Speeding Up Gibbs Sampling in Probabilistic Optical Flow

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Dongzhen Piao

B.S., Electrical Engineering, Tsinghua University
M.S., Electrical Engineering, Columbia University

Carnegie Mellon University
Pittsburgh, PA

December, 2014

Keywords: Gibbs Sampling, Optical Flow, Parallelization, Speed Up, Probabilistic Graphical Model, Markov Random Field, Simulated Annealing

Acknowledgments

One day while I was still writing the thesis, me and my labmate had this short conversation: "How much have you written?" "About 20 pages." "How soon are you expecting to finish it?" "About one month." "Wow, that's a really short time considering how long you have been in the program!"

Indeed, compared to the years-lasting doctorate program, the time spent on writing up a thesis is almost negligible. However, if there was not such time spent in the long running program, if I did not build up enough knowledge or worked on enough projects, I would never be able to complete a thesis in such a short period. If thesis writing is a stage performance, it is the long time spent training oneself that makes any stage performance look easy.

Numerous people generously gave their helping hands throughout this journey. They guided me through the days when I lost directions; They comforted me when I was frustrated by no-progress; They shared their sincere happiness when I made any tiny achievement. These are the people that really made my journey possible. I would like to take this opportunity to show them my most sincere thankfulness.

The very first person, whom I would like to give my deepest appreciation to, is my advisor, Professor Ole J. Mengshoel. Without him, I could not have done all this. He gave me the call four years ago when I applied for CMU, discussed ideas about machine learning and its broad applications, and led me into the door of this prestigious school. He met with me multiple times a week to discuss new ideas on projects, and provided detailed feedback on any of my questions or progress. He spent hours going over my draft papers for conferences and presentations, and had no problem writing more materials than I had on my drafts to correct my errors and suggest better write-ups. Discussions with him were always enjoyable and filled with good laughs - He could always find the interesting things hidden in conversations and bring them out to me, and I almost feel I took a separate course about American humor, and best of all, it's free. Thank you, for all the time you spent guiding me. It is a great honor to be advised by you during

this journey.

Professor Ying Joy Zhang is another faculty member I feel deeply connected with. He and Prof. Mengshoel called me together at time of my application to CMU, and when I flew from New York to California to visit this wild-west campus, he offered me a tour, fed me with food, and discussed what it will be like to study here, particularly in the machine learning field. His research projects are quite application focused, and his abundant knowledge in applied machine learning really opened up my eyes in the course he offered.

Also, throughout the course of the doctoral program, I was able to learn from many awesome instructors and researchers, and was lucky enough to have collaborated with some of them. In the course of "How to write fast code", Professor Jike Chong and Ian Lane's introduction to the high performance computing platforms opened a door to a completely new research direction for me, and it provided an important foundation to my thesis research. Also, thank you for providing me the chance to present my course project in Silicon Valley High Performance Computing group, which is a very rare opportunity for a student just heading into the field - I am completely honored. In the machine learning course, I was able to collaborate with Dr. Rong Yan on a research project around social media prediction, which not only broadened my perspective in machine learning and social media, but also trained my engineering skills in big data and data mining. In the course of "Probabilistic Graphical Model" offered by Professor Eric Xing, he presented the giant and cutting edge research area of probabilistic graphical models, which was immensely helpful in my own research and is another important foundation in my thesis work. In the course of "Machine learning with large datasets", Professor William Cohen generously offered his knowledge in running machine learning algorithms in large scale and distributed setting, which is one of the most important task in todays' machine learning research and is immensely helpful in my research projects and thesis work.

I was lucky to have some of the people I respected the most as my thesis committee board members. Prof. Mengshoel, my advisor, serves as the chair of the committee. The other members are Prof. Joy Zhang, Prof. Jike Chong, and Branislav Kveton.

During my proposal of thesis topic, they kindly gave lots of feedback and guidance to my proposal. I can't thank you enough for your time and effort put in making my thesis writing and proposal such a smooth process.

I would also like to thank my qualification exam committee members for rigorously testing my knowledge in the field. The members include Professor Marios Savvides, Professor Daniel Siewiorek and Prof. Joy Zhang. Especially with Prof. Daniel Siewiorek, thank you for sending me the pointer to formal method of analyzing parallel computation after my qualification exam, which was very helpful in my thesis work.

As a non-native English speaker, I received immense personal instruction from Jennifer Wolfeld. She coached me, hand by hand, on how to become a better English speaker, and how to do better public presentation. I am completely astonished by the level of details she reached in guiding me, and that helped me become a much better presenter. It's a wealth I could use for my entire life. Thank you!

Except for the faculty members, I also received lots of help from many staff members. Thank you, Gerry Elizondo, for your ability to solve any of my program related problems. I will always miss your smile when I walk into your office and open with "I have a problem...". In the Pittsburgh campus, Samantha Goldstein and Elaine Lawrence offered equally much, if not more help. I also appreciated the help and collaboration with Wendy Fong, Stacy Marshall and Hillary Nicholson.

I am very grateful that I have some of the most helpful colleagues - Lu Zheng, Priya Sundararajan and Brian Ricks. I always had a lot of questions and weird ideas that I need to consult with them, and they never disappointed me by providing their insights into my question in doubt. My research direction has the most overlap with Lu's, so when I had any problems with graphical models or machine learning in general, I turned to him for help. Lu also possessed knowledge in many other fields, and I learnt a lot from him. When Lu graduated, Brian succeeded Lu's role as my chief consultancy officer, and discussions with Brian are always joyful and inspiring. I could never forget those numerous nights I spent with Brian in the lab. You are the reason I could stay so late in the lab, and do the things that really mattered.

I collaborated with many students in various course and research projects. I worked

with Jiang Zhu and Huan-Kai Peng (Pumbaa) on a twitter retweet prediction project (advised by Dr. Rong Yan), and it is one of the most exhausting but fruitful projects I did in CMU. With Avneesh Saluja and Mahdi Pakdaman, together with TA advisor Ankur Parikh, we worked on dynamically determining topic numbers for movie recommendation. With Yogesh Dalal, we studied and compared the characteristics of Hadoop and GraphLab platform. With Xianheng Ma, Erik Reed, Priya Sundararajan and Briana Johnson, we worked on various projects within Prof. Mengshoel's research group. I must thank you all for having me as your partner, spending days and weeks sweating on the projects and cracking down the tough problems.

A number of people lent their help in my research. Yucheng Low and Joseph Gonzalez, creators of GraphLab, generously answered my questions on usage of GraphLab and parallel Gibbs sampling, which was critical in my qualification exam and is foundational in my thesis work.

I also owe my appreciations to my colleagues in CMU Silicon Valley - Frank Mokaya, Zheng Sun, Le Nguyen, Eric Chen, Yuseung Kim, Yuan Tian, Bing Liu, Akshay Chandrashekaran, Erik Reed, Ming Zeng, Rahul Rajan, Xiao Wang, David Cohen, David Huang, Shijia Pan, Song Luan, Tong Yu, Senaka Buthpitiya, Jungsuk Kim, Aniruddha Basak, William Chan, Heng-Tze Cheng, Vadim Zaliva, Wonkyum Lee, Paul Maergner, Bruce Debruhl, Aveek Purohit, among many other students here. You guys are the reason I had such a great memory working in the campus and enjoying PhD life. Thank you for having those conversations, whether in-depth discussions about technology, theory, startup, or those more lightweight ones such as the awesomeness of being a PhD, or foods and cups. They are truly inspirational. Thank you for having me in those numerous extracurricular activities that added a lot more fun to the research life.

My parents are the reason I was able to come to US for the doctoral education, and they are the people I should thank most. It's not easy to see your kid fly to a place in the opposite side of the earth and deal with people who speak a completely different language, but they firmly supported my journey to US, and spent almost half their savings on my expenses, so that I have a chance to pursue a better career. I am proud that I have parents like you.

My wife Zixuan Zhang has been my firmest supporter throughout all these years. Back in the day when she was still my girlfriend and both of us were in New York, she supported my move to CMU Silicon Valley to pursue higher education and chase my dream. We both knew that it means we wouldn't be able to stay together for the next several years, but she was still supportive and never complained about my decision. I am so grateful to have you as my wife. Though we have a continent's worth of distance whenever we want to meet, and three hours of time difference when we want to talk, but I will always love you no matter what is separating us.

In the end, credits must be given to all the fundings that sponsored my research. I was supported by Carnegie Mellon University Dean's Fellowship in my first year of study. I also received support from National Science Foundation (NSF) and Advanced Research Projects Agency - Energy (ARPA-E). The optical flow project is funded by Carnegie Mellon University - Sun Yat-sen University (CMU-SYSU) Collaborative Innovation Research Center (CIRC) and Sun Yat-sen University - Carnegie Mellon University (SYSU-CMU) International Joint Institute of Engineering (JIE). John Bertucci and Claire Bertucci generously offered Bertucci Graduate Fellowship to me that funded my research. Thank you all for the generous support that made all these projects and research activities a reality.

Abstract

In today's machine learning research, probabilistic graphical models are used extensively to model complicated systems with uncertainty, to help understanding of the problems, and to help inference and predict unknown events. For inference tasks, exact inference methods such as junction tree algorithms exist, but they suffer from exponential growth of cluster size and thus is not able to handle large and highly connected graphs. Approximate inference methods do not try to find exact probabilities, but rather give results that improve as algorithm runs. Gibbs sampling, as one of the approximate inference methods, has gained lots of traction and is used extensively in inference tasks, due to its ease of understanding and implementation.

However, as problem size grows, even the faster algorithm needs a speed boost to meet application requirement. The number of variables in an application graphical model can range from tens of thousands to billions, depending on problem domain. The original sequential Gibbs sampling may not return satisfactory result in limited time.

Thus, in this thesis, we investigate in ways to speed up Gibbs sampling. We will study ways to do better initialization, blocking variables to be sampled together, as well as using simulated annealing. These are the methods that modifies the algorithm itself.

We will also investigate in ways to parallelize the algorithm. An algorithm is parallelizable if some steps do not depend on other steps, and we will find out such dependency in Gibbs sampling. We will discuss how the choice of different hardware and software architecture will affect the parallelization result.

We will use optical flow problem as an example to demonstrate the various speed up methods we investigated. An optical flow method tries to find out the movements of small image patches between two images in a temporal sequence. We demonstrate how we can model it using probabilistic graphical model, and solve it using Gibbs sampling. The result of using sequential Gibbs sampling is demonstrated, with comparisons from using various speed up methods and other optical flow methods.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

We live in a world filled with uncertainties. A certain event E may happen with a probability P_1 given certain condition C_1, and given another condition C_2, it may happen with a different probability P_2. For example, when the weather is cloudy, the chance of rain is higher, and when it's sunny, the chance of rain is lower.

Probability theory studies how uncertain events affect each other. For the above example, there are two random variables, C representing the weather condition, and E representing raining event. We can compactly represent the relationship as $P = P(E|C)$, which shows the probability of rain event E conditioned on ther weather condition C. For example, $P(E = rain|C = sunny)$ shows the probability of a rain when the weather is sunny.

In this simple example, there are only two random variables, E and C. For some complicated event model, there can be many more random variables. Representing the entire probabilistic relation among all these many random variables can be difficult, and thus Probabilistic Graphical Model (PGM) are introduced to simplify the matter. Using a PGM, we use a graph to store the random variables and probabilities. The nodes (vertices) of the graph represent random variables, the edges represent relationship, and additionally, the probabilities associated with a group of random variables are stored in the form of factors.

One of the problems we are interested in, after having a PGM representation of a complicated event, is to get samples from the entire random variable collection. With the samples, we can find

out rough characteristics of the probability distribution, for example, get the expectation of some function that depends on these variables. There are many methods for sampling from probability distributions, such as uniform sampling, importance sampling, rejection sampling, etc. When the dimension (*i.e.*, number of random variables) of the probability distribution is high, many of these sampling methods perform poorly.

Gibbs sampling is one of the sampling methods that allow us to deal with high dimensional sampling easily. It goes as follows: First, all variables are initialized randomly. Then, it samples one random variable at a time, using information available from random variables in its Markov blanket, given which the random variable is conditionally independent from all other random variables. It does this sampling for each variable in the PGM, sequentially, which marks one round of Gibbs sampling. This procedure is executed multiple rounds, until the distribution of samples approach the real underlying distribution.

Though Gibbs sampling is faster than many other sampling algorithms, we still want it to be faster, to cope with problems that either have a large number of random variables, or require fast computation. Here are a few directions to improve it:

- The original Gibbs sampling algorithm is sequential. We can change the algorithm so that it can be parallelized. Then we can use many parallel computing platforms to speed up sampling.
- Random variables are sampled one at a time. Sometimes it's better to sample a group of random variables together, when they have a strong correlation.
- The initial sample is traditionally chosen randomly. We can try to use better initialization, so that the samples approach underlying distribution faster.

An interesting problem that we have successfully applied Gibbs sampling and various speed-up optimizations, is the optical flow problem. Optical flow algorithms try to find out the movements of small image patches or pixels between two consecutive images in a sequence. The result of optical flow can be used in areas such as image segmentation, video compression and object recognition. Many optical flow methods have been developed, such as differential methods (Horn-Schunck [20] and Lucas-Kanade [32]), region-based matching, frequency-based methods, and phase-based meth-

ods.

One key assumption in most existing optical flow methods is the invariance of pixel intensity, meaning that the intensity of each pixel in the original image should be the same as the intensity of the corresponding pixel in the moved image. This assumption will not hold in many real world situations due to changing illumination conditions or changing object surface direction. In these circumstances, algorithms that assume invariant pixel intensity often yield poor results.

We present a new approach of modeling and solving optical flow problems, using probabilistic graphical models. We will first discuss the intuitions that guide the design of the model, then we build the model using Markov random field. We show that our method is able to handle changing pixel intensities.

The rest of the thesis is organized as follows. In Chapter 2, we walk through several preliminary concepts, *e.g.*, optical flow, probabilistic graphical model and Gibbs sampling. These concepts will be the foundation of the entire thesis. In Chapter 3, we present how we use probabilistic graphical model and Gibbs sampling to model and solve optical flow problem. In Chapter 4, we discuss ways to speed up Gibbs sampling. Discussions of parallelizing Gibbs sampling are in Chapter 5. In Chapter 6, we report on experiments using various speed-up methods, and compare results. We come to a conclusion in Chapter 7, and discuss future work.

Chapter 2

Preliminaries

In this chapter, we describe several fundamental concepts that are used throughout the thesis. We use bold typefaced letters (such as $\boldsymbol{X}$) to represent a vector or a collection, and normal ones (such as X) to represent a scalar or one element from a collection. We use uppercase to represent random variable (such as X), and lowercase to represent an instantiation of that variable (such as x).

2.1 Probabilistic Graphical Model

2.1.1 Motivation

In probability theory, we use a random variable, such as X, to denote a event that has random possible outcomes, or states. The random outcome can be discrete, such as when tossing a coin. In this case, there are two possible results: Head or tail. We use $P(X)$ to denote the probability of possible outcomes. In this case, the probability of having a head or tail is equal (assuming a fair coin), which we can describe as $P(X = head) = 0.5, P(X = tail) = 0.5$. The sum of all possible outcomes' probabilities should be equal to 1.

The random variable can also be continous. For example, when a person is throwing a coin away from him, the distance of the throw is continous. In this case, the probability of the distance being exactly same as any value x is infinitely small. But we can still do an integration on a range of distances. The integration on the entire possible range will be 1, $\int P(X)\,\mathrm{d}X = 1$.

In some situations, the outcome of an event is dependent on another event. Let's denote R as a raining event, which has two possiblities: true and false. It depends on the current weather, denoted as W. If $W = \text{sunny}$, then there is a low chance of raining, which we can represent as a **conditional probability**: $P(R = \text{true}|W = \text{sunny}) = 0.1$, $P(R = \text{false}|W = \text{sunny}) = 0.9$. When $W = \text{cloudy}$, then it's more likely to rain, which can be represented as $P(R = \text{true}|W = \text{cloudy}) = 0.4$, $P(R = \text{false}|W = \text{cloudy}) = 0.6$.

In this simple model, there are two random variables: W and R. R depends on W, and W depends on nothing. Assuming that sunny and cloudy weather are equally likely, we can say $P(W = \text{sunny}) = 0.5$, $P(W = \text{cloudy}) = 0.5$.

What is the chance of the weather being sunny with a rain? It comes down to the **joint probability** of W and R, denoted as $P(W, R)$. The chance of both a sunny and raining day, will be $P(W = \text{sunny}, R = \text{true}) = P(R = \text{true}|W = \text{sunny}) \cdot P(W = \text{sunny}) = 0.1 \cdot 0.5 = 0.05$, which understandably is quite low. Here, the joint probability is broken down to multiplication of a conditional probability, as well as a standalone probability.

