# Dynamic Factor Graphs for Gene Regulatory Networks (DFG4GRN)

– Quick comparison box that has stats of each algorithm such as: number of parameters, data inputs (required/optional), recommended data size

## What it does:

A major problem with microarray data is measurement noise and data scarcity. The data that is used to infer gene regulatory networks comes from a small number of microarrays that sample a highly complex biochemical process (the evolution of the concentrations of mRNA over time) using relatively large time intervals. To combat noise, several replicate microarrays can be sampled at the same time point, and then techniques such as taking the median or mean of separate microarray replicates can be used. There are two problems with this approach: first, microarrays (or other approaches to measure expression) are currently quite expensive, and, second, data averaging reduces an already small dataset. The inferred regulatory network may be based on only a few replicates. What the Dynamic Factor Graphs for Gene Regulatory Networks (DFG4GRN) algorithm does is try to reduce this problem by using the noisy data to model the ideal data set: one that would be measured without noise. Inference is then done on this modeled ideal dataset instead of the noisy one. Predictions about the noisy dataset are made assuming a fitted Gaussian noise model.

In order to do this, we use a dynamical graphical model. A graphical representation is merely the representation of the structure within the data. In our case, this structure is unrolled in time, where the data at each time point is represented by a variable **y**(*t*). This variable corresponds to a node in the graph. In a dynamical graph, each node’s value, can be decided by some number of previous nodes. For example, if each node at time *t* is influenced by the node directly preceding it, that is the node from time *t*-1), then the graph is known as a “first Order Markov Chain”. We can also model the influence coming from many previous nodes: this is called an “*m*-Order Markov Chain”. Keeping this idea in mind, we can tweak this model a bit, so now instead of the value of each previous node deciding the value of the next node by some function *f*, we can instead model the dynamics of the system, that is, the change between values. In our case, this means that each node **y**(*t*) represents all of our expression values at time point *t*, and we want to come up with some function *f* that models the change in expression **y** between times *t* and *t*+1.

\*\* FIGURE OF MARKOV CHAIN \*\*

[Jesse: the trouble with widgets is that there can be seasonable effects, advertising effects etc. So, while I don’t “hate” it, I don’t love it either. What we want is an example where the causal factors are within the network (viz no advertising effects). That is, something where a network is really the right model and previous states affect current ones. If we can think of nothing better, then let’s use genes as we already have a simulator from the DREAM people.] We can think of a Markov chain as a way of predicting the current event based on *m* recent past events. For example, suppose that we are in charge of predicting how many widgets a store is going to sell today. We have at our disposal the number of widgets that have been sold on each day prior to today, going back several years. The number of widgets that the store sold nine months ago probably isn’t a good predictor of how many widgets will be sold today, as the product has probably grown or shrunk in popularity since that time. A better approximation can be found in how many widgets have been sold in the past few days. So, using only the data from the past two days, we can get a good idea of how many widgets will be sold today. In this way, we can build a model that gives us an idea of how the number of widgets sold in the previous N number of days affects the number of widgets sold on each day.

However, one problem with the above model is that it is susceptible to noise. For example, using our widget example, suppose we didn’t have the number of widgets actually sold each day, but, due to some faulty software in the registers, only an estimate of how many widgets are sold each day. This introduces a whole new problem, as now both the number of widgets and the amount of noise in the estimate are varying each day, making our model less accurate when predicting new data. DFG4GRN addresses this problem by modeling the idealized, that is, noiseless, version of the data. To do this, we replace the dynamical graphical model with only observed variables with a so-called state-space model, introducing what is called a “hidden state” to the model. These are represented in the figure below by the **z**(*t*) nodes. These hidden states represent the estimate of the noiseless version of the expression data.

The hidden states are then used to generate the observed expression values by incorporating measurement uncertainty. Although state space models can formalize any kind of functional mapping from the hidden states to the observed variables, the relationship between the observed states and the hidden (or latent) states in this DFG4GRN is assumed to be the hidden state’s value with added Gaussian noise.

So, we are now modeling two aspects of each gene over time. That is, for each observed value, we hypothesize that noise has been added to an idealized value, resulting in the observation. At the same time, we also model the dynamics between the hidden states as we move through time. We can now draw our state space like this:

\*\* DRAW PIOTR’S DFG4GRN mRNA STATE SPACE \*\*

In the figure, **y**(*t*) are our observed expression values (observed states), and **z**(*t*) are our idealized expression values (hidden states). The box between **y**(*t*) and **z**(*t*) represents the Gaussian noise used to calculate **z**(*t*) from **y**(*t*), and the box between **z**(*t*) and **z**(*t*+1) represents the function *f*. The function *f* must be learned.

