);
}
sum2+=theta.nu*theta.sigma0_sq+theta.kappa*SQR(theta.mu0);
sum1+=theta.kappa*theta.mu0;
sum=sum2-SQR(sum1)/(gene[i].nobs -1 + theta.kappa);

probvec[k] = 0.5*(log(theta.kappa) - log(gene[i].nobs-1+theta.kappa))+0.5*theta.nu*(log(theta.nu)
probvec[k] += gsl_sf_lngamma(0.5*(theta.nu + gene[i].nobs-1)) - gsl_sf_lngamma(0.5*theta.nu);
probvec[k] -= 0.5*(theta.nu +gene[i].nobs-1)*log(sum);
probvec[k] += log(gamma->prec);

for(j=0; j<=k; j++)
{
probvec[j]=exp(probvec[j]);
}

aa=gsl_ran_discrete_preproc(k+1, probvec);
idx=gsl_ran_discrete(r, aa);
gsl_ran_discrete_free(aa);

if(idx>=k)
/*get a new value for c_i*/
{
nu_star = theta.nu + gene[i].nobs - 1;
sum1 = 0.0;
sum2 = 0.0;
for( j = 0; j < (gene[i].nobs - 1); j++)
{
sum1 += gene[i].dat[j].slope;
sum2 +=SQR(gene[i].dat[j].slope);
}
```

```
sum4 = sum2 + theta.kappa * SQR(theta.mu0) + theta.nu * SQR(theta.sigma0_sq);
sum3 = sum1 + theta.kappa * theta.mu0;
sum = sum4 - SQR(sum3)/(gene[i].nobs - 1 + theta.kappa);
kappa_star = theta.kappa + gene[i].nobs -1;
sigma0_star_sq = sum / nu_star;
mu0_star = sum3 / (gene[i].nobs - 1 + theta.kappa);

new_sigma_sq = nu_star * sigma0_star_sq / gsl_ran_chisq(r, nu_star);
new_mu = gsl_ran_gaussian(r, sqrt(new_sigma_sq/kappa_star)) + mu0_star;


gamma->tau_star[k].mu = new_mu;
gamma->tau_star[k].sigma_sq  = new_sigma_sq;
gamma->tau_star[k].count=1;
config[i]=k;
k++;
}
else
/*c_i is one of the other c_j values*/
{
config[i]=idx;
gamma->tau_star[idx].count++;
}
}
*num_unique=k;
/* printf("done\n");*/
}

void update_distinct(Gene *gene, int num_unique, Prior theta, Param *gamma, int num_gene, int *cc
{
int i, j, l;
double nu_star, kappa_star, sigma0_star_sq, mu0_star;
double sum1, sum2, sum3;
double new_sigma_sq, new_mu;

/* printf("updating the distinct values...");*/
for(i=0; i<(num_unique); i++)
/*update the ith distinct mean-variance pair*/
{
sum1=0.0;
sum2=0.0;
sum3=0.0;
for(j=0; j<num_gene; j++)
{
if(config[j]==i)
{
for(l=0; l<(gene[j].nobs-1); l++)
{
sum1 +=1;
sum2+=gene[j].dat[l].slope;
sum3+=SQR(gene[j].dat[l].slope);
}
}
}
```

```
nu_star=sum1+theta.nu;
kappa_star=theta.kappa+sum1;
mu0_star=(theta.kappa*theta.mu0+sum2)/kappa_star;
sigma0_star_sq=theta.kappa*SQR(theta.mu0)+theta.nu*theta.sigma0_sq+sum3-SQR(mu0_star)*kappa_star;
sigma0_star_sq /= (sum1 + theta.nu);
new_sigma_sq = nu_star * sigma0_star_sq/gsl_ran_chisq(r, nu_star);

new_mu=gsl_ran_gaussian(r, sqrt(new_sigma_sq/kappa_star))+mu0_star;
gamma->tau_star[i].mu=new_mu;
gamma->tau_star[i].sigma_sq=new_sigma_sq;
}
/* printf("done\n");*/
}