Another question is, what is the probability of a raining event, regardless of weather condition? It is the **marginal probability** $P(R)$. It can be computed as follows:

$$\begin{aligned} P(R = \text{true}) &= \sum_W P(R = \text{true}, W) \\ &= \sum_W P(R = \text{true}|W)P(W) \\ &= P(R = \text{true}|W = \text{sunny})P(W = \text{sunny}) + \\ &\quad P(R = \text{true}|W = \text{cloudy})P(W = \text{cloudy}) \\ &= 0.1 * 0.5 + 0.4 * 0.5 = 0.25 \end{aligned}$$

It means, if we did not know anything about current weather condition, the chance of having a rain is 25%.

The examples we have used so far have at most two random variables. In more complicated models, the number of random variables can be quite large.

Think about a case, where we are trying to find out whether a student will have a happy life

H, together with the following evidences: His intelligence I, course difficulty D, course grade G, reference letter from professor L, SAT score S, his job status J. The entire joint distribution can be written as $P(H, D, I, G, S, L, J)$.

Now, to compute the marginal probability of H under varying evidence, we will need to store all combination of the evidences involved. Assuming there are 2 different states for each of the random variable, the number of combinations will be $2^6 * (2 - 1) = 64$, since there are 6 random variables to be marginalized, and we only need to know the probability of H being true (thus the "$2 - 1$" term). When there are more random variables in a model, the number of values to store grows exponentially with the number of random variables, and it quickly becomes a big problem.

There are better ways to store the probabilities. With careful observation of the random variables, we find that some of them are dependent on each other. Here's a list of reasonable dependencies:

- Course grade G depends on course difficulty D and intelligence I
- SAT score S depends on intelligence I
- Reference letter L depends on course grade G
- Job status J depends on reference letter L and SAT score S
- Happiness H depends on his course grade G and job status J

With these dependencies, we can decompose the original joint probability into smaller terms:

$$P(H, D, I, G, S, L, J) = P(H|G, J)P(J|L, S)P(L|G)P(G|D, I)P(S|I)P(D)P(I) \tag{2.1}$$

Now, to find out the probability of the student being happy, we only need to store the individual probabilities of the smaller terms, and multiply them together. There are 3 terms with 3 random variables: $P(H|G, J), P(J|L, S), P(G|D, I)$, 2 terms with 2 random variables: $P(L|G), P(S|I)$, and two random variables that have no dependencies: $P(D)$ and $P(I)$. The total number of values to store is thus $3*2*2+2*2+2*1 = 18$, which is much better than the original 64. Using this method, the number of values to store grows linearly with the number of random variables (assuming the joint probability can be perfectly broken down to smaller terms), rather than exponentially.

2.1.2 Probabilistic Graphical Model

As shown, knowing the local dependencies among random variables can be very helpful in storing the entire probability distribution. It would be even better, if we could visualize this kind of local dependencies. This is where **Probabilistic Graphical Model (PGM)** comes into play [29]. Figure 2.1 shows a PGM representation of the student probability distribution.

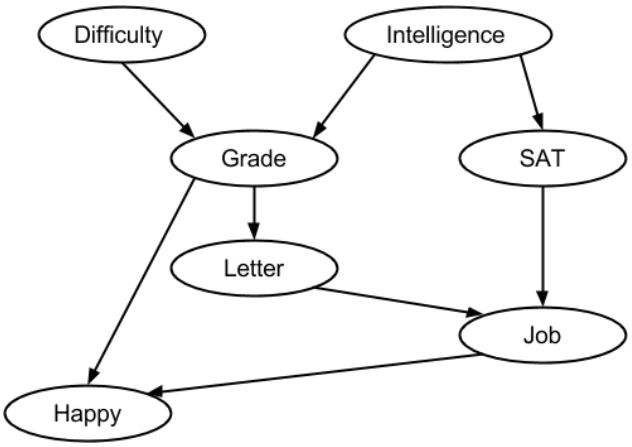


Figure 2.1: Probabilistic graphical model representation of student probability distribution

In this PGM, the nodes represent random variables, and directed edges represent dependencies. We were able to compactly represent the dependencies discussed earlier into this graph.

When edges are directed, as in this example, we call the PGM a **Bayesian network** [23, 38]. The directed edges often (but not necessarily) represent causal relationships.

There are cases where there are no clear causal relationships, and it is hard to decide which direction the edge should point to. For example, consider the image denoising PGM model in Figure 2.2.

Here, the dark nodes represent pixels in a noisy image we observed (denoted as Y_i), and the white nodes represent the de-noised pixel values we are trying to uncover (denoted as X_i). For X_i, we connect neighboring nodes, since usually a pixel's value is influenced by its neighboring pixels. For Y_i, since their values are already known, there is no influence from one to another and there is no need to connect them. Each Y_i is connected to corresponding X_i to show the impact of observed pixel to hidden pixel. This kind of undirected PGM is called **Markov random field** [26, 30].

In a PGM, except for nodes (representing random variables) and edges (representing dependen-

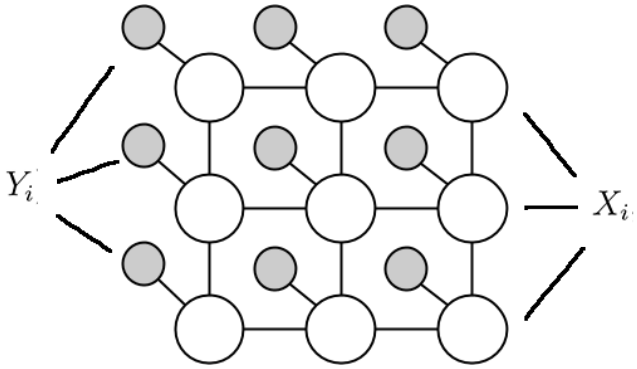


Figure 2.2: Probabilistic graphical model representation of image denoising model

cies), we also need to specify how much the random variables depend on each other. We call them **factors**. A factor, denoted as $\phi(\boldsymbol{X}_\phi)$, quantifies the relative likelihoods of all possible states for a subset of random variables $\boldsymbol{X}_\phi \subseteq \boldsymbol{X}$. A factor may have only one variable, or it may contain the entire set of variables in the graph.

For a Bayesian network, each factor is defined as the conditional probability of a child node conditioned on its parents, $\phi_i(\boldsymbol{X}_{\phi_i}) = P(X_i|\boldsymbol{X}_{PA_i})$, where $\boldsymbol{X}_{PA_i}$ is the parent set of X_i. For example, in Figure 2.1, there is a dependency between grade G, difficulty D and intelligence I, and we can represent this kind of dependency as a factor $\phi(G, D, I) = P(G|D, I)$. For node that doesn't have a parent, such as D, the factor will contain only one node, $\phi(D) = P(D)$.

In an MRF, a factor does not have a concrete meaning as in a Bayesian network, and it merely serves as an indicator of relative likelihood of different states of nodes in that factor. For example, in a simple four-node MRF shown in Figure 2.3, there is no causal dependency among nodes (such as between A and B) like in a Bayesian network. In this case, the factor simply shows relative likelihood. For $\phi(A, B)$, a possible factor table is shown in Table 2.1. It shows that an assignment (a', b') is 100 times more likely than (a, b).

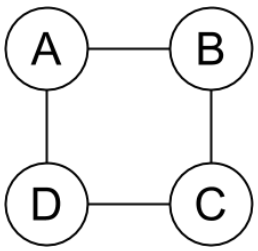


Figure 2.3: Simple four node MRF

A	B	$\phi(A,B)$
a	b	1
a	b'	10
a'	b	10
a'	b'	100

Table 2.1: Possible factor table for $\phi(A, B)$

With factors introduced, now we can formally define a PGM as $G(\boldsymbol{X}, \boldsymbol{E}, \boldsymbol{\phi})$ - It includes the node set $\boldsymbol{X}$, the edge set $\boldsymbol{E}$ and the factor set $\boldsymbol{\phi}$.

Also with factors, the joint probability of the entire graph can be represented as

$$P(\boldsymbol{X}) = \frac{1}{Z} \prod_i \phi_i(\boldsymbol{X}_{\phi_i}) \tag{2.2}$$

Here Z is normalization term,

$$Z = \sum_{\boldsymbol{X}} \prod_i \phi_i(\boldsymbol{X}_{\phi_i})$$

For a Bayesian network, since the factors are defined as conditional probabilities,

$$P(\boldsymbol{X}) = \prod_i \phi_i(\boldsymbol{X}_{\phi_i}) = \prod_i P(X_i | \boldsymbol{X}_{PA_i}) \tag{2.3}$$

2.1.3 Markov Blanket

Given a PGM as in Figure 2.1 and Figure 2.2, one question we want to answer is the probability of an individual random variable. For example, in Figure 2.1, what is the probability of happiness H, given all other variables? We can compute it as follows:

$$\begin{aligned} P(H|D,I,G,S,L,J) &= P(H,D,I,G,S,L,J)/P(D,I,G,S,L,J) \\ &= P(H,D,I,G,S,L,J)/\sum_H P(H,D,I,G,S,L,J) \end{aligned}$$

Using the decomposition found in Equation 2.1, we can further simplify the equation:

$$\begin{aligned} P(H|D,I,G,S,L,J) &= P(H,D,I,G,S,L,J)/\sum_H P(H,D,I,G,S,L,J) \\ &= \frac{P(H|G,J)P(J|L,S)P(L|G)P(G|D,I)P(S|I)P(D)P(I)}{\sum_H P(H|G,J)P(J|L,S)P(L|G)P(G|D,I)P(S|I)P(D)P(I)} \\ &= \frac{P(H|G,J)P(J|L,S)P(L|G)P(G|D,I)P(S|I)P(D)P(I)}{P(J|L,S)P(L|G)P(G|D,I)P(S|I)P(D)P(I)\sum_H P(H|G,J)} \\ &= P(H|G,J) \end{aligned}$$

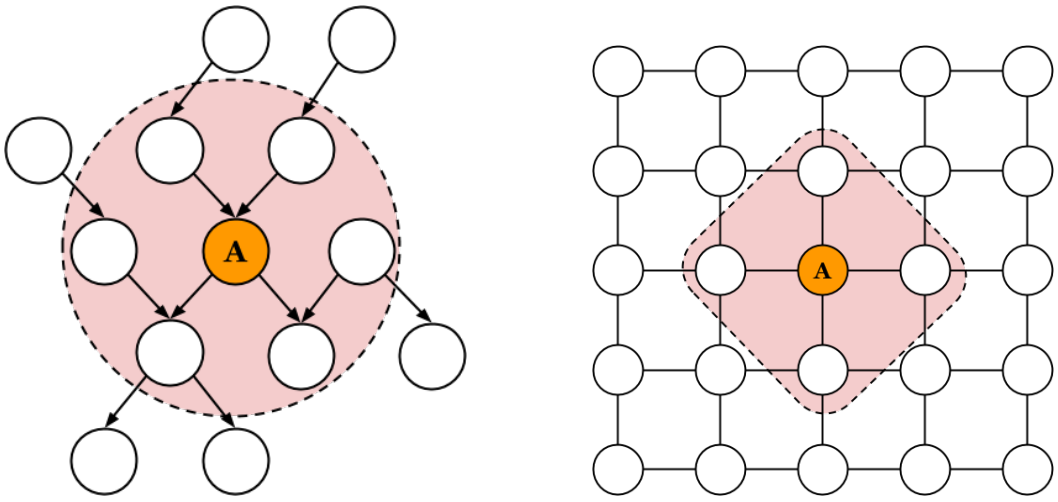
The conclusion is, as long as we know the values for G and J, we can determine the probability of H, and do not care about all other variables. In other words, H is **conditionally independent** from all other random variables, given G and J.

In a PGM with a random variable set $\boldsymbol{X}$, for a random variable X, the minimal set of random variables $\boldsymbol{X}_{MB_X}$ given which X is conditionally independent from all other random variables $\boldsymbol{X} - \{X\} - \boldsymbol{X}_{MB_X}$, is called the **Markov blanket** of random variable X. In other words, $\boldsymbol{X}_{MB_X}$ is the minimal set of nodes to make the following equation hold:

$$P(X|\boldsymbol{X}_{MB_X}) = P(X|\boldsymbol{X}_{MB_X}, \boldsymbol{X} - \{X\} - \boldsymbol{X}_{MB_X})$$

In the student probability example in Figure 2.1, the Markov blanket for H is thus $\{G, J\}$.

In a Bayesian network, the Markov blanket of any arbitrary random variable consists of its parents, its immediate decendents, as well the parents of its immediate decendents. It can be visualized as in Figure 2.4a. In a Markov random field, the Markov blanket of a random variable consists of its immediate neighbors, as shown in Figure 2.4b.



(a) Bayesian Network **(b)** Markov Random Field

Figure 2.4: Markov blanket for node A in a PGM

2.1.4 Inference in PGM

With a PGM, one of the task we wish to perform is inference. **Inference** means computing the probability distribution of a set of variables $\boldsymbol{Y} \subseteq \boldsymbol{X}$, given certain conditions $\boldsymbol{E} = \boldsymbol{e}$, *i.e.*, find $P(\boldsymbol{Y}|\boldsymbol{E} = \boldsymbol{e})$, where $\boldsymbol{Y} \subseteq \boldsymbol{X}, \boldsymbol{E} \subset \boldsymbol{X}$.

The most straightforward way of computing above is the following. Assuming $\boldsymbol{W} = \boldsymbol{X} - \boldsymbol{Y} - \boldsymbol{E}$,

$$P(\boldsymbol{Y}|\boldsymbol{E} = \boldsymbol{e}) = \frac{P(\boldsymbol{Y}, \boldsymbol{E} = \boldsymbol{e})}{P(\boldsymbol{E} = \boldsymbol{e})} = \frac{\sum_{\boldsymbol{W}} P(\boldsymbol{Y}, \boldsymbol{W}, \boldsymbol{E} = \boldsymbol{e})}{\sum_{\boldsymbol{Y},\boldsymbol{W}} P(\boldsymbol{Y}, \boldsymbol{W}, \boldsymbol{E} = \boldsymbol{e})}$$

However, when the number of random variables grows, the number of summation grows exponentially. For example, if the joint distribution $P(\boldsymbol{X})$ has 100 variables in $\boldsymbol{X}$, each with 2 states, and there are 10 variables in $\boldsymbol{W}$, then the summation of $\sum_{\boldsymbol{W}} P(\boldsymbol{Y}, \boldsymbol{W}, \boldsymbol{E} = \boldsymbol{e})$ involves $2^{100} - 2^{100-10} \approx 1.27 \times 10^{30}$ summations, making it infeasible to do the bruteforce inference.

Many algorithms have been proposed to solve the problem. They can be roughly put into two categories: **Exact inference methods** and **approximate inference methods**.

For exact inference methods, the outcome of the inference is exact. Methods include **variable elimination** [7, 49], **junction tree algorithm** [24, 31, 42], among others. Running these methods involve doing summation of factor tables of dependent random variables. It's much better than the

bruteforce method shown above, because we don't need to sum the entire probability distribution. However, as we will show in Section 2.2, the runtime of an exact inference algorithm is exponential to the **treewidth** of the graph, making it infeasible for graphs with large treewidth.

Approximate inference methods do not try to solve the problem exactly, instead they give results that are close to exact results. The runtime of approximate methods are typically much shorter than exact methods. Though they don't give exact solution, most of the time approximate results are good enough for problems of our interests. Methods of approximate inference include sampling methods such as **Gibbs sampling** [17], **variational methods** [25], among others. A detailed description of Gibbs sampling method is presented in Section 2.3.

2.2 Junction Tree Algorithm

Given a PGM $G(\boldsymbol{X}, \boldsymbol{E}, \boldsymbol{\phi})$ whose joint distribution is given as $P(\boldsymbol{X}) = \frac{1}{Z}\prod_i \phi_i(\boldsymbol{X}_{\phi_i})$, we would like to answer queries such as computing marginal probability of $P(X_i)$, or conditional probability given some evidence $P(X_i|X_j = x_j)$. **Junction Tree Algorithm (JTA)** [24, 31, 42], being one of the exact inference algorithms, can be used to answer such queries. It first converts the given PGM G into a **clique tree** $\mathcal{T}$, whose nodes are cliques $\{C_i\}$, each consisting of a subset of random variables in G: $\boldsymbol{X}_{C_i} \subseteq \boldsymbol{X}$. It will then "caliberate" the clique tree $\mathcal{T}$ to find out marginal probabilities of variables in each clique C_i. The detailed procedure of JTA is presented in Algorithm 2.1.

First, the input PGM is **moralized** if it is a Bayesian network. Moralization involves connecting all parents of a node, and removing edge directions.

Then, the undirected graph is **triangulated** so that no cycles of length bigger than 3 is chordless. A **cycle** is a closed path in a graph with no repeated vertices other than the starting and ending vertice. A **chordless cycle** is a cycle where no non-adjacent vertices on the cycle are joined by an edge.

Then, from the triangulated graph, we **find maximal cliques**. A **clique** is a group of nodes where every pair of nodes is connected by an edge. A **maximal clique** in a graph is a clique where no extra node in the graph can be added to the clique and still remain as a clique.