To do this, we must first make an assumption about how the function *f* works. The assumption in DFG4GRN is that the hidden states behave like a first-order Markov chain, that is, each hidden state **z**(*t*) is dependent only on the value of the previous hidden state **z**(*t*-1). This means that each latent gene expression value is dependent on some linear combination of any number of the latent gene expression values at the previous time point. We can formulate *f* as a kinetic ordinary differential equation (ODE) involving a kinetic time constant τ, the rate of change of the expression values, mRNA degradation, and a linear function *f*i that is simply the transcription factor concentrations for gene *i*:

\*\* tau \* (dzi(*t*) / dt) + zi(*t*) = fi(**z**(t)) + etai(*t*) \*\*

If we linearize the above equation, we can see a bit better how each component influences the others.

\*\* Linearized version of above eq \*\*

Let’s start with the left hand side of the linearized equation. The parameter τ can be thought of as a weight of how much influence the difference in time between **z**(*t*) and **z**(*t*+1) should have. If τ is high, then the change in zi between *t* and *t*+1 is weighted more heavily than the expression value of zi at time *t*, and if τ is low, then the change in zi is not weighted as heavily as the value of zi at time *t*.

Moving to the right hand side of the linearized equation, we can now think of *f* as an *n* x *m* matrix **F**, where *m* is, in the case of genes, the number of transcription factors and n is the total number of genes, including transcription factors. In general, m is the number of possible influential factors and n is the number of possible targets of those influential factors (a target may be an influential factor). We will treat transcription factor genes as influential factors and all genes (including the transcription factors) as targets. This matrix **F** contains the weights of how each transcription factor affects each gene. Calculation of these weights is done in the learning step, which will be described shortly. So for each gene, we multiply the weight of each transcription factor Fi,j in the matrix **F** for that gene at time *t* by the latent expression of that gene, zj. We are essentially weighting the influence that each transcription factor has on each gene at a given time. We then add a bias factor *b* and our Gaussian error term, ηi. The bias factor *b* is a weight of the importance of each transcription factor, and that is another parameter that is learned. At each time point, each transcription factor also has a Gaussian error term associated with it. A key simplification is that the error term for each gene is uncorrelated with the error term of any other gene.

We can imagine this in more concrete terms by expanding the widget example. The company has recently expanded and now sells M number of products. Our job is to still predict the number of widgets that will be sold today, but now we have some more data to work with. If two products tend to be sold together, we can use that data to get better predictions of how many of each will be sold today. For example, suppose that the number of widgets W at time *t* that we sold each day can be approximated by some weighted combination of products P1, P2, P3, and the value of W at the previous time point, *t*-1. Further, we know that either P1  and P2 are necessary for P3 to work, but P1 and P2 operate just fine without P3. Thus, we can think of P1 and P2 as transcription factors, and P3 as a target gene. The amount of P3 that is sold is directly influenced by how many P1 and P2 are sold, but the reverse is not true. Also remember that the sales data for the widgets has noise in it, so what we want to do is to model this noise in order to obtain our estimate of the actual value, then use these estimated actual values to model the interactions between widgets. These are values that must be learned. [I do like the concreteness of widgets. Let’s think more]

**LEARNING**

Previously, it was stated that the model of our observed variables was simply the value of gene *y*i at time *t* plus some Gaussian noise. We'll define this model now as **H**(*t*), where **H**(*t*) is an *n* x *n* diagonal matrix where each value **H**(*t*)**i***,i* is simply the value of *y*i(*t*) plus some Gaussian error term. More formally:

**H**i,i = *y*i(*t*) + epsi(*t*)

During learning, we will want to treat our dynamical and observational models in the same way, so we can combine then into a single matrix **W** = [**F**, **H**]. DFG4GRN learns using a modified Expectation-Maximization algorithm. We want to adjust the parameters in **W** in order to minimize the loss L(**W**,Y,Z).

L(**W**,Y,Z) = E(**W**,Y) + Rz(Z) + R(**W**)  
 Z\_tilde = argminzL(**W**\_tilde, **Y**, Z)

**W**\_tilde = argminwL(**W**, **Y**, Z\_tilde)

**R**(**F**) is a regularization term on the weights in **W**, and Rz(Z) are constraints on the latent (estimated actual) variables Z. We can think of equation ##Z\_tilde## as the E-step, where we hold the weights in **W** in place and relax ours states, Z. Relaxation in this step is performed by minimizing the sum of the quadratic error of **W** in the above loss equation by freezing **W** in place and performing a gradient descent on our latent variables, Z. Equation ##W\_tilde## can be thought of as the M-step, where we hold our state in Z in place, and then relax our weight matrix **W**. Relaxation for equation ##W\_tilde## is performed by minimizing the sum of the quadratic errors of only the dynamical model **F** using LARS optimization (further explained in section ##), while holding **Z** steady,

--Widget related explanation of learning

--Intuitions behind Gaussian Noise

--Selection of network by bootstrapping

--Hyperparameter explanation / selection tips

–Continuing example (10 node network), results with different hyperparameters

--When to use

\*\* TODO: Learning step. Formalization of observation model. Explanation of Gaussian noise and how we are using it to influence the data between Y and Z, intuitions behind it.