void update_all(Gene *gene, int num_gene, int *num_unique, int *config, Prior theta, Param *gamma
{
update_prec(gene, theta, gamma, num_gene, *num_unique, r);
update_sigma_sq(gene, theta, gamma, num_gene, r);
update_slopes(gene, *gamma, config, num_gene, r);
update_config(gene, theta, gamma, config, num_gene, num_unique, r);
update_distinct(gene, *num_unique, theta, gamma, num_gene, config, r);
}

void get_optimal_cluster(Gene *gene, int num_gene, int num_unique, int *config, Prior theta, Para
/*gets the optimal configuration (and the corresponding cluster) that is closest to the "average'
{
gsl_matrix *dist_mat;
int count;
int iter;
gsl_matrix *all_config;
gsl_vector *dist_vec;
double sum;
int i, j, idx;
gsl_vector *opt_config;
double opt_distinct;
int k;

dist_mat = gsl_matrix_calloc(num_gene, num_gene);
count = 0;
for(iter = 0; iter < NUM_ITER; iter++)
{
update_all(gene, num_gene, &num_unique, config, theta, gamma, r);
if(iter % NUM_THIN == 0)
{
count++;
get_dist(num_gene, config, dist_mat);
}
}
gsl_matrix_scale(dist_mat, 1.0/count);

all_config = gsl_matrix_alloc(count, num_gene);
dist_vec = gsl_vector_alloc(count);

count = 0;
for(iter = 0; iter < NUM_ITER; iter++)
{
update_all(gene, num_gene,  &num_unique, config, theta, gamma, r);
if(iter % NUM_THIN == 0)
{
sum = 0.0;
for(j = 0; j < num_gene; j++)
{
gsl_matrix_set(all_config, count, j, config[j]);
for(k = 0; k < num_gene; k++)
sum += SQR(gsl_matrix_get(dist_mat, j, k) - (config[j] == config[k]));
}
gsl_vector_set(dist_vec, count, sum);
count++;
}
}

idx = gsl_vector_min_index(dist_vec);
opt_config = gsl_vector_alloc(num_gene);
for(i = 0; i < num_gene; i++)
gsl_vector_set(opt_config, i, gsl_matrix_get(all_config, idx, i));
opt_distinct = gsl_vector_max(opt_config)+1;

printf("optimal config: (");
for(i = 0; i < num_gene; i++)
printf("%f ", gsl_vector_get(opt_config, i));
printf(")\n");
printf("\noptimal cluster:\n");
for(i=0; i < opt_distinct; i++)
{
printf("%d: ", i);
for(j = 0; j < num_gene; j++)
if(gsl_vector_get(opt_config, j) == i) printf("%d, ", j);
printf("\n");
}
gsl_matrix_free(dist_mat);
gsl_matrix_free(all_config);
gsl_vector_free(dist_vec);
gsl_vector_free(opt_config);
}

void get_dist(int num_gene, int *config, gsl_matrix *dist_mat)
{
int i, j;
for(i = 0; i < num_gene; i++)
{
for(j = 0; j < num_gene; j++)
{
if(config[i] == config[j]) gsl_matrix_set(dist_mat, i, j, gsl_matrix_get(dist_mat, i, j) + 1);
}
}
}
```

```
/*-----------------------main program-------------------------*/
int main(void)
{
int num_gene=64;
Gene gene[500];
Prior theta;
const gsl_rng_type *T;
gsl_rng *r;
int config[MAX_GENE];
int num_unique;
Param gamma;
int i, j;

read_data(gene, num_gene);
/* print_data(gene, num_gene);*/
get_prior_parameters(&theta);