Algorithm 2.1 JTA: Junction Tree Algorithm

Input: PGM $G(\boldsymbol{X}, \boldsymbol{E}, \boldsymbol{\phi})$

Output: Junction tree $\mathcal{T}$, caliberated cliques $\{\beta_i\}$

1. Generate moralized graph

$G' \leftarrow G$

if G' is a Bayesian network **then**

In G', for each node X_i, connect each pair of its parents $\boldsymbol{X}_{PA_i}$

Make G' undirected

end if

2. Generate triangulated graph

$G'' \leftarrow G'$

If there exists chordless cycles in G'' with more than 3 nodes, add an edge to each of the cycles to make them chordal.

3. Find maximal cliques

In G'', find maximal cliques $\boldsymbol{C}_{MAX} \equiv \{C_i\}$.

4. Generate clique tree

$\mathcal{T} \leftarrow$ empty graph. Let $\mathcal{T}$'s node set $\boldsymbol{C}_{\mathcal{T}} \leftarrow \boldsymbol{C}_{MAX}$ and edge set $\boldsymbol{E}_{\mathcal{T}} \leftarrow \varnothing$.

For each clique C_i initialize clique value: $\psi_i(\boldsymbol{X}_{C_i}) \leftarrow \prod_{j:\boldsymbol{X}_{\phi_j} \subseteq \boldsymbol{X}_{C_i}} \phi_j(\boldsymbol{X}_{\phi_j})$.

For each pair of cliques (C_i, C_j), add undirected edge (i, j) to edge set $\boldsymbol{E}_{\mathcal{T}}$. Find the edge weight $w_{i,j} \leftarrow \left|\boldsymbol{X}_{C_i} \cap \boldsymbol{X}_{C_j}\right|$.

Find maximum spanning tree (MST) of $\mathcal{T}$, remove all other edges.

5. Caliberate clique tree

Initialize message set $\Delta \leftarrow \varnothing$

while $\exists i, j$ such that $\forall k \in NB_i - \{j\}, \delta_{k \to i} \in \Delta$ **AND** $\delta_{i \to j} \notin \Delta$ **do**

if $NB_i = \{j\}$ **then**

$\delta_{i \to j} \leftarrow \sum_{\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_j}} \psi_i(\boldsymbol{X}_{C_i})$

else

$\delta_{i \to j} \leftarrow \sum_{\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_j}} \left\{\psi_i(\boldsymbol{X}_{C_i}) \prod_{k \in NB_i - \{j\}} \delta_{k \to i}\right\}$

end if

$\Delta \leftarrow \Delta + \{\delta_{i \to j}\}$

end while

for all $C_i \in \boldsymbol{C}_{\mathcal{T}}$ **do**

$\beta_i \leftarrow \psi_i(\boldsymbol{X}_{C_i}) \prod_{k \in NB_i} \delta_{k \to i}$

end for

return $(\mathcal{T}, \{\beta_i\})$

After maximal cliques are found, a **clique tree** $\mathcal{T}(\boldsymbol{C}_\mathcal{T}, \boldsymbol{E}_\mathcal{T})$ is constructed with tree nodes $\boldsymbol{C}_\mathcal{T}$ being the maximal cliques. For each clique C_i, the variables in it is denoted as $\boldsymbol{X}_{C_i}$. The corresponding clique value $\psi_i(\boldsymbol{X}_{C_i})$, similar to a factor value $\phi(\boldsymbol{X})$, shows a relative likelihood for each possible state combination of the variables. The initial value $\psi_i(\boldsymbol{X}_{C_i})$ is computed as the product of all factors contained in the clique:

$$\psi_i(\boldsymbol{X}_{C_i}) = \prod_{j:\boldsymbol{X}_{\phi_j} \subseteq \boldsymbol{X}_{C_i}} \phi_j(\boldsymbol{X}_{\phi_j}) \tag{2.4}$$

Note that each factor should be included in only one clique.

The tree edges $\boldsymbol{E}_\mathcal{T}$ are added between cliques so that it is a **maximal spanning tree**, *i.e.*, out of all possible clique trees constructed from the cliques, the sum of edge weights is largest. The edge weight between a pair of cliques is the number of common nodes between the two cliques.

In the last step, the clique tree is **caliberated**. Caliberation is done through **message passing**. Messages are passed from one clique C_i to its adjacent cliques $\{C_j\}, (i,j) \in \boldsymbol{E}_\mathcal{T}$. The message passed from C_i to C_j, denoted as $\delta_{i \to j}$, is computed as follows:

$$\delta_{i \to j}(\boldsymbol{X}_{C_i} \cap \boldsymbol{X}_{C_j}) = \sum_{\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_j}} \left\{ \psi_i(\boldsymbol{X}_{C_i}) \prod_{k \in NB_i - \{j\}} \delta_{k \to i} \right\} \tag{2.5}$$

As can be seen in Equation 2.5, message from C_i to C_j can only be computed when all messages from C_i's neighbors NB_i are passed to C_i, except from the destination clique C_j. Thus, the message passing must be started from cliques who have only one neighbor. In that case, the message passed from C_i to its only neighbor C_j is:

$$\delta_{i \to j} = \sum_{\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_j}} \psi_i(\boldsymbol{X}_{C_i}) \tag{2.6}$$

Message from one clique to another clique is passed only once. After messages are passed between each pair of cliques (in both directions), the caliberated clique value is computed as

$$\beta_i(\boldsymbol{X}_{C_i}) = \psi_i(\boldsymbol{X}_{C_i}) \prod_{k \in NB_i} \delta_{k \to i} \tag{2.7}$$

The caliberated clique value $\beta_i(\boldsymbol{X}_{C_i})$ can be shown (in [29]) to be proportional to the marginal probability of $\boldsymbol{X}_{C_i}$:

$$\beta_i(\boldsymbol{X}_{C_i}) \propto \sum_{\boldsymbol{X} - \boldsymbol{X}_{C_i}} P(\boldsymbol{X}) = P(\boldsymbol{X}_{C_i}). \tag{2.8}$$

Thus, after running junction tree algorithm, individual marginal probabilities for any random variable can be computed, by marginalizing that variable from any clique that contains that variable:

$$P(X_i) = \frac{P(\boldsymbol{X}_{C_j})}{\sum_{\boldsymbol{X}_{C_j} - \{X_i\}} P(\boldsymbol{X}_{C_j})} = \frac{\beta_i(\boldsymbol{X}_{C_j})}{\sum_{\boldsymbol{X}_{C_j} - \{X_i\}} \beta_i(\boldsymbol{X}_{C_j})}, \text{ where } X_i \in \boldsymbol{X}_{C_j}$$

Minimum triangulation, one where the largest clique in the resulting chordal graph is of minimum size, is $\mathcal{NP}$-hard [4]. Exact algorithms of finding optimal triangulation are exponential in the size of the largest clique in the graph. However, heuristic triangulation methods exist that have runtimes linear to the number of nodes in the graph [14, 28]. Finding maximal cliques in a graph is also $\mathcal{NP}$-hard, however, for chordal graphs more efficient algorithms exist [29].

When passing message from one clique C_i to another clique C_j (Equation 2.5), all incoming messages to C_i, *i.e.*, $\{\delta_{k \to i}\}, k \in NB_i - \{j\}$, have to be multiplied to clique value ψ_{C_i}. Then, the multiplied result is summed on variables $\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_j}$, as the message passed to C_j. Assume there are L variables in $\boldsymbol{X}_{C_i}$, and M variables in $\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_j}$ $(0 < M < L)$, and each variable has S states. Then C_i has a total of S^L combination of states, and $\delta_{i \to j}$ has S^{L-M} states. The number of summation is thus $S^L - S^{L-M} = O(S^L)$, which is exponential to the number of nodes L in the clique C_i. Since this kind of summation occurs for every clique in message passing, assuming the largest clique has L_{MAX} nodes, the summation has a complexity of $O(S^{L_{MAX}})$. The **treewidth** Tw of a graph is defined as the $L_{MAX} - 1$, thus, junction tree algorithm has a runtime exponential

to the treewidth.

2.3 Gibbs Sampling

Sampling methods are used in various fields of machine learning and statistics. Given a probability distribution $P(\boldsymbol{X})$, we wish to either

1. Draw samples from this distribution, or
2. Estimate expectations of target function $\phi(\boldsymbol{X})$ with the input of $\phi(\boldsymbol{X})$ obeying the distribution of $P(\boldsymbol{X})$, such that

$$\Phi = \int P(\boldsymbol{X})\phi(\boldsymbol{X})d\boldsymbol{X}. \tag{2.9}$$

Drawing samples from $P(\boldsymbol{X})$ is not always straightforward. For example, we may not know the normalized distribution $P(\boldsymbol{X})$, but unnormalized distribution $\hat{P}(\boldsymbol{X}) = ZP(\boldsymbol{X})$, whose normalizing constant $Z = \sum_{\boldsymbol{X}} \hat{P}(\boldsymbol{X})$ is not easy to compute. Even if we can compute Z, sampling from $P(\boldsymbol{X})$ can still be challenging, since we need to evaluate $P(\boldsymbol{X})$ for every possible $\boldsymbol{X}$ so that the samples come from regions where $P(\boldsymbol{X})$ is big. For high dimensional distribution, the problem is even bigger.

Several methods have been developed to overcome the difficulty. With uniform sampling, the algorithm first draws samples $\boldsymbol{x}_i, i = 1, \ldots, R$, uniformly from the input space $\boldsymbol{X}$, then use the probability of the drawn sample $P(\boldsymbol{x}_i)$ as a weight, so that samples drawn from a less likely region of $P(\boldsymbol{X})$ play less important role in evaluating the expectation of the target function. When the state space is huge for $\boldsymbol{X}$, the chance of picking a sample from a high likelihood region of $P(\boldsymbol{X})$ becomes small, and we will need to draw large number of samples so that some of them come from high likelihood regions.

Importance sampling is a variation of uniform sampling, where instead of uniformly sampling from input space $\boldsymbol{X}$, we use another function, $Q(\boldsymbol{X})$, as an approximation of $P(\boldsymbol{X})$, and draw samples from $Q(\boldsymbol{X})$. We weight them by the ratio $P(\boldsymbol{X})/Q(\boldsymbol{X})$. This method is better than uniform sampling in that now we are able to draw more samples from regions where $P(\boldsymbol{X})$ is huge (given $Q(\boldsymbol{X})$ approximates $P(\boldsymbol{X})$ correctly). However, it is very difficult to find a good

approximation of $P(\boldsymbol{X})$.

Other methods exist, such as rejection sampling, but they have the problem of not being able to sample efficiently from high-dimensional probability space [33].

Gibbs sampling [17] is introduced to address the problem. Given an N-dimensional variable to sample, Gibbs sampling tries to sample 1 dimension at a time, using probability of that dimension conditioned on all other dimensions. It is a type of **Markov Chain Monte Carlo (MCMC)** method [18], where we start from an initial state of $\boldsymbol{X} = \boldsymbol{x}^0$, and generate a series of samples $\boldsymbol{X}^t, t = 0, \ldots, N$, one after another, with each successive sample drawn solely based the previous sample.

Unlike uniform sampling or importance sampling, the series of samples drawn from Gibbs sampling is dependent on each other, since we are drawing new samples based on the probability distribution suggested by previous samples. Gibbs sampling is especially suitable for distributions whose joint probability $P(\boldsymbol{X})$ is hard to compute, but whose local conditional probability is easy to compute. For example, in a Markov random field where a node's conditional probability only depends on the immediate neighboring nodes (*i.e.*, nodes in its Markov Blanket), we can sample this node using the values of its neighbors without considering the rest of the nodes.

Gibbs sampling works as follows. Assume we have a joint probability shown in Equation 2.10, which can be represented graphically as in Figure 2.5.

$$P(X) = P(X_1, X_2, \cdots, X_N), N = 11 \tag{2.10}$$

With Gibbs sampling, we do not attempt to sample all variables at the same time, but rather sample one variable at a time, assuming all other variables are fixed. This idea can be expressed as follows:

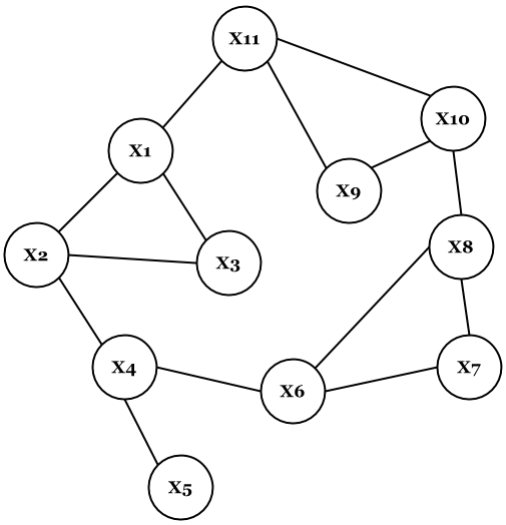


Figure 2.5: A PGM representation of an eleven-dimensional joint probability

$$
\begin{aligned}
x_1^{t+1} &\sim P(X_1|x_2^t, x_3^t, \cdots, x_N^t) = P(X_1|\boldsymbol{x}_{MB_1}^{current}) \\
x_2^{t+1} &\sim P(X_2|x_1^{t+1}, x_3^t, \cdots, x_N^t) = P(X_2|\boldsymbol{x}_{MB_2}^{current}) \\
x_3^{t+1} &\sim P(X_3|x_1^{t+1}, x_2^{t+1}, \cdots, x_N^t) = P(X_3|\boldsymbol{x}_{MB_3}^{current}) \\
&\vdots \\
x_N^{t+1} &\sim P(X_N|x_1^{t+1}, x_2^{t+1}, \cdots, x_{N-1}^{t+1}) = P(X_N|\boldsymbol{x}_{MB_N}^{current})
\end{aligned}
$$

Note that the order of nodes to be sampled is chosen randomly. Pseudocode for the Gibbs sampling is shown in Algorithm 2.2.

Algorithm 2.2 Sequential Gibbs Sampling

```
for all t ∈ 1, 2, ..., N do
    for all X_i do
        Sample X_i^(t+1) ~ P(X_i | x_MB_i^current)
    end for
end for
```

Figure 2.6 shows one round of Gibbs sampling. Many rounds are executed until certain termination condition is met. Section 4.1 will provide more details on termination condition in Gibbs sampling.

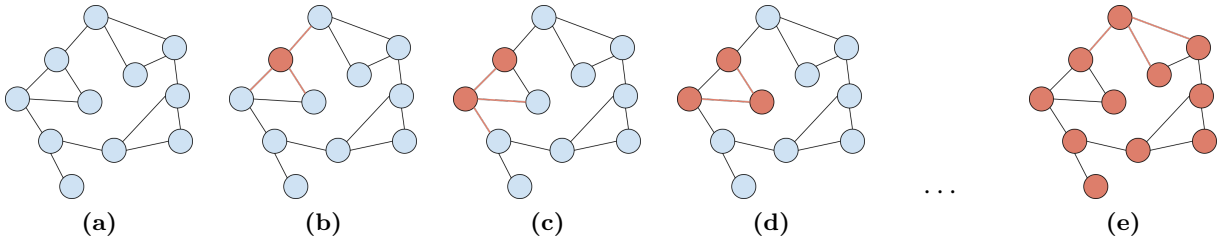


Figure 2.6: Illustration of Gibbs sampling steps in one round. (a) Initial state. (b)-(d) New node values are sampled sequentially, based on their Markov blankets (connected by red lines). (e) End of a sampling round.

The reason we can sample a node purely based on its neighbors is that, in a probabilistic graphical model, a variable is conditionally independent from all other variables, given its Markov blanket, as discussed in Section 2.1.

How do we use Gibbs sampling for inference tasks? Suppose that we are interested in $P(X, Y|\boldsymbol{E} = \boldsymbol{e})$, the joint probability of X and Y given evidence $\boldsymbol{E} = \boldsymbol{e}$. When sampling, we fix random variable $\boldsymbol{E}$ to have value $\boldsymbol{e}$. After having enough samples, we can simply count the occurrences of X and Y in different joint states, and divide by the total number of samples, and use it as an approximation of $P(X, Y|\boldsymbol{E} = \boldsymbol{e})$. Assume we have R independent samples, then

$$P(X = x, Y = y|\boldsymbol{E} = \boldsymbol{e}) = \frac{\sum_i^R I(X_i = x, Y_i = y)}{R}$$

$$I(B) = \begin{cases} 1, & \text{if } B = \text{true} \\ 0, & \text{otherwise} \end{cases}$$

2.4 Optical Flow

Optical flow algorithms measure the movement of objects and pixels between consecutive images in a temporal sequence [5, 20]. Result of optical flow computation can be used in a wide range of tasks, including image segmentation [46, 47], 3D shape acquisition [43], perceptual organization [41], object recognition [48], and more. Many optical flow methods have been developed, such as differential methods Horn-Schunck in [20], Lucas-Kanade in [32], etc, region-based matching [2],

frequency-based methods [1] and phase-based methods [16].

As input to an optical flow algorithm, we are given two images $\boldsymbol{I}$ and $\boldsymbol{I'}$. Each image consists of N pixels: $\boldsymbol{I} = \{I(p_i)\}, i = 1, \ldots, N$, where $I(p_i)$ is the pixel's intensity. A pixel p is represented by its coordinates (e.g., $p = (p_x, p_y)$ for 2-D image). The output of an optical flow algorithm is a list of vectors $\boldsymbol{F} = \{F(p_i)\}$, $(F(p_i) = (F_x(p_i), F_y(p_i))$ in 2-D image), corresponding to the movement of each pixel in the first image. For simplicity, we will use F_i and $F(p_i)$ interchangeably.

For example, consider the images shown in Figure 2.7.

(a) A moving car **(b)** Optical flow for the moving car **(c)** Color palette for optical flow

(d) A man juggling balls **(e)** Optical flow for the juggling

Figure 2.7: Optical flow examples. The input images (a) and (d) are two consecutive images overlaid on each other. Optical flows from the first to the second images are shown in colors ((b) and (e)), with the corresponding flow vector for each color shown in the color palette (c).

In the example, Figure 2.7a shows two images of a car moving on a country road, overlaid on each other. The car moved a bit to the left side. The optical flow between the two images are shown in Figure 2.7b. Here, optical flow is represented by colors, and the corresponding flow matching each color can be found in the color palette in Figure 2.7c. For example, green color is on the left side of the palette, meaning the flow vector is pointing to the left. The section corresponding to

the car is green, meaning the pixels in this section moved leftwards. Other parts are red, meaning a movement to the right side, perhaps due to a shift in camera's angle.

In Figure 2.7d, a man is juggling balls. Most of the image is static, while the balls and hands have movements. The corresponding optical flow is shown in Figure 2.7e. Indeed, there are only very few movements in areas with bright colors, while most areas are black, meaning there are no movement.

The key assumption for existing optical flow methods is the invariance of pixel intensities, *i.e.*, the pixels in the first image, after the movement, should have the same pixel intensity in the second image: $I'(p + F(p)) = I(p)$. An optical flow method tries to find an assignment of $\boldsymbol{F}$ such that the intensity assumption holds for every pixel. There are usually more than one way of doing the assignment, thus various optical flow methods impose different extra assumptions to help find the best optical flow.

Chapter 3

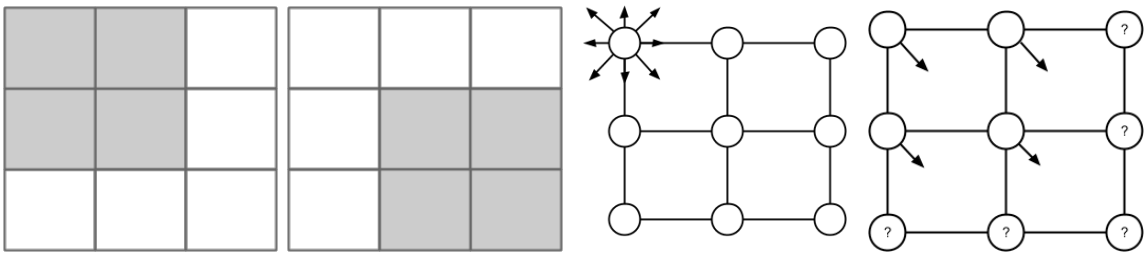
Probabilistic Optical Flow

In this chapter, we demonstrate how we can solve optical flow problems in a probabilistic setting, using Markov random fields. We first show a few intuitions in the general optical flow problem, then discuss how they can be expressed probabilistically. We formalize the algorithm as Probabilistic Optical Flow (POF), and show a few results of this method. Comparisons of our novel POF method with other optical flow methods will be presented in Chapter 6.

3.1 Intuition

Optical flow algorithms usually assume *invariance of pixel intensity*, meaning that the intensity of each pixel in the original image should be the same as the intensity of the corresponding pixel in the moved image. It is an assumption that does not hold in many situations, for example, when illumination condition changes or object surface direction changes. We can relax this assumption by assigning probablity to different flow vector values. For example, a flow vector that does not change pixel intensity has a higher likelihood, while one that changes intensity has a lower likelihood. The change in likelihood can be modeled as a function of change in intensity.

Consider the example in Figure 3.1. Here we are trying to find the optical flow from Image 1 to Image 2. For simplicity, both images are of 3x3 size, with 3 pixels on each edge. There is a black 2x2 square in the top left corner of Image 1, and it moved to the bottom right corner in Image 2. We can see the entire black square moved in bottom right direction for 1 pixel. The true



(a) Image 1 **(b)** Image 2 **(c)** Possible movements for one pixel **(d)** True Flow

Figure 3.1: Illustration of probabilistic formulation of optical flow. The block square in Image 1 moved to bottom right corner in Image 2, with corresponding ground truth flow in (d). For each pixel, it can move to 8 of its neighbors, and stay in its current position.

optical flow is shown in Figure 3.1d. The true optical flow for the white section is unknown, and we marked them with question marks. When calculating optical flow, we are only interested in the flow for the black box.

Now, if we did not know the true flow, how many possible optical flow assignments are there? For simplicity, assume each pixel in Image 1 can only move 1 pixel (to its 8 neighboring pixels), or not move at all (shown in Figure 3.1c). Note that we allow pixels to move out of the image. Then, each pixel's optical flow has 9 possible assignments. Since there are 9 pixels, there are 9^9 possible assignments, for this simple 3x3 image. Finding the optimal optical flow involves finding the one "best" assignment from these 9^9 possible assignments.

We denote all the pixels in Image 1 as $\boldsymbol{p} \equiv \{p_i\}, i = 1, \ldots, 9$, and the corresponding optical flow as $\boldsymbol{F} \equiv \{F_i\}$. How do we tell whether one optical flow assignment is better than another?

There are several intuitions:

- **Intensity**: A pixel's intensity should not change, or change very little, after the movement.
- **Distance**: A pixel's movement should favor short distance moves over long distance moves.
- **Neighbors**: Neighboring pixels should move similar distances and in similar directions.

For the intensity intuition, traditional optical flow methods have made very tight assumptions, namely that pixel intensity does not change after movement. Ideally, in our method, we can still have flows that don't change intensities, but when necessary, allow for intensity changes.

The neighbor intuition is understandable: In an image, blocks of pixels usually represent the

same objects, and when the object moves, all pixels depicting it will move in a similar fashion. It also implies that optical flow at boundary of objects should be smooth, and not have sudden changes.

3.2 Probabilistic Optical Flow

With the above intuitions, we assign each optical flow instantiations a probability, with higher probability given to instantiations that satisfy the intuitions better. We wil call our method **Probabilistic Optical Flow (POF)**.

First let us consider how to express intensity intuition. For a pixel p_i in image $\boldsymbol{I}$ (the first image), the intensity of the pixel is $I(p_i)$. The optical flow vector corresponding to the pixel is $F(p_i)$, which we will denote as F_i for conciseness. In a 2D image, a pixel p is represented by its x and y coordinates: $p \equiv (x, y)$, and F_i is a 2-dimensional vector, $F_i \equiv (F_x, F_y)$. After the movement, the new position of the pixel in the second image $\boldsymbol{I'}$ is now $p_i + F_i$, and thus the new intensity for that pixel is $I'(p_i + F_i)$. The change in intensity can be represented as $\delta I = I'(p_i + F_i) - I(p_i)$.

We wish to give higher probability to F_i that results in smaller δI, and smaller probability for higher δI. Since we use a distribution to express our "belief" in how likely a flow vector takes different values, it is subjective and any distribution that gives higher probability to smaller δI can be used. We will use Gaussian distribution to model this probability, shown in Equation 3.1.

Gaussian distribution is used here since it is part of exponential family and allows easy chaining of multiple probabilities within the family. The variance parameter α is used to control the sensitivity of the probability to intensity change. When α is big, changes in intensity has little impact in the probability, while when α is small, a small change in intensity can vary the probability a lot.

$$\begin{aligned} P_I(F_i; \boldsymbol{I}, \boldsymbol{I'}) &\propto \exp\left(-\frac{\delta I^2}{\alpha^2}\right) \\ &\propto \exp\left(-\frac{(I'(p_i + F_i) - I(p_i))^2}{\alpha^2}\right) \end{aligned} \tag{3.1}$$

Figure 3.2 shows the probability variation for different δI.

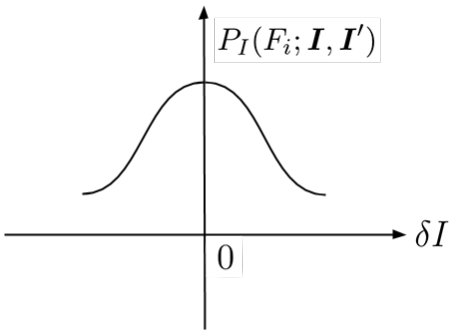


Figure 3.2: Probability distribution of flow value for different intensity changes for a pixel. The bigger intensity change, the less likely it will happen.

Then, we will continue with modeling based on distance intuition. We assign higher likelihood to shorter pixel movements, still using Gaussian distribution. The distance of pixel p_i's movement is $|F_i|$, thus:

$$P_D(F_i; \boldsymbol{I}, \boldsymbol{I}') \propto \exp\left(-\frac{|F_i|^2}{\beta^2}\right) \tag{3.2}$$

The variance parameter β is used to control sensitivity to distance changes. Bigger β gives more tolerance in distance changes.

The third intuition is about neighboring pixels: They should move in a similar fashion. We represent two neighboring pixels as p_i and p_j, and denote the entire set of neighboring pixels as $\boldsymbol{E} \equiv \{(i,j)\}$ for all p_i and p_j that are adjacent in $\boldsymbol{I}$. Higher probability is assigned to flow values that are similar. Again, the variance parameter γ is used to control sensitivity to difference in neighboring pixels' intensities:

$$P_N(F_i, F_j; \boldsymbol{I}, \boldsymbol{I}') \propto \exp\left(-\frac{|F_i - F_j|^2}{\gamma^2}\right), \quad \forall (i,j) \in \boldsymbol{E} \tag{3.3}$$

The joint distribution for all optical flow vectors $\boldsymbol{F} = \{F_i\}$ is a product of all the individual probability distributions, as seen in Equation 3.4:

$$
\begin{aligned}
P(\boldsymbol{F};\boldsymbol{I},\boldsymbol{I}') &= \frac{1}{Z}\prod_i P_I(F_i;\boldsymbol{I},\boldsymbol{I}')\cdot\prod_i P_D(F_i;\boldsymbol{I},\boldsymbol{I}')\cdot\prod_{(i,j)\in\boldsymbol{E}} P_N(F_i,F_j;\boldsymbol{I},\boldsymbol{I}') \\
&= \frac{1}{Z}\prod_i \exp\left(-\frac{(I'(p_i+F_i)-I(p_i))^2}{\alpha^2}\right)\cdot\prod_i \exp\left(-\frac{|F_i|^2}{\beta^2}\right)\cdot\prod_{(i,j)\in\boldsymbol{E}} \exp\left(-\frac{|F_i-F_j|^2}{\gamma^2}\right) \\
&= \frac{1}{Z}\exp\left(-\left(\sum_i \frac{(I'(p_i+F_i)-I(p_i))^2}{\alpha^2}+\sum_i\frac{|F_i|^2}{\beta^2}+\sum_{(i,j)\in\boldsymbol{E}}\frac{|F_i-F_j|^2}{\gamma^2}\right)\right) \qquad (3.4)
\end{aligned}
$$

Here, Z is the normalization term,

$$
Z=\sum_{\boldsymbol{F}}\exp\left(-\left(\sum_i \frac{(I'(p_i+F_i)-I(p_i))^2}{\alpha^2}+\sum_i\frac{|F_i|^2}{\beta^2}+\sum_{(i,j)\in\boldsymbol{E}}\frac{|F_i-F_j|^2}{\gamma^2}\right)\right)
$$

The exponential form of probability distribution shown in Equation 3.4 can be represented more compactly using **energy function**, as the following:

$$
\begin{aligned}
P(\boldsymbol{F};\boldsymbol{I},\boldsymbol{I}') &= \frac{1}{Z}\exp\left(-\left(\sum_i \frac{(I'(p_i+F_i)-I(p_i))^2}{\alpha^2}+\sum_i\frac{|F_i|^2}{\beta^2}+\sum_{(i,j)\in\boldsymbol{E}}\frac{|F_i-F_j|^2}{\gamma^2}\right)\right) \\
&= \frac{1}{Z}\exp\left(-E(\boldsymbol{F})\right)
\end{aligned}
$$

Where

$$
E(\boldsymbol{F})=\sum_i \frac{(I'(p_i+F_i)-I(p_i))^2}{\alpha^2}+\sum_i\frac{|F_i|^2}{\beta^2}+\sum_{(i,j)\in\boldsymbol{E}}\frac{|F_i-F_j|^2}{\gamma^2}. \qquad (3.5)
$$

We can represent individual terms in the energy function as separate energy components:

$$
\begin{aligned}
E(\boldsymbol{F}) &= \sum_i \frac{(I'(p_i+F_i)-I(p_i))^2}{\alpha^2}+\sum_i\frac{|F_i|^2}{\beta^2}+\sum_{(i,j)\in\boldsymbol{E}}\frac{|F_i-F_j|^2}{\gamma^2} \\
&= \sum_i E_I(F_i)+\sum_i E_D(F_i)+\sum_{(i,j)\in\boldsymbol{E}} E_N(F_i,F_j), \qquad (3.6)
\end{aligned}
$$

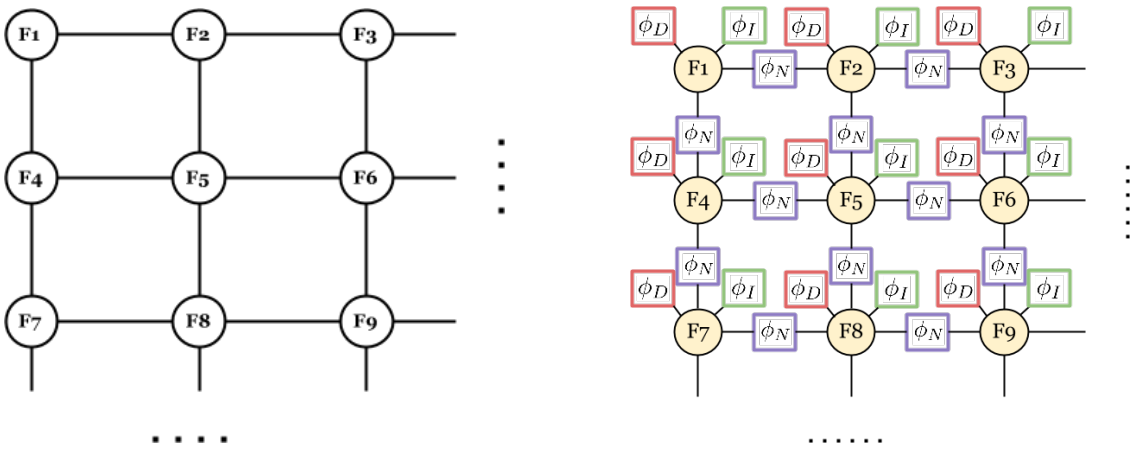
where

$$E_I(F_i) = (I'(p_i + F_i) - I(p_i))^2/\alpha^2 \tag{3.7}$$

$$E_D(F_i) = |F_i|^2/\beta^2 \tag{3.8}$$

$$E_N(F_i, F_j) = |F_i - F_j|^2/\gamma^2. \tag{3.9}$$

We will call these energy components **intensity energy** (E_I), **distance energy** (E_D) and **neighbor energy** (E_N) respectively. Intensity energy and distance energy are defined on individual pixels, and neighbor energy is defined on pairs of pixels that are adjacent to each other, in Image $\boldsymbol{I}$. The factors corresponding to these energies are **intensity factor** ϕ_I, **distance factor** ϕ_D and **neighbor factor** ϕ_N.



(a) Markov random field model for probabilistic optical flow. Nodes (F_i) represents flow vectors, and adjacent pixels' flow vectors are connected by edges.

(b) Factor graph of POF MRF. Intensity factors ϕ_I and distance factors ϕ_D are associated with individual nodes, and neighbor factors ϕ_N are associated with adjacent nodes.

Figure 3.3: Markov random field model and corresponding factor graph for POF.

Now we will use a probabilistic graphical model (PGM) representation of the distribution in Equation 3.4. As mentioned in Section 2.1, there are two types of probabilistic graphical models: Bayesian network (with directed edges) and Markov random fields (with undirected edges). Since in our probabilistic modeling of optical flow there is no clear causal dependency from one flow vector to another flow vector, we will use Markov random fields (MRF) as the representation. Figure 3.3a shows the MRF. Here, each pixel p_i's optical flow F_i is a random variable (the nodes), and

neighboring pixels' flows are connected by edges. A fully annotated factor graph of the MRF is shown in Figure 3.3b.