/*start the random number generator*/
gsl_rng_env_setup();
T=gsl_rng_default;
r=gsl_rng_alloc(T);
get_starting_values(gene, num_gene, theta, &gamma, config,  &num_unique, r);

for(i=0; i<NUM_BURN; i++)
{
/* printf("%d %f %f %f  %f", num_unique, gamma.prec, gamma.sigma_sq, gene[10].dat[0].slope, gene
printf("\n");*/
update_all(gene, num_gene, &num_unique, config, theta, &gamma, r);
}

get_optimal_cluster(gene, num_gene, num_unique, config, theta, &gamma, r);

gsl_rng_free(r);
return 0;
}
```

A.4 Chap. 6

```
clear;
load gwork;
%palpha = palpha(:,1:3:18);

p = 18; %number of time points
t = [1:p]';
N = 798; %number of data points

%q=2; %degree of polynomial
r=5; %number of groups

gnums = [0   113   412   483   604   798];

A = zeros(length(gnums)-1,N);
for i = 1:(length(gnums)-1),
    A(i,(gnums(i)+1):gnums(i+1)) = 1;
end;

X=palpha';
%for i = 1:798,
%    X(:,i) = repmat(i,18,1) + 10*randn(18,1);
%end

for q = 1:6,

for i = 1:q,
    B(:,i) = t.^(i-1);
end;

%for i = 1:2q,
%    B(:,2*i-1) = cos((i-1)*t);
%    B(:,2*i) = sin(i*t);
%end;

S=X*(eye(N)-A'*inv(A*A')*A)*X';
%S2=X*(eye(N)-A'*inv(A*A')*A)*X';

%alternate calculations of S
%S2 = zeros(p);
%for i = 1:5,
%    S2 = S2 + (gnums(i+1)-gnums(i)-1)*cov(palpha(gnums(i)+1:gnums(i+1),:));
%end

%S2 = cov(X');
%S = zeros(p);
%for i=1:18,
%    S(i,i) = S2(i,i);
%end

%S = eye(p);


%(B'*inv(S)*B)

psi=inv(B'*inv(S)*B)*(B'*inv(S)*X)*A'*inv(A*A')


%H0
meanfn=B*psi;
B2=(eye(p)-B*inv(B'*B)*B');
B2=B2(:,1:p-q);
E0=B2'*S*B2;
dm = p-q;
de = N-r;
dh = r;
lamb = (dm*dh-2)/4;
s = sqrt(((dm*dh)^2 - 4)/(dm^2 + dh^2 - 5));
m = N - (dm + dh + 1)/2;
```

```
df1 = dm * dh;
df2 = m*s - 2*lamb;

H0=B2'*X*[A'*inv(A*A')*A]*X'*B2;
L0=det(E0)/(det(E0+H0))

%f = -m*log(L0);
f = ((1-L0^(1/s))/(L0^(1/s)))*(df2/df1)

pvalue0 = 1-cdf('f',f,df1,df2)

%H1
dm = q;
dh = r-1;
de = N-r-(p-q);

R11=inv(A*A')*(eye(r)+A*X'*(inv(S)-inv(S)*B*inv(B'*inv(S)*B)*B'*inv(S))*X*A'*inv(A*A'));
L=eye(q);
M=[1 0 0 0
   0 1 0 0
   0 0 1 0
   0 0 0 1
  -1 -1 -1 -1];

H1=(L*psi*M)*inv(M'*R11*M)*(L*psi*M)';
E1=L*inv(B'*inv(S)*B)*L';
L1=det(E1)/det(E1+H1)

lamb = (dm*dh-2)/4;
s = sqrt(((dm*dh)^2 - 4)/(dm^2 + dh^2 - 5));
m = N - (dm + dh + 1)/2;

df1 = dm * dh;
df2 = m*s - 2*lamb;

%f1 = -m*log(L1);
f1 = ((1-L1^(1/s))/(L1^(1/s)))*(df2/df1)

pvalue1 = 1-cdf('f',f,df1,df2)

% Mu = B*psi*A;
% Sigma = kron(eye(N),S);
% for i = 1:(N*p),
%    vecX(i) = X(i);
%    vecMu(i) = Mu(i);
% end;
% Lik = mvnpdf(vecX,vecMu,Sigma)