3.3 Computing POF Using Gibbs Sampling

Now we modeled optical flow with a probability distribution $P(\boldsymbol{F}; \boldsymbol{I}, \boldsymbol{I}')$. What we are interested in is to find out an optical flow assignment $\boldsymbol{f}^*$ that yields maximum probability, *i.e.*, $\boldsymbol{f}^* = \boldsymbol{f}_{MPE}$. When we consider the energy function representation $P(\boldsymbol{F}; \boldsymbol{I}, \boldsymbol{I}') = \frac{1}{Z} \exp(-E(\boldsymbol{F}))$, we want to find $\boldsymbol{f}^*$ that minimizes the energy $E(\boldsymbol{F})$ (Equation 3.6).

Consider, as input, we are given two images, each with N pixels. For each pixel, assume it can move at most d pixels (both horizontally and vertically), in integer values. It means the flow vector is discrete, and the total number of possible movements is $(2d+1)^2$, as seen in Figure 3.4. Here the black pixel in the center can move to all the grey pixels, as well as stay in its original position, if the maximum moving distance is d. Thus, the total possible assignments for all pixels in an N pixel image is $[(2d+1)^2]^N = (2d+1)^{2N}$.

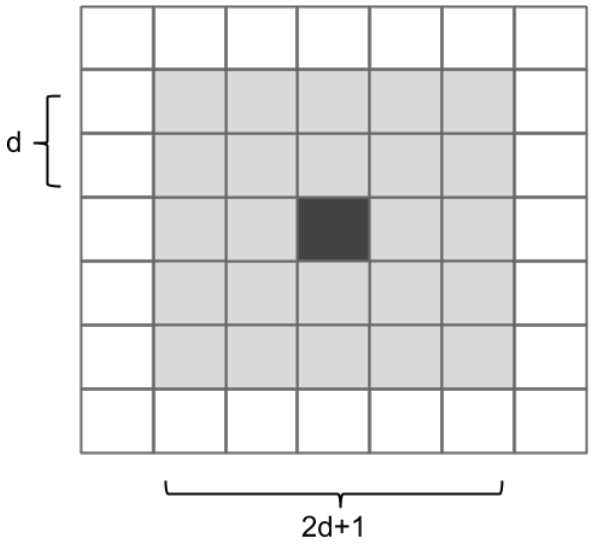


Figure 3.4: Possible movements of a pixel. It can stay in its current position, or move to neighbors within d pixel's range.

With this exponentially many possible assignments, it is impractical to calculate $P(\boldsymbol{F}; \boldsymbol{I}, \boldsymbol{I}')$ for all possible $\boldsymbol{f}$ and pick the optimal one. Even for a quite small case, $d = 1$, $N = 100$, there are

2.66×10^{95} values to compute, which is clearly infeasible for current technology.

We can turn to randomized methods, such as Gibbs sampling. With Gibbs sampling, we can perform a random walk in a probability space $P(\boldsymbol{F})$, with the goal of finding an optical flow instantiation $\hat{\boldsymbol{f}}$ such that $\hat{\boldsymbol{f}}$ approaches $\boldsymbol{f}^* = \text{argmax}_{\boldsymbol{f}} P(\boldsymbol{f})$. Note that we are omitting the terms $\boldsymbol{I}, \boldsymbol{I}'$ from now on to simplify notations.

To be able to use Gibbs sampling, we need to be able to sample F_i given its Markov blanket $\boldsymbol{F}_{MB_i}$. For Markov random field, a random variable's Markov blanket is its immediate neighbors, $\boldsymbol{F}_{MB_i} = \{F_j\}, (i,j) \in \boldsymbol{E}$. Thus, the probability of F_i is

$$\begin{aligned} P(F_i|\boldsymbol{f}_{MB_i}) &= \frac{1}{Z_i} P_I(F_i) \cdot P_D(F_i) \cdot \prod_{(i,j)\in\boldsymbol{E}} P_N(F_i, f_j) \\ &= \frac{1}{Z_i} \exp\left(-\frac{(I'(p_i + F_i) - I(p_i))^2}{\alpha^2} - \frac{|F_i|^2}{\beta^2} - \sum_{(i,j)\in\boldsymbol{E}} \frac{|F_i - f_j|^2}{\gamma^2} \right) \end{aligned} \tag{3.10}$$

We can apply Equation 3.10 directly to the Gibbs sampling algorithm: First we initialize flow vectors $\boldsymbol{F}$ randomly, then in each round t, sample new flow vectors $\boldsymbol{f}^t$ based on the current values of their 4 neighbors, using Equation 3.10. We keep sampling until there is no improment in $P(\boldsymbol{f})$ for certain amount of rounds, then return the sample $\hat{\boldsymbol{f}}$ with highest probability as an estimate of $\boldsymbol{f}^*$. The pseudocode is presented in Algorithm 3.1.

Algorithm 3.1 POF: Computing Probabilistic Optical Flow with Gibbs Sampling

Input: Image $\boldsymbol{I}$, Image $\boldsymbol{I}'$, α, β, γ, observe rounds R
Output: Optical flow $\hat{\boldsymbol{f}}$

```
Initialize f^0: ∀i, f_i^0 = (0,0); f̂ = f^0; t̂ = 0; Round t = 1;
repeat
    for all F_i ∈ F do
        Sample f_i^t ~ P(F_i|f_{MB_i}) {Refer to Equation 3.10}
    end for
    if E(f^t) < E(f̂) then
        f̂ = f^t; t̂ = t
    end if
    t = t + 1
until t − t̂ > R
return f̂
```

The sampling process of one round can be seen in Figure 3.5.

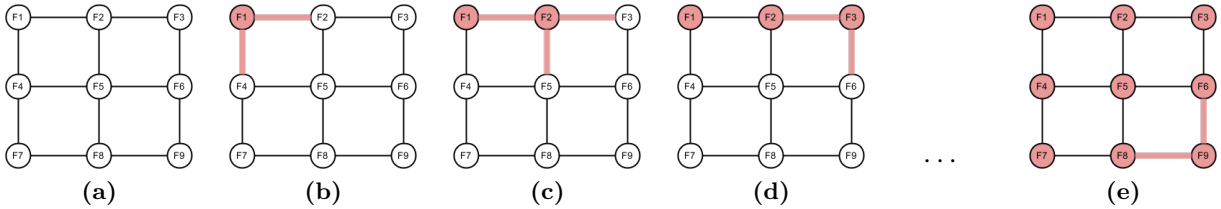
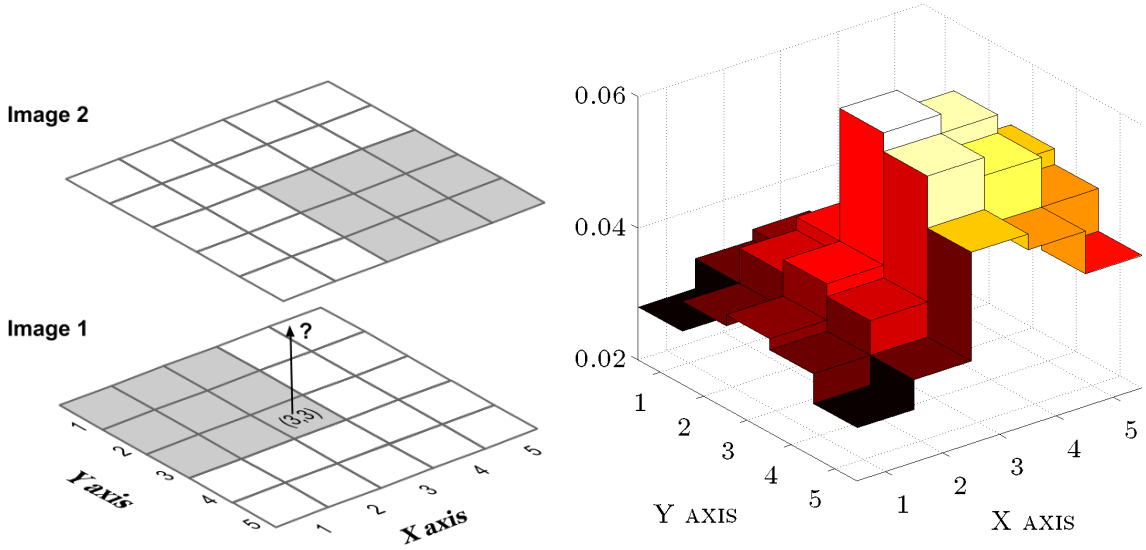


Figure 3.5: Illustration of Gibbs sampling steps in one round for optical flow MRF. (a) Initial state. (b)-(d) New flow values are sampled sequentially; The node currently being sampled is red and has red neighboring edges. (e) End of a sampling round; Node F9 in the bottom right corner is the last one to be sampled.

An example of sampling one flow vector is shown in Figure 3.6. In Figure 3.6a, two input images are shown. The dark 3x3 rectangle moved from top left corner in Image 1 to bottom right corner in Image 2. Assume we are just starting with the sampling, and all flow vectors are initialized as $F_{i,j} = (0, 0)$. Now, if we sample the next value for $F_{3,3}$ (the pixel in the center), what is $P(F_{3,3})$?



(a) Sampling flow vector $F_{3,3}$ **(b)** Probability of each possible movement destination for $F_{3,3}$

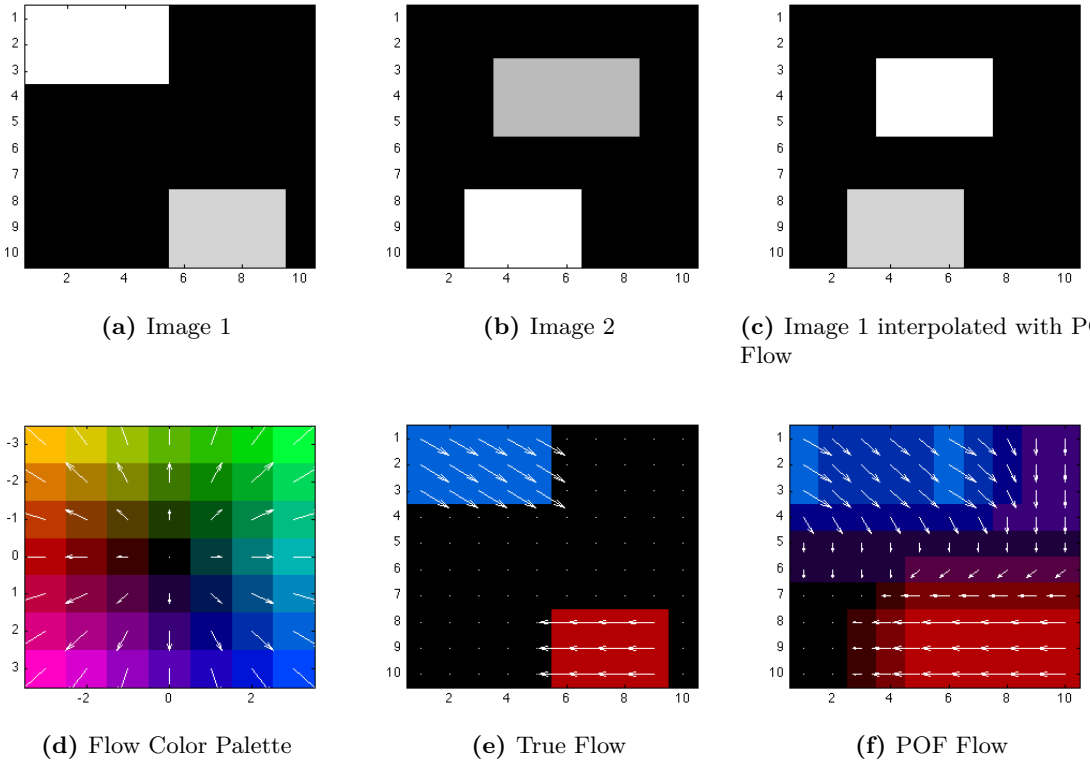
Figure 3.6: Illustration of sampling one flow vector. The dark rectangle in Image 1 moved from top left corner to bottom right corner in Image 2. The probability of pixel (3,3) to move to each pixel is shown in (b).

In Figure 3.6b, the probability of each possible movement desitination for $F_{3,3}$ is shown, computed using $\alpha = 5, \beta = 10, \gamma = 10$. Because of intensity energy's constraint (Equation 3.7), probability for the pixel to move to $X \geq 3, Y \geq 3$ is bigger than moving to other pixels. Also,

probability is highest in the center $(3,3)$, and drops for surrounding pixels. The first reason is due to distance energy (Equation 3.8), moving to a pixel closer to $(3,3)$ has higher probability than moving to a further pixel. Then, since all flow vectors are currently $F_{i,j} = (0,0)$, the flow vector $F_{3,3}$ has higher probability for staying as $F_{3,3} = (0,0)$, since that minimizes the neighbor energy.

3.4 Examples of Running POF

In this section we show a few examples of running our probabilistic optical flow method. Extensive experiments and results will be presented in Chapter 6.



(a) Image 1 **(b)** Image 2 **(c)** Image 1 interpolated with POF Flow

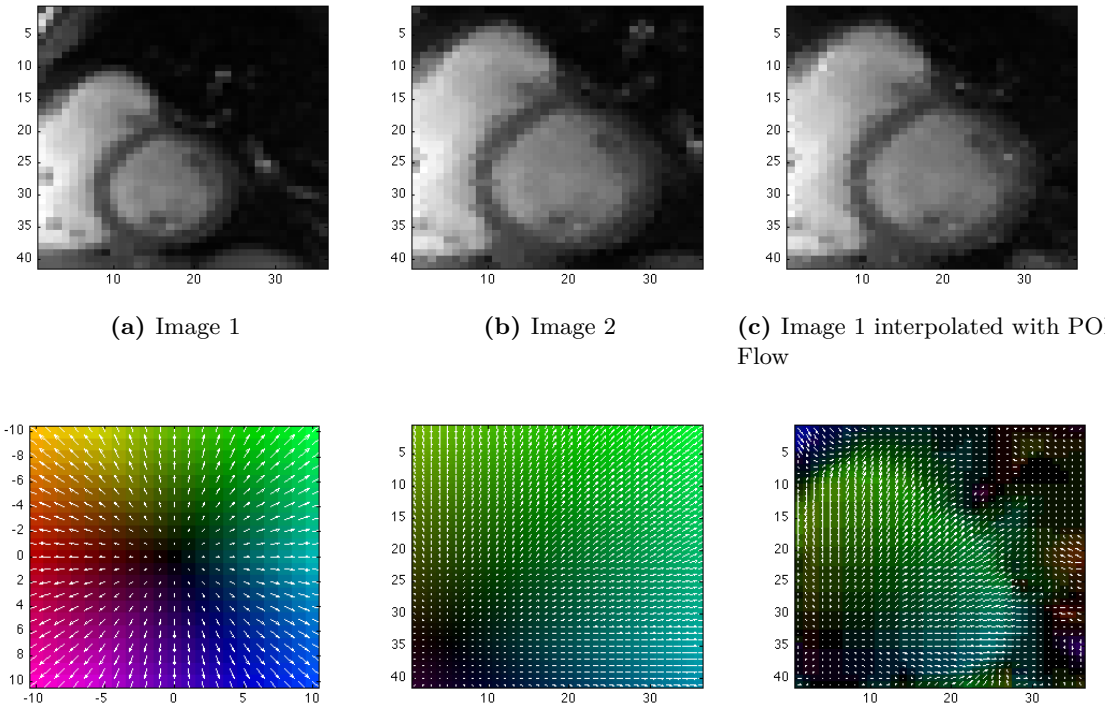
(d) Flow Color Palette **(e)** True Flow **(f)** POF Flow

Figure 3.7: Running POF on synthetic images. The bright rectangle in top left corner moved in bottom right direction, with slight dimming. The bottom rectangle moved leftwards and became brighter. The POF flow (f) captured the true flow (e), with smoothing flow between objects.

In Figure 3.7, we apply the POF method on a pair of synthetic images. The first image, in Figure 3.7a, consists of two bright rectangles with different intensity levels. In the second image (see Figure 3.7b), both rectangles moved and changed their intensity levels. The ground truth optical flow is shown, with black section representing areas where the true flow is unknown. From the POF

result in Figure 3.7f, we can see it correctly found out true flow for most pixels. Additionally, the flow connects areas between two rectangles with smooth transition.

In Figure 3.8, we use images of the left ventricle of a patient, from York University's Cardiac MRI dataset[1] [3]. The dataset contains cardiac MR images acquired from 33 patients, each patient's sequence consisting of 20 frames and 8-15 slices along the long axis, for a total of 7980 images. The input Image 1 corresponds to patient 3, slice 5, at time frame 1. The dataset does not have ground truth flows between images, thus we manually created a ground truth flow (Figure 3.8e), and applied it on Image 1, to create Image 2. The result of POF running on Image 1 and Image 2 is shown in Figure 3.8f. We applied the POF's optical flow result on Image 1, and generated interpolated image Figure 3.8c. As can be seen, the interpolated image is almost identical to input Image 2. More detailed experiment descriptions using cardiac images will be presented in experiment Section 6.1.6.



(a) Image 1 **(b)** Image 2 **(c)** Image 1 interpolated with POF Flow

(d) Flow Color Palette **(e)** Ground truth flow **(f)** POF flow

Figure 3.8: Running POF on cardiac images. Image 1 is taken from York dataset, patient 3, slice 5 at time frame 1. Image 2 is generated by applying manually created ground truth flow (e) to Image 1.

[1] http://www.cse.yorku.ca/~mridataset/

3.5 Conclusion of the Chapter

We presented a probabilistic model of optical flow, and showed how to use Gibbs sampling to compute an approximate optimal flow assignment. This model is first presented in [39]. We also showed a couple of experiments where we applied POF to input images. Ideas of forumulating optical flow using a probability can also be found in [15].

In our POF method, we use three different energies to constrain flow values, namely intensity, distance, and neighbor energies. The POF method is robust to changing pixel intensity, results in consistent flow values among neighboring pixels and within tracked objects, and yields smooth transitions along object boundaries. Formulating optical flow in a probabilistic setting also enables adding more constraints on flow values, should the need arise.

However, we have not mentioned the speed of running POF method. Actually, compared to several other numerical and deterministic optical flow algorithms, such as [12, 13, 20], the POF method is very slow. Fortunately there are many different ways to speed up Gibbs sampling, which is the primary component of POF method. We will discuss general ways to speed up Gibbs sampling in Chapter 4, and how to parallelize Gibbs sampling in Chapter 5. We will apply the techniques discussed in those chapters to our POF method, and present the experimental results in Chapter 6.

Chapter 4

Speeding Up Gibbs Sampling

In this chapter, we look into ways to make Gibbs sampling run faster, in other words, converge faster. We will first explain what "convergence" is, then discuss three different ways to have faster convergence: Clever initialization, blocking, and simulated annealing. We will discuss how these methods can be applied to our POF algorithm, when applicable.

There is another very important aspect of having faster convergence: Parallelization. We will separate it into Chapter 5.

4.1 Convergence in Gibbs Sampling

When we are doing Gibbs sampling, we are trying to get samples from an unknown probability space, and use them as an approximation of the probability space. For example, consider a 2-dimensional probability space $P(X, Y)$, depicted in Figure 4.1.

There are several peaks in the probability space, representing highly likely regions for (X, Y). Ideally, in our samples, we have samples drawn evenly from the probability space, so that we have a good understanding of the probability distribution.

Consider another simpler example. In Figure 4.2, we show the probability space for a simple 2-dimensional normal distribution $P(X, Y) \sim \mathcal{N}(\mu, \Sigma)$.

In Gibbs sampling, we typically don't know the entire joint probability, but only the conditional

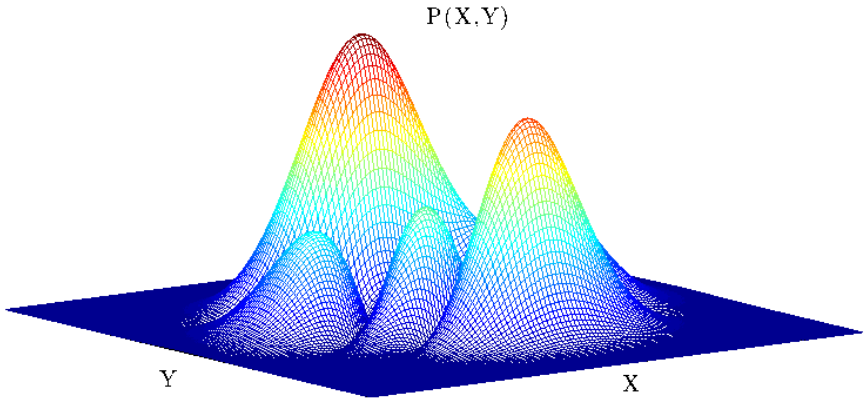
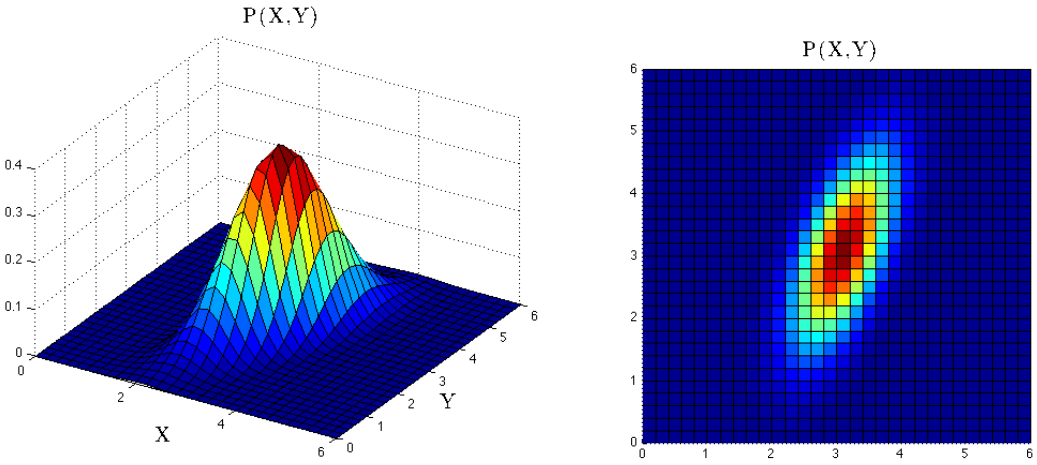


Figure 4.1: Probability space for random distribution $P(X, Y)$



(a) Probability space **(b)** Heat map

Figure 4.2: Probability space for 2-dimensional normal distribution $P(X, Y)$

probabilities. In this case, assume $P(X, Y)$ is unknown, but only $P(X|Y)$ and $P(Y|X)$ are known. Drawn as a Markov random field, we have the following 2-node MRF in Figure 4.3.

In this simple MRF, one round of Gibbs sampling involve sampling two nodes X and Y sequentially. Assume we start from an initial state of $(x_0, y_0) = (0, 0)$. When $Y = 0$, $P(X|Y = 0)$ is a 1-dimensional normal distribution, centered somewhere around $X = 2$ (See heat map in Figure 4.2b). Thus when we sample X, it will be more likely to have a value closer to 2. Let's say the sampled value is 2.3. Then, we use this new value to sample Y. Still based on the heat map, $P(Y|X = 2.3)$ is a 1-dimensional normal distribution, centered somewhere around $Y = 2$. Let's say Y is sampled to be 1.8.

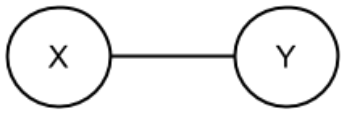


Figure 4.3: MRF for 2-dimensional normal distribution $P(X, Y)$

Now we finished the first round of sampling, and have the sample $(x_1, y_1) = (2.3, 1.8)$. We continue using this new value for the next round of sampling. Since the value is closer to the mean of the $P(X, Y)$, the next sample will have a even higher chance of being close to the center. We illustrated the possible sampling process in Figure 4.4.

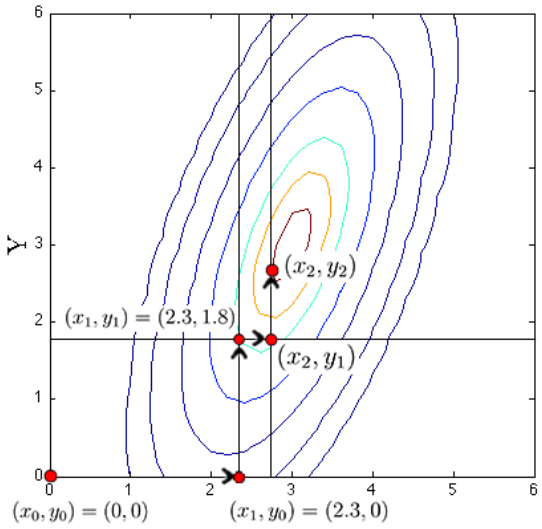


Figure 4.4: Gibbs sampling process for two dimensional normal distribution. Starting from initial state (x_0, y_0), each dimension is sampled sequentially, conditioned on the current value of the other dimension. After a few iterations, the sample (x_2, y_2) is drawn from the more likely probability space.

As can be seen, the samples are gradually drawn from region with higher likelihood. To be able to monitor the sampling process, we can plot the log likelihoods of the samples, that is, $\log P(X, Y)$. It is shown in Figure 4.5.

Initially, since we randomly picked a sample $(x_0, y_0) = (0, 0)$, the log likelihood is very low. As we continue sampling, each new sample is sampled from a region with higher likelihood, thus the log likelihoods keep improving. In the end, when there is no more improvement to make, the log likelihood plateaus at certain level, with fluctuations.

The plateau we observed in the log likelihood plot is what we call "convergence" in Gibbs

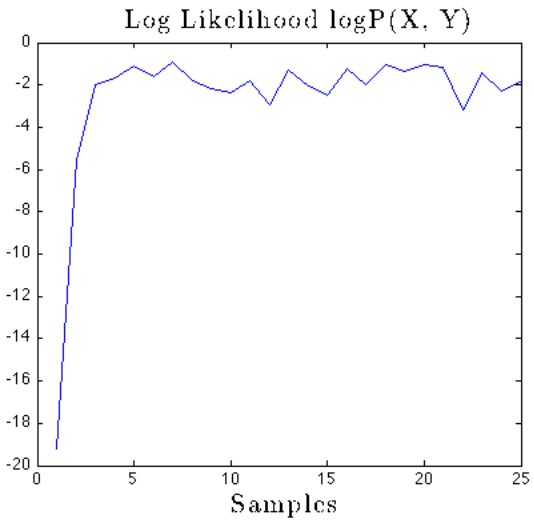


Figure 4.5: Log likelihood of Gibbs sampling process for two dimensional normal distribution

sampling. When sampling converged, we know that the random walk has gone through large portion of the probability space, and the samples can be said to be independent of the initial sample we used as input. Note that the notion of "convergence" is heuristic. There is no concrete definition of convergence, and thus there is no scientifically proven way to detect when a sampling chain has converged.

Another thing to note is that in a probability distribution with multiple "peaks", *i.e.*, highly likely regions (as in Figure 4.1), the samples may be drawn from just one of the many peaks. In this case the plateau we observed may not represent a real convergence, since the sample chain has not traversed a large enough region of the probability space, and samples cannot be seen as a truthful representation of the distribution. Convergence in this kind of distribution is harder to determine than a distribution with a single peak, since we cannot tell whether the plateau comes from one of many peaks, or a single peak. One way to counter such a problem is to extend the sampling for certain period after a plateau is reached, and see if the samples display big change in likelihoods. If there is, then it might be because the sample chain is transitioning from one peak to another peak, in which case we should keep the sampling run longer. Another way to detect a fake convergence, is by running multiple sample chains in parallel, and see if the log likelihood curves converge to the same value.

4.2 Initialization

In Figure 4.4, we started with a random sample $(x_0, y_0) = (0, 0)$. This sample is randomly chosen, and is quite far from the center of the distribution. This led to the slow start in log likelihood convergence in Figure 4.5.

If we started somewhere near the mean of the normal distribution, the log likelihood of the samples would have started right from a high value, leading to a faster convergence.

Thus, when doing Gibbs sampling, one way of having a faster convergence, is doing better initialization. We want to choose initial samples close to regions with high probability mass, so that later samples can be sampled from that region directly.

In the Gibbs sampling process of POF algorithm (Section 3.3), currently we are initializing the optical flows with all 0's (Algorithm 3.1 line 1). In order to have a faster convergence, we would like to initialize with flow values that have higher likelihoods. One way to find such an initialization is to use results from some other optical flow algorithms. The reason is, an optical flow assignment another optical flow algorithm believes to be a good fit for the input images should have higher likelihood in our POF model than a random flow assignment. Of course, the time spent on the other optical flow algorithm to find an initial sample as input to POF should be relatively short, and the flow result has to "fit" into POF algorithm, *i.e.*, has a high likelihood in the POF model. The question is, which optical flow algorithm's result has this property, and how fast can it generate such a result? We will experiment with using results from different optical flow methods in Section 6.3, and see how they improve the POF runtime.

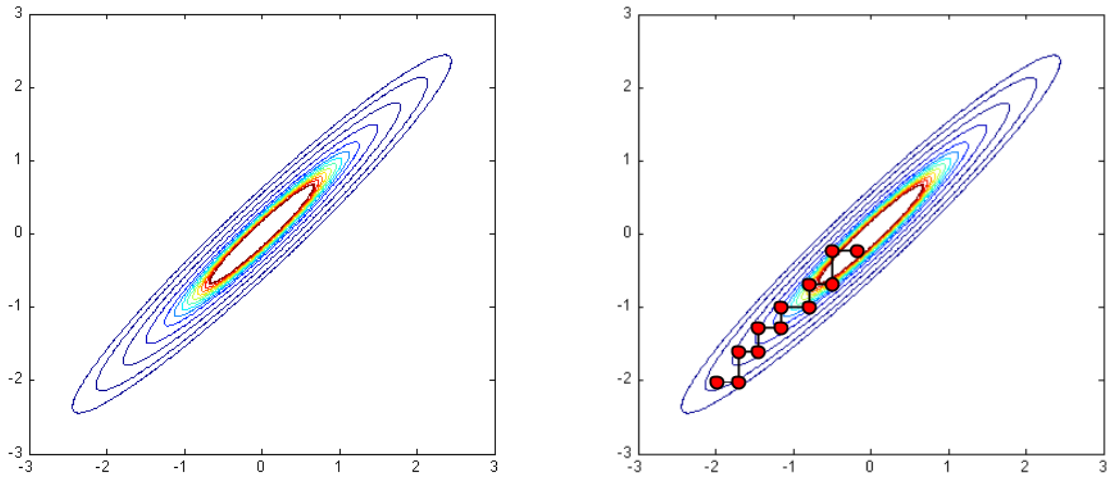
4.3 Block Gibbs Sampling

Another way to speed up Gibbs sampling, is to sample nodes jointly, i.e, doing **block Gibbs sampling**.

4.3.1 Motivation

We will use another normal distribution to illustrate the motivation.

Consider a two dimensional normal distribution $P(X, Y) \sim \mathcal{N}(\mu, \Sigma)$ shown in Figure 4.6a. As can be seen, the two dimensions are highly correlated.



(a) Highly correlated two dimensional normal distribution

(b) Gibbs sampling process

Figure 4.6: Gibbs sampling in highly correlated two dimensional normal distribution

If we start sampling from $(x_0, y_0) = (-2, -2)$, it will take many sampling rounds for the sample chain to reach the center, which is depicted in Figure 4.6b.

The convergence for this sampling process is very slow. Since the variables are highly correlated, when we fix one variable and sample another, it will have very small changes in its sampled value, thus the sampling is "stuck" in a local region.

If we sample the highly correlated variables jointly, in this example, sample $(x, y) \sim \mathcal{N}(\mu, \Sigma)$, the correlation problem can be solved.

In PGM with large number of random variables, sampling all variables jointly will be impractical, and we have to sample small groups of random variables in the hope that they can bring the sampling chain out of those "local" regions quickly. This is exactly the definition of **block Gibbs sampling**.

Another way to understand why sampling blocks of random variables is better, is to consider **state coverage**. Assume we have a 10 node PGM, each node with 2 possible states, then the entire graph will have $2^{10} = 1024$ states. If we sample each node sequentially, each time we can

transition from one state to two possible states. However, if we jointly sample 5 nodes, each time we can transition to $2^5 = 32$ possible states, thus greatly speeding up the sampling.

The general framwork of doing block Gibbs sampling is shown in Algorithm 4.1. In Figure 4.7, we illustrate block Gibbs sampling process. Compare it with Figure 2.6 of regular Gibbs sampling on page 20.