%evaluate error

err = 0;

for i = 1:113,
    err = err + sum((X(:,i) - B*psi(:,1)).^2);
```

```
end;

for i = 114:412,
    err = err + sum((X(:,i) - B*psi(:,2)).^2);
end;

for i = 413:483,
    err = err + sum((X(:,i) - B*psi(:,3)).^2);
end;

for i = 484:604,
    err = err + sum((X(:,i) - B*psi(:,4)).^2);
end;

for i = 605:798,
    err = err + sum((X(:,i) - B*psi(:,5)).^2);
end;

error(q) = err;

%% evaluate likelihood

lik = 0;

lik = lik + sum(log(mvnpdf(X(:,1:113)',(B*psi(:,1))',S/(N-5*q))));
lik = lik + sum(log(mvnpdf(X(:,114:412)',(B*psi(:,2))',S/(N-5*q))));
lik = lik + sum(log(mvnpdf(X(:,413:483)',(B*psi(:,3))',S/(N-5*q))));
lik = lik + sum(log(mvnpdf(X(:,484:604)',(B*psi(:,4))',S/(N-5*q))));
lik = lik + sum(log(mvnpdf(X(:,605:798)',(B*psi(:,5))',S/(N-5*q))));

likli(q) = lik;

aic(q) = lik - 5*q;

bic(q) = lik - (5/2)*q*log(N);

end
```

Bibliography

[1] D. F. Andrews and C. L. Mallows. Scale mixtures of normal distributions. *Jour. Royal Stat. Soc.*, 36:99–102, 1974.

[2] Charles E Antoniak. Mixtures of Dirichlet processes with applications to non-parametric problems. *Annals of Statistics*, 2:1152–1174, 1974.

[3] Z. Bar-Joseph, G. Gerber, T.S. Jaakkola, D.K. Gifford, and I. Simon. Continuous representations of time series gene expression data. *Journal of Computational Biology*, 3:341–356, 2003.

[4] Asa Ben-Hur, Andre Elisseeff, and Isabelle Guyon. A stability based method for discovering structure in clustered data. In *Pacific Symposium on Biocomputing*, pages 6–17, 2002.

[5] N.G. Best, M. K. Cowles, and S. K. Vines. *CODA Manual Version 0.30*, 1995.

[6] J Bigelow and D B Dunson. Posterior simulation across nonparametric models for functional clustering. Technical report, Duke University, Department of Statistics, 2006.

[7] D.A. Binder. Bayesian cluster analysis. *Biometrika*, 65:31–38, 1978.

[8] David Blackwell and James B MacQueen. Ferguson distributions via Pólya urn schemes. *Annals of Statistics*, 1(2):353–355, 1973.

[9] Elizabeth Brown and Joseph G Ibrahim. A Bayesian semiparametric joint hierarchical model for longitudinal and survival data. *Biometrics*, 59:221–228, 2003.

[10] C. A. Bush and S. N. MacEachern. A Semiparametric Bayesian Model for Randomized Block Designs. *Biometrika*, 83:275–285, 1996.

[11] Yizong Cheng and George M. Church. Biclustering of expression data. In *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*, pages 93–103. AAAI Press, 2000.

[12] S. Chib and B. H. Hamilton. Semiparametric Bayes analysis of longitudinal data treatment models. *Journal of Econometrics*, 110:67–89, 2002.

[13] M.K. Cowles and B. P. Carlin. Markov Chain Monte Carlo Diagnostics: A Comparative Review. *Journal of the American Statistical Association*, 91:883–904, 1995.

[14] David Dahl. Model-based clustering for expression data via a Dirichlet process mixture model. In *Bayesian Inference for Gene Expression and Proteomics*, pages 201–218. Cambridge University Press, 2006.

[15] D. B. Dunson, A. H. Herring, and S. M. Engel. Bayesian selection and clustering of polymorphisms in functionally-related genes. *Journal of the American Statistical Association*, 103:534–546, 2008.

[16] M. D. Escobar and M. West. Computing Nonparametric Hierarchical Models. In Peter Müller Dipak Dey and Debajyoti Sinha, editors, *Practical Nonparametric and Semiparametric Bayesian Statistics*, volume 133 of *Lecture Notes in Statistics*, pages 1–22. Springer, New York, 1998.

[17] M.D. Escobar and M. West. Bayesian Density Estimation and Inference Using Mixtures. *Journal of the American Statistical Association*, 90(430):577–588, 1995.

[18] Thomas S Ferguson. A Bayesian analysis of some nonparametric problems. *Annals of Statistics*, 1(2):209–230, 1973.

[19] E. B. Fowlkes and C. L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, 78(383):553–569, 1983.

[20] G. K. Gerber, R. D. Dowell, T. S. Jaakkola, and D. K. Gifford. Automated discovery of functional generality of human gene expression programs. *PLoS Computational Biology*, 3:1426–1440, 2007.

[21] Kaushik Ghosh and Ram C Tiwari. Prediction of US cancer mortality counts using semiparametric Bayesian techniques. *Journal of the American Statistical Association*, 102:7–15, 2007.

[22] Pulak Ghosh, Kaushik Ghosh, and Ram C Tiwari. Joint modeling of longitudinal data and informative dropout time in the presence of multiple changepoints. submitted, 2009.

[23] Wally R Gilks and P Wild. Adaptive rejection sampling for Gibbs sampling. *Applied Statistics*, 41(2):337–348, 1992.

[24] Nicholas A. Heard, Chrostopher C. Holmes, and David A. Stephens. A quantitative study of gene regulation involved in the immune response of anopheline mosquitoes: An application of Bayesian hierarchical clustering of curves. *Jour. of American Statistical Association*, 101:18–29, 2006.

[25] Joseph G Ibrahim, Ming-Hui Chen, and Debajyoti Sinha. Bayesian methods for joint modeling of longitudinal and survival data with applications to cancer vaccine trials. *Statistica Sinica*, 14:863–883, 2004.

[26] G. James and T. Hastie. Functional linear discriminant analysis for irregularly sampled curves. *Jour. of the Royal Statistical Society*, B63:533–550, 2001.

[27] M. Kacperczyk, P. Damien, and S. G. Walker. A new class of Bayesian semiparametric models with applications to option pricing. Technical report, University of Michigan Business School, 2003.

[28] A. Kottas, M. D. Branco, and A. E. Gelfand. A nonparametric Bayesian modeling approach for cytogenetic dosimetry. *Biometrics*, 58:593–600, 2002.

[29] A. M. Kshirsagar and W. B. Smith. *Growth Curves.* Marcel Dekker, New York, 1995.

[30] Hiroshi Kurata. A generalization of Rao's covariance structure with applications to several linear models. *Journal of Multivariate Analysis*, 67:297–305, 1998.

[31] D. J. Laws and A. O'Hagan. A hierarchical Bayes model for multilocation auditing. *Journal of the Royal Statstical Society, Series D*, 51:431–450, 2002.