Algorithm 4.1 Block Gibbs Sampling General Framework

for all $t \in 1, 2, \ldots, N$ **do**
  Decide on blocks $\boldsymbol{B} = \{B_i\}, B_i = \boldsymbol{X}_{B_i}, \cup_i \boldsymbol{X}_{B_i} \subseteq \boldsymbol{X}$
  for all B_i **do**
    Jointly sample $\boldsymbol{X}_{B_i}^{(t+1)} \sim P(\boldsymbol{X}_{B_i} | \boldsymbol{x}_{MB_{B_i}}^{current})$
  end for
  for all $X_i \in \boldsymbol{X} - \cup_j \boldsymbol{X}_{B_j}$ **do**
    Sample $X_i^{(t+1)} \sim P(X_i | \boldsymbol{x}_{MB_i}^{current})$
  end for
end for

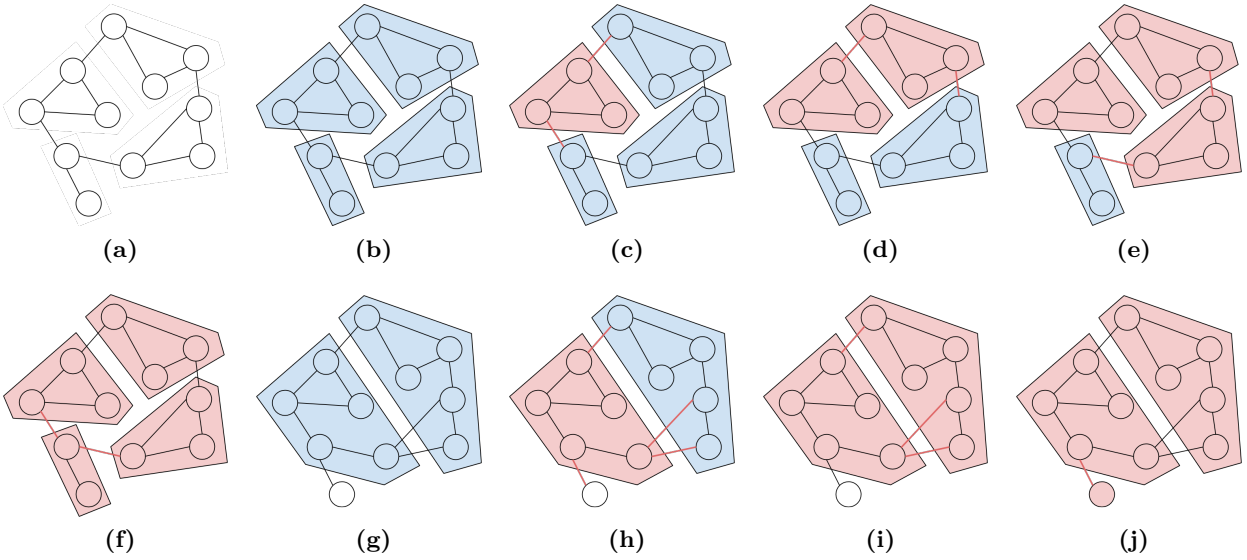


Figure 4.7: Illustration of block Gibbs sampling in two rounds. (a) Initial graph. (b) Round 1 blocking. (c)-(f) Round 1 sampling of blocks, conditioned on current values of the blocks' Markov blanket (red edges). (g) Round 2 blocking. (h)-(i) Round 2 sampling of blocks. (j) Sampling individual nodes not in blocks.

Given a PGM (Figure 4.7a), at the beginning of each round, the blocking is determined (Figure 4.7b, 4.7g). They don't need to include the entire set of random variables (such as in Figure 4.7g). Then, each block is sampled, conditioned on nodes in its Markov blanket. If there are any variables not included in blocks, they are sampled individually just as in regular Gibbs sampling.

What is a Markov blanket for a block? In Section 2.1 Page 11, we defined the Markov blanket of a single random variable X: It is the minimum set of random variables $\boldsymbol{X}_{MB_X}$ given which X is conditionally independent of all other random variables $\boldsymbol{X} - \{X\} - \boldsymbol{X}_{MB_X}$. The Markov blanket for a group of variables $\boldsymbol{X}_B$ has a similar definition: It is the minimum set of random variables $\boldsymbol{X}_{MB_{\boldsymbol{X}_B}}$ given which $\boldsymbol{X}_B$ is conditionally independent of all other random variables $\boldsymbol{X} - \boldsymbol{X}_B - \boldsymbol{X}_{MB_{\boldsymbol{X}_B}}$, as in Equation 4.1.

$$P(\boldsymbol{X}_B|\boldsymbol{X}_{MB_{\boldsymbol{X}_B}}) = P(\boldsymbol{X}_B|\boldsymbol{X}_{MB_{\boldsymbol{X}_B}}, \boldsymbol{X} - \boldsymbol{X}_B - \boldsymbol{X}_{MB_{\boldsymbol{X}_B}}) \tag{4.1}$$

A graphical representation of a block's Markov blanket is shown in Figure 4.8. The black nodes are nodes in the block, and the grey ones are nodes in the Markov blanket.

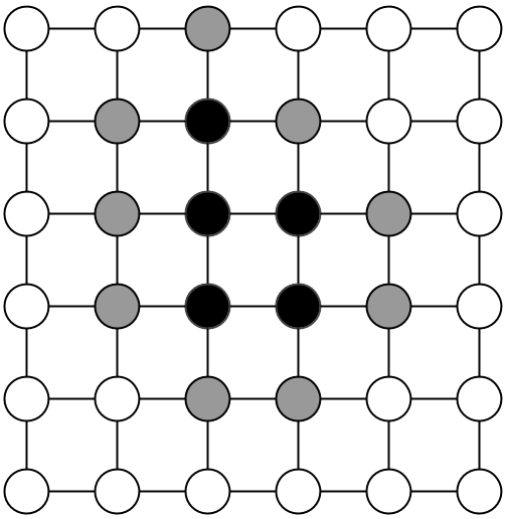


Figure 4.8: Markov blanket of a block in MRF graph

4.3.2 Sampling a Block Jointly

Now, how do we sample nodes in a block jointly? The most straightforward way is to compute the joint distribution for all possible states and sample from the joint distribution. However, the number of possible states of a block is exponential to number of nodes in the block, making it computationally infeasible.

We can sample with the help of junction tree algorithm. Details of junction tree algorithm is

shown in Algorithm 2.1. In junction tree algorithm, the original input PGM $G(\boldsymbol{X}, \boldsymbol{E}, \boldsymbol{\phi})$ is first converted to a clique tree $\mathcal{T}(\boldsymbol{C}_\mathcal{T}, \boldsymbol{E}_\mathcal{T})$, where $\boldsymbol{C}_\mathcal{T} = \{C_i\}$ is the node set consisting of maximal cliques, and $\boldsymbol{E}_\mathcal{T}$ is the edge set. Each clique C_i consists of a subset of random variables $\boldsymbol{X}_{C_i}$ in the input PGM, *i.e.*, $\boldsymbol{X}_{C_i} \subseteq \boldsymbol{X}$. The value of the clique $\psi_i(\boldsymbol{X}_{C_i})$ is computed by multiplying all factors contained in the clique, *i.e.*, $\psi_i(\boldsymbol{X}_{C_i}) = \prod_{j:\boldsymbol{X}_{\phi_j} \subseteq \boldsymbol{X}_{C_i}} \phi_j(\boldsymbol{X}_{\phi_j})$. The clique values are "caliberated" through message passing. The caliberated clique values, denoted as $\beta_i(\boldsymbol{X}_{C_i})$, is proportional to the marginal distribution of nodes in the clique, *i.e.*,

$$\beta_i(\boldsymbol{X}_{C_i}) \propto \sum_{\boldsymbol{X} - \boldsymbol{X}_{C_i}} P(\boldsymbol{X}). \tag{4.2}$$

The unnormalized joint distribution of the PGM, denoted as $\tilde{P}(\boldsymbol{X})$, can be shown as Equation 4.3 (see proof in [29]):

$$\tilde{P}(\boldsymbol{X}) = \frac{\prod_i \beta_i(\boldsymbol{X}_{C_i})}{\prod_{(i,j) \in \boldsymbol{E}_\mathcal{T}} \mu_{i,j}(\boldsymbol{X}_{\mu_{i,j}})} \tag{4.3}$$

Here $\mu_{i,j}$ is the separator of C_i and C_j. It includes variables that are common to C_i and C_j, *i.e.*, $\boldsymbol{X}_{\mu_{i,j}} = \boldsymbol{X}_{C_i} \cap \boldsymbol{X}_{C_j}$, and the value of the separator is the marginalization of the common variables:

$$\mu_{i,j}(\boldsymbol{X}_{\mu_{i,j}}) = \sum_{\boldsymbol{X}_{C_i} - \boldsymbol{X}_{\mu_{i,j}}} \beta_i(\boldsymbol{X}_{C_i}) = \sum_{\boldsymbol{X}_{C_j} - \boldsymbol{X}_{\mu_{i,j}}} \beta_i(\boldsymbol{X}_{C_j}) \tag{4.4}$$

To jointly sample all variables in the junction tree, we can do a **backward sampling** of cliques in the junction tree. Since the cliques form a tree, we pick a random clique as the root clique C_r. Then, for each remaining clique C_i, we denote C_{PA_i} as its parent clique. Each clique, except for the root clique, will have exactly one parent clique. We can re-write the joint distribution in Equation 4.3 using the new notation:

$$
\begin{aligned}
\tilde{P}(\boldsymbol{X}) &= \frac{\prod_i \beta_i(\boldsymbol{X}_{C_i})}{\prod_{(i,j)\in \boldsymbol{E}_{\mathcal{T}}} \mu_{i,j}(\boldsymbol{X}_{\mu_{i,j}})} \\
&= \beta_r(\boldsymbol{X}_{C_r}) \frac{\prod_i \beta_i(\boldsymbol{X}_{C_i})}{\prod_i \mu_{i,PA_i}(\boldsymbol{X}_{\mu_{i,PA_i}})} \\
&= \beta_r(\boldsymbol{X}_{C_r}) \prod_i \frac{\beta_i(\boldsymbol{X}_{C_i})}{\mu_{i,PA_i}(\boldsymbol{X}_{\mu_{i,PA_i}})} \\
&= \beta_r(\boldsymbol{X}_{C_r}) \prod_i \frac{\beta_i(\boldsymbol{X}_{C_i})}{\sum_{\boldsymbol{X}_{C_i}-\boldsymbol{X}_{C_{PA_i}}} \beta_i(\boldsymbol{X}_{C_i})} \qquad (4.5)
\end{aligned}
$$

(4.6)

As shown in Equation 2.8, $\beta_i(\boldsymbol{X}_{C_i})$ is the unnormalized marginal probability of $\boldsymbol{X}_{C_i}$, *i.e.*, $\beta_i(\boldsymbol{X}_{C_i}) = Z_i P(\boldsymbol{X}_{C_i})$, where $Z_i = \sum_{\boldsymbol{X}_{C_i}} \beta_i(\boldsymbol{X}_{C_i})$. Plugging it to Equation 4.5, we get

$$
\begin{aligned}
\tilde{P}(\boldsymbol{X}) &= \beta_r(\boldsymbol{X}_{C_r}) \prod_i \frac{\beta_i(\boldsymbol{X}_{C_i})}{\sum_{\boldsymbol{X}_{C_i}-\boldsymbol{X}_{C_{PA_i}}} \beta_i(\boldsymbol{X}_{C_i})} \\
&= \beta_r(\boldsymbol{X}_{C_r}) \prod_i \frac{Z_i P(\boldsymbol{X}_{C_i})}{\sum_{\boldsymbol{X}_{C_i}-\boldsymbol{X}_{C_{PA_i}}} Z_i P(\boldsymbol{X}_{C_i})} \\
&= \beta_r(\boldsymbol{X}_{C_r}) \prod_i \frac{P(\boldsymbol{X}_{C_i})}{\sum_{\boldsymbol{X}_{C_i}-\boldsymbol{X}_{C_{PA_i}}} P(\boldsymbol{X}_{C_i})} \\
&= \beta_r(\boldsymbol{X}_{C_r}) \prod_i P(\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_{PA_i}} | \boldsymbol{X}_{C_i} \cap \boldsymbol{X}_{C_{PA_i}})
\end{aligned}
$$

(4.7)

If we replace β_r by its normalized version, we get the joint normalized distribution:

$$
P(\boldsymbol{X}) = P(\boldsymbol{X}_{C_r}) \prod_i P(\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_{PA_i}} | \boldsymbol{X}_{C_i} \cap \boldsymbol{X}_{C_{PA_i}})
$$

(4.8)

This means that we can first sample nodes in the root clique C_r, then, sample nodes that are

adjacent to the root clique, conditioned on the sampled values of nodes in the root clique. We keep sampling downstream cliques conditioned on sampled values from upstrea cliques, until all nodes are sampled. The whole sampled values will obey the distribution of $P(\boldsymbol{X})$.

The above junction tree sampling algorithm is summarized in Algorithm 4.2.

Algorithm 4.2 JTA-SAMPLE: Junction Tree Sampling Algorithm

Input: PGM $G(\boldsymbol{X}, \boldsymbol{E}, \boldsymbol{\phi})$
Output: Sample $\boldsymbol{x}$
  $(\mathcal{T}(\boldsymbol{C}_\mathcal{T}, \boldsymbol{E}_\mathcal{T}), \{\beta_i(\boldsymbol{X}_{C_i})\}) \leftarrow$ JTA$(G(\boldsymbol{X}, \boldsymbol{E}, \boldsymbol{\phi}))$ {Algorithm 2.1}
  Pick a random clique from $\boldsymbol{C}_\mathcal{T}$ and set as root clique C_r
  Initialize unsampled clique set $\boldsymbol{C}_U \leftarrow \boldsymbol{C}_\mathcal{T}$
  Sample $\boldsymbol{x}_{C_r} \sim \beta_r(\boldsymbol{X}_{C_r})$
  $\boldsymbol{C}_U \leftarrow \boldsymbol{C}_U - \{C_r\}$
  while $\exists C_i \in \boldsymbol{C}_U$ **AND** $C_{PA_i} \notin \boldsymbol{C}_U$ **do**
    Sample $(\boldsymbol{x}_{C_i} - \boldsymbol{x}_{C_{PA_i}}) \sim \beta_i(\boldsymbol{X}_{C_i} - \boldsymbol{X}_{C_{PA_i}} | \boldsymbol{x}_{C_{PA_i}})$
    $\boldsymbol{C}_U \leftarrow \boldsymbol{C}_U - \{C_i\}$
  end while
  return $\boldsymbol{x}$

4.3.3 Previous Work

There are a few existing works exploring the blocking of Gibbs sampling. Jensen et al. first introduced the concept of blocking to Gibbs sampling in [21]. In their method, they propose to first put all nodes in a single block, and gradually trim down nodes so that the joint sampling of nodes in the block is computationally feasible. They try to make the block as large as possible to fully take advantage of speed of joint sampling.

Later, Joseph [19] proposed a way to construct multiple "splashes" of nodes: Starting from a certain number of empty blocks, symultaneiously add new nodes to the splashes, using heuristics to select the best nodes to be added. This method is designed for parallel computing, where each of the block is given to a separate computation unit and sampled simultaneously.

Deepak et al. [45] presented a way to do dynamic blocking and collapsing for Gibbs sampling. Their algorithm periodically updates partitioning into blocked and collapsed variables by leveraging correlation statistics gathered from generated samples and enables rapid mixing. They try to find the best way to both block variables and collapse them, by using scoring functions, and cast it as

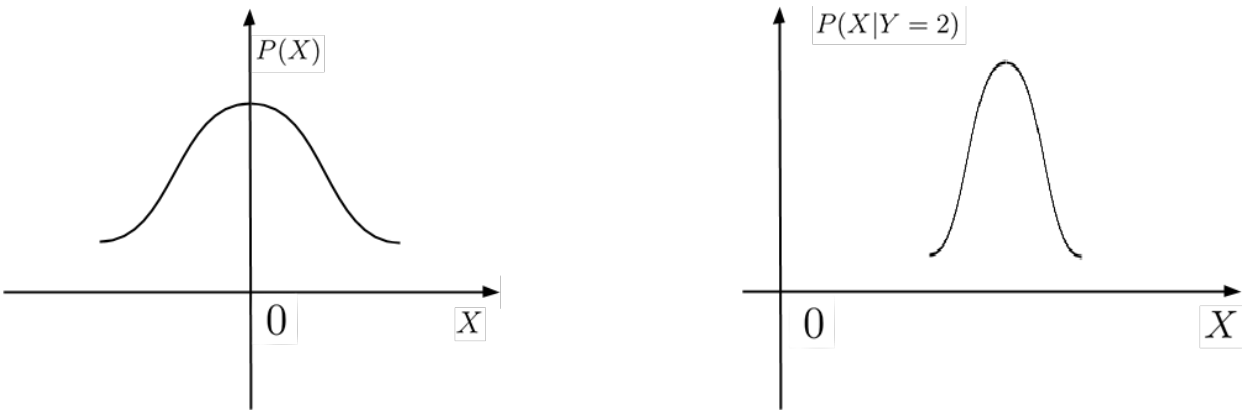
a multi-objective optimization problem. They use Hellinger distance to compute distance between $P(X_i, X_j)$ and $P(X_i)P(X_j)$ and use it as a meatric of correlation between X_i an X_j.

With several blocking methods proposed, what is the best way of blocking variables? Also, when we later would like to apply parallelization to sampling individual blocks, what is the appropriate size of each block, and what is the proper number of blocks? Since we are using junction tree algorithm to sample the nodes in the blocks, what is the proper treewidth for each block?

4.3.4 Generating Blocks

Now we come to the first question: What is the best way of blocking variables? We have discussed that we want to jointly sample variables that are tightly coupled, or in other words, highly correlated. When the variables are highly correlated, when sampling one variable, the possible results will come from a small section from its entire possible choices.

Still using the two variable distribution example in Figure 4.6a. When we don't know Y, the choice of X range in a broad region. When Y is fixed, the possible choice of X is limited to a small region. This is depicted in Figure 4.9.



(a) $P(X)$ **(b)** $P(X|Y = 2)$

Figure 4.9: Distribution of one dimension of two dimensional normal distribution. The marginal distribution $P(X)$ has a more even distribution across all possible values than the conditional distribution $P(X|Y = 2)$.

To be able to tell X and Y are highly correlated assuming we did not know the true underlying joint distribution of these two variables, we can compare the ratio of the conditional probability

of X given Y to the marginal probability of X, and see how much they differ. There are two situations: 1) The conditional probability is much bigger than the marginal. 2) The conditional is much smaller than the marginal. Both situations signal a high correlation between the two variables. This is depicted in Figure 4.10.

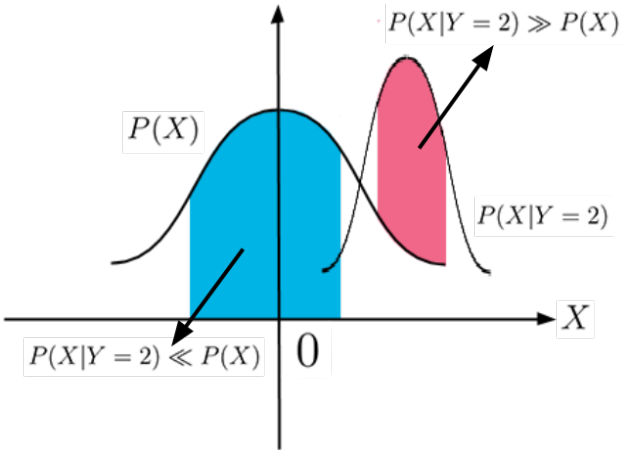


Figure 4.10: Symptoms of high correlation between two variables. When the conditional distribution of one variable differs a lot from its marginal distribution, it may be due to high correlation.

To be able to quantitively measure correlation, we propose a correlation score $C_{X|Y}$ for correlation X to Y, as defined in Equation 4.9.

$$C_{X|Y} = \left| \log \frac{P(X|Y)}{P(X)} \right| \tag{4.9}$$

We are taking an absolute logarithm of the ratio between conditional and marginal probability, which allows fair comparison of both when conditional is much larger or much smaller than marginal probability.

We can extend Equation 4.9 to a block of nodes $\boldsymbol{X}_B$, as in Equation 4.10:

$$C_{X|\boldsymbol{X}_B - \{X\}} = \left| \log \frac{P(X|\boldsymbol{X}_B - \{X\})}{P(X)} \right| \tag{4.10}$$

Note that the correlation measure can only be computed when the variables being conditioned on are fixed on certain values, such as $Y = 2$ in Equation 4.9.

Except for the correlation score we proposed, there are other ways to find out if one variable

is correlated with a set of variables. We can compute mutual information between two variables to determine how much information one can tell about one variable, based on knowledge about another variable, and use it as approximation of correlation. We can also use symptoms of high correlation, such as slow transition in a variable's state, to detect blocks. We will implement some of the methods and compare how they perform in block Gibbs sampling.

4.3.5 Time to Generate Blocks

Another thing to think about when implementing block Gibbs sampling, is the time to generate the blocks. If we choose to generate the blocks before sampling starts (call it **static blocking**), we will have to completely rely on information given by the factors of the PGM. For example, to compute $C_{X|\boldsymbol{X}_B-\{X\}}$, we will need to compute $P(X|\boldsymbol{X}_B - \{X\})$ and $P(X)$. When the block size increases, the computation of $P(X|\boldsymbol{X}_B - \{X\})$ grows exponentially. Computing $P(X)$ is even harder, since that involves integrating out all other random variables.

We can also try to compute the blocks after we have some initial samples, after which we have some insights to the distribution. We call it **dynamic blocking**. For example, say one variable X has 10 possible values, but in the first 10 samples X has only 1 sampled value, then probably we can put X into a larger block and sample them jointly to make X transition to other states faster. Also, after having some initial samples, we can use them as an approximation of the underlying distribution, and compute the correlation measures using these samples directly. For example, to compute $P(X = x|Y = y)$, we can simply count the co-occurrences of $(X = x, Y = y)$, and divide by the number of times $Y = y$. Computing the measures in such a heuristic way may not yield accurate results, but they are very fast to compute, and are able to provide insight into correlation status.

In our experiment, we will focus on finding the blocks dynamically, at the beginning of each sampling round.

4.3.6 Number and Size of Blocks

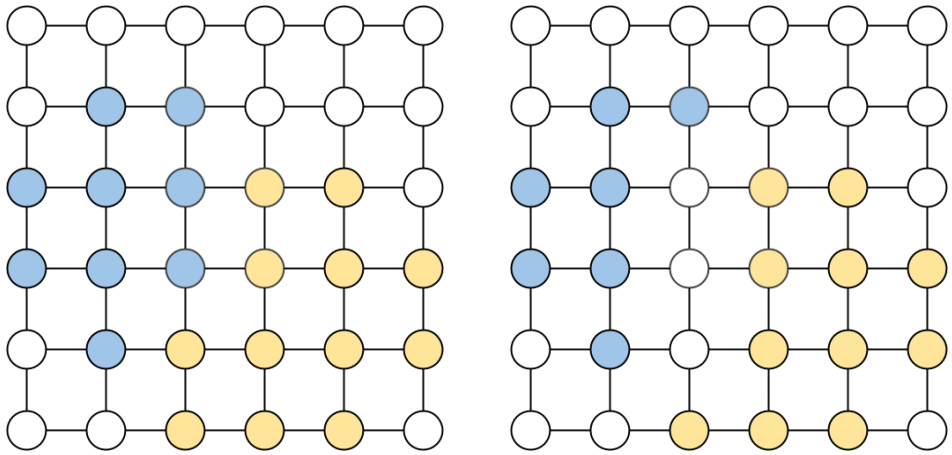
When we are generating the blocks, we need to control the number of blocks, as well as the size of them. The number of blocks will affect how we can parallelize the computation, and the size of a block affects the time and resource required for jointly sampling the block. There are several possible strategies:

- Generate very few blocks, with huge sizes. The most extreme example of this is Jensen's method [22], where only one block is generated that is made as large as possible. The advantage of this method is that the huge block can take advantage of exact inference algorithm (such as junction tree algorithm) to the fullest extent, but the drawback is there is very less opportunity is exposed for parallelization. The sampling of the entire round has to be done sequentially, since there is only one block.
- Generating many small blocks. When the blocks are small, sampling each block is very fast. However, splitting the blocks, sending blocks to individual computation units, and sampling the blocks may incur a significant overhead, when the number of blocks is huge. If the number of blocks is not determined beforehand, deciding on the number can be an algorithmic burden.
- Generate fixed number of blocks, each with controlled size. In this case, we are manually controlling the size and number of blocks. The design of algorithm can be simplified since it doesn't have to be clever about choosing these parameters, but that burden is on the person who is running the algorithm. Choosing the right size and number of blocks can be a tough decision. In Joseph's work [19], the number of blocks is matched to the number of computation unit, and the size of block is restricted such that the treewidth of the block is below a certain threshold.

Deciding on the number and size of blocks is an engineering choice. We will experiment with different configurations, and present the result in Section 6.5.

4.3.7 Adjacency of Blocks

When we are generating multiple blocks and sample them in parallel, there is an important note: The blocks that needs to be sampled at the same time should not be adjacent to each other. That is, if $\boldsymbol{X}_{B_1}$ and $\boldsymbol{X}_{B_2}$ are going to be sampled simultaneously, there should be no nodes $X_1 \in \boldsymbol{X}_{B_1}$ and $X_2 \in \boldsymbol{X}_{B_2}$ such that X_1 and X_2 are neighbors. In other words, there should be a "safety zone" at least one node wide that separates blocks. We show an example in Figure 4.11.



(a) Incorrect blocking **(b)** Correct blocking

Figure 4.11: Block separation in block Gibbs sampling. Blocks that will be sampled simultaneously should not have Markov blanket nodes in each other. That means the nodes in them should not be adjacent.

Why can't adjancent blocks be sampled at the same time? Because sampling a block requires the current value of its Markov blanket, and sampling the block that is adjacent to this block will change the value of its Markov blanket, leading to a conflict situation. The exact theory behind it has to do with property of Markov Chain Monte Carlo (MCMC), which will be explained Section 5.3.1.

4.3.8 Algorithm

In Algorithm 4.3 we summarize the process we will use to experiment with different blocking methods.

Algorithm 4.3 Block Gibbs Sampling Algorithm Customized

Input: PGM $G(\boldsymbol{X}, \boldsymbol{E}, \boldsymbol{\phi})$, Number of blocks B, max treewidth Tw, minimum block size S_{min}, maximum block size S_{max}, minimum samples before blocking N_S, block growth method $\mathcal{M}$

Output: Samples $\boldsymbol{S}$

1: Initialize sample set $\boldsymbol{S} \leftarrow \varnothing$
2: **for all** $t \in 1, 2, \ldots, N$ **do**
3: **if** $t < N_S$ **then**
4: Sample $\boldsymbol{X}$ sequentially (Algorithm 2.2)
5: **else**
6: Generate B blocks $\boldsymbol{B} = \{B_i\} = \{\boldsymbol{X}_{B_i}\}_{i=1}^{B}$, each with one randomly chosen variable $X \in \boldsymbol{X}$
7: For each block B_i, create neighbor set $\boldsymbol{X}_{N_{B_i}}$, that contains all neighbors of nodes in B_i
8: **while** $\exists |\boldsymbol{X}_{N_{B_i}}| > 0$ **do**
9: **for all** B_i with non empty neighbor set $\boldsymbol{X}_{N_{B_i}}$ **do**
10: **for all** $X_j \in \boldsymbol{X}_{N_{B_i}}$ **do**
11: $S_j \leftarrow$ ComputeScore
12: **end for**
13: Add $\hat{X}$ to $\boldsymbol{X}_{B_i}$ with minimum score $\hat{S} = \min_j S_j$
14: Update $\boldsymbol{X}_{N_{B_i}}$: Remove $\hat{X}$ and add its neighbors. If any node X belongs to multiple neighbor sets, remove X from all of them.
15: **end for**
16: **end while**
17: **for all** $B_i \in \boldsymbol{B}$ **do**
18: Remove B_i from $\boldsymbol{B}$ if $|\boldsymbol{X}_{B_i}| < S_{min}$ or $|\boldsymbol{X}_{B_i}| > S_{max}$ or treewidth of $\boldsymbol{X}_{B_i} > Tw$
19: **end for**
20: **for all** $B_i \in \boldsymbol{B}$ **do**
21: Sample $\boldsymbol{X}_{B_i}$ using JTA
22: **end for**
23: Sample remaining nodes $\boldsymbol{X} - \cup_i \boldsymbol{X}_{B_i}$ sequentially
24: **end if**
25: Add current sample to sample set $\boldsymbol{S} \leftarrow \boldsymbol{S} + \{\boldsymbol{X}^t\}$
26: **end for**

As input, we specify a number of parameters, such as number of blocks, max treewidth, etc, to control block generation. We also specify minimum number of samples before blocking, so that various heuristics are computed only when there are enough samples. As of block growth method $\mathcal{M}$, we implemented a few of the measures we discussed above, and the details of the measures and how they are calculated are presented in Algorithm 4.4.

Then, we start the sampling process. We use regular Gibbs sampling, until we have enough samples N_S to start blocking (Line 4).

When generating blocks, we first initialize blocks with single randomly chosen variables (Line

6). Then, we loop through the blocks, and grow the blocks. Each time a single node is added to the block, chosen from the block's neighbor set. The nodes in the block's neighbor set is sorted by the score (Line 11), and the node with minimum score is added to the block. Algorithm 4.4 shows how to compute the score.

After block is grown, its neighbor set is updated. At any time, if a node is added to more than one neighbor set, it is removed from all those neighbor sets to guarantee that no blocks are adjacent to each other.

After we grow the blocks until no more node can be added to any of the blocks, we filter out the blocks that do not meet requirements: Too small, too large, or has a big treewidth (Line 18). We will show how controlling the size and treewidth of blocks can affect the runtime, in Section 6.5.

Then, we start sampling the blocks, using junction tree algorithm (Line 21). If there are nodes that do not belong to any blocks, they are sampled using regular Gibbs sampling (Line 23).

Algorithm 4.4 ComputeScore: Compute score for a node given a block

Input: Node X, block $\boldsymbol{X}$, block growth method $\mathcal{M}$, sample set $\boldsymbol{S} = \{\boldsymbol{x}^t\}, t = 1, \cdots, R$, current sample round t^*
Output: Score S

$S \leftarrow \infty$
if $\mathcal{M} =$ `max_correlation` **then**
  $S = -C_{X|\boldsymbol{X}_B}$
else if $\mathcal{M} =$ `min_conditional` **then**
  $S = \sum_{t=1}^{R} I(\boldsymbol{x}_B^t = \boldsymbol{x}_B^{t^*}, x^t = x^{t^*})$
else if $\mathcal{M} =$ `min_marginal` **then**
  $S = \sum_{t=1}^{R} I(x^t = x^{t^*})$
else if $\mathcal{M} =$ `min_coverage` **then**
  $S = |\text{unique}(\{x^t\}_{t=T-|X|}^{T})|/|X|$
else
  print Unknown method $\mathcal{M}$
end if
return S

In Algorithm 4.4 ComputeScore, we show how to compute score for various metrics. We implemented the correlation measure in Equation 4.10, and call it `min_correlation`. there is also `min_conditional` metric which is essentially the heuristics measure used in Joseph's splash sampler [19], which will be explained below. We also implemented two additional measures: `min_marginal`

and `min_coverage`.

- `max_correlation`: In Equation 4.10, we defined the correlation between a random variable X and a block of variables $\boldsymbol{X}_B$ to be $C_{X|\boldsymbol{X}_B-\{X\}} = \left|\log \frac{P(X|\boldsymbol{X}_B-\{X\})}{P(X)}\right|$. Assuming $\boldsymbol{X}_B$ and $\{X\}$ are disjoint, which is the case in ComputeScore, we can simplify the equation to be $C_{X|\boldsymbol{X}_B} = \left|\log \frac{P(X|\boldsymbol{X}_B)}{P(X)}\right|$. We will be using samples we have collected up to the current sampling round, $\boldsymbol{S} = \{\boldsymbol{x}^t\}, t = 1, \ldots, R$, to approximate both $P(X|\boldsymbol{X}_B)$ and $P(X)$. We will be calculating the correlation score for the *current* status of X and $\boldsymbol{X}_B$, since it is the correlation in the current status that affects the convergence and state transition. So, what we are interested in is really $C_{X=x^{t^*}|\boldsymbol{X}_B=\boldsymbol{x}_B^{t^*}} = \left|\log \frac{P(X=x^{t^*}|\boldsymbol{X}_B=\boldsymbol{x}_B^{t^*})}{P(X=x^{t^*})}\right|$

$$P(X = x^{t^*}|\boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}) = \frac{P(X = x^{t^*}, \boldsymbol{X}_B = \boldsymbol{x}_B^{t^*})}{P(\boldsymbol{X}_B = \boldsymbol{x}_B^{t^*})} \tag{4.11}$$

where

$$P(X = x^{t^*}, \boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}) = \sum_{t=1}^{R} I(\boldsymbol{x}_B^t = \boldsymbol{x}_B^{t^*}, x^t = x^{t^*})/R \tag{4.12}$$

$$P(\boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}) = \sum_{t=1}^{R} I(\boldsymbol{x}_B^t = \boldsymbol{x}_B^{t^*})/R \tag{4.13}$$

Note $I(x)$ is the indicator function, and takes value 1 when x is true, and 0 when x is false. Similarly,

$$P(X = x^{t^*}) = \sum_{t=1}^{R} I(x^t = x^{t^*})/R \tag{4.14}$$

Putting Equation 4.12 and 4.13 back into Equation 4.11, and putting both Equation 4.11 and 4.14 into the definition of $C_{X=x^{t^*}|\boldsymbol{X}_B=\boldsymbol{x}_B^{t^*}}$, we get

$$C_{X=x^{t^*}|\boldsymbol{X}_B=\boldsymbol{x}_B^{t^*}} = \left|\log \frac{R \cdot \sum_{t=1}^{R} I(\boldsymbol{x}_B^t = \boldsymbol{x}_B^{t^*}, x^t = x^{t^*})}{\sum_{t=1}^{R} I(x^t = x^{t^*}) \cdot \sum_{t=1}^{R} I(\boldsymbol{x}_B^t = \boldsymbol{x}_B^{t^*})}\right| \tag{4.15}$$

This means we can simply count the occurrences of samples when X is equal to current value, $\boldsymbol{X}_B$ is equal to current values, and $X \cup \boldsymbol{X}_B$ jointly takes their current values, and use them to compute the approximation of their correlation.

- `min_conditional`: For this metric, we try to find the conditional probability of $P(X =$

$x^{t^*}|\boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}$). In splash sampler [19], when they are growing the blocks, the metric they used to find the node to add to the block is a heuristic measure, defined as follows:