[32] X Liu, S Sivaganesan, K Yeung, J Guo, R E Baumgarner, and M Medvedovic. Context-specific infinite mixtures for clustering gene expression profiles across diverse microarray data set. *Bioinformatics*, 22:1737–1744, 2006.

[33] D.J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS – a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000.

[34] J.K. Magidson, J.; Vermunt. Latent class models for clustering: a comparison with *k*-means. *Canadian Journal of Marketing Research*, 20(1):36–43, 2002.

[35] Francesca Martella, Marco Alfo, and Maurizio Vichi. Biclustering of gene expression data by an extension of mixtures of factor analyzers. *International Journal of Biostatistics*, 4:1078–1078, 2008.

[36] Mario Medvedovic and Siva Sivaganesan. Bayesian infinite mixture model based clustering of gene expression profiles. *Bioinformatics*, 18(9):1194–1206, 2002.

[37] Mario Medvedovic, K. Y. Yeung, and R.E. Baumgarner. Bayesian mixture model based clustering of replicated microarray data. *Bioinformatics*, 20:1222–1232, 2004.

[38] Marina Meila. Comparing clusterings – an information based distance. submitted, 2006.

[39] R. M. Neal. Markov Chain Sampling Methods for Dirichlet Process Mixture Models. *Journal of Computational and Graphical Statistics*, 9(2):249–265, 2000.

[40] Jian-Xin Pan and Kai-Tai Fang. *Growth Curve Models and Statistical Diagnostics.* Springer, New York, 2002.

[41] R. Potthoff and S.N. Roy. A generalized multivariate analysis of variance models useful especially for growth curve problems. *Biometrika*, 51:313–326, 1964.

[42] J. Qian, M. Dolled-Filhart, J. Lin, H. Yu, and M. Gerstein. Beyond synexpression relationships: local clustering of time-shifted and inverted gene expression profiles identifies new, biologically relevant interactions. *Jour. Molecular Biology*, 314:1053–1066, 2001.

[43] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66:846–850, 1971.

[44] C. R. Rao. *Linear Statistical Inference.* Wiley, New York, 2 edition, 1973.

[45] Abel RodrÌguez, David B. Dunson, and Alan E Gelfand. The nested Dirichlet process. *Journal of the American Statistical Association*, 103:1131–1144, September 2008.

[46] J Sethuraman. A constructive definition of Dirichlet priors. *Statistica Sinica*, 4:639–650, 1994.

[47] J Sethuraman and Ram C Tiwari. Convergence of Dirichlet measures and the interpretation of their parameter. In S S Gupta and James O Berger, editors, *Statistical Decision Theory and Related Topics III*, volume 2, pages 305–315. Academic Press, New York, 1982.

[48] Ron Shamir, Roded Sharan, and Dekel Tsur. Cluster graph modification problems. *Discrete Applied Mathematics*, 144:173–182, 2004.

[49] Rohit Singh, Nathan Palmer, David Gifford, Bonnie Berger, and Ziv Bar-Joseph. Active learning for sampling in time-series experiments with application to gene expression analysis. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 832–839, New York, NY, USA, 2005. ACM Press.

[50] J. Z. Song, K. M. Duan, T. Ware, and M. Surette. The wavelet-based cluster analysis for temporal gene expression data. *EURASIP Journal on Bioinformatics and Systems Biology*, 2007:1–7, 2007.

[51] Paul T Spellman, Gavin Sherlock, Michael Q Zhang, Vishwanath R Iyer, Kirk Anders, Michael B Eisen, David Botstein, and Bruce Futcher. Comprehensive Identification of Cell Cycle-regulated Genes of the Yeast Saccharomyces cervisiae by Microarray Hybridization. *Molecular Biology of the Cell*, 9(12):3273–3297, 1998.

[52] Yee Whye Teh, Michael I Jordan, Matthew J Beal, and David M Blei. Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.

[53] Anbupalam Thalamuthu, Indranil Mukhopadhyay, Xiaojing Zheng, and George C. Tseng. Evaluation and comparison of gene clustering methods in microarray analysis. *Bioinformatics*, 22(19):2405–2412, 2006.

[54] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition.* Academic Press, 2 edition, 2003.

[55] D. Titterington, A. Smith, and U. Makov. *Statistical Analysis of Finite Mixture Distributions.* John Wiley & Sons, 1985.

[56] Martin J. Wainwright and Eero P. Simoncelli. Scale mixtures of Gaussians and the statistics of natural images. In *in Adv. Neural Information Processing Systems*, pages 855–861. MIT Press, 2000.

[57] Yan Wang and Jeremy M G Taylor. Jointly modeling longitudinal and event time data with application to acquired immunodeficiency syndrome. *Journal of the American Statistical Association*, 96(455):895–905, 2001.

[58] M West, P Müller, and M D Escobar. Hierarchical Priors and Mixture Models with Applications in Regression and Density Estimation. In A F M Smith and P R Freeman, editors, *Aspects of Uncertainty: A Tribute to D. V. Lindley*, pages 363–386. John Wiley and Sons: London, 1994.

[59] Samuel S Wilks. *Mathematical Statistics.* John Wiley and Sons, New York, 1962.

[60] Ming Yuan and Christina Kendziorski. Hidden Markov Models for Microarray Time Course Data in Multiple Biological Conditions. *Journal of the American Statistical Association*, 101(476):1323–1332, 2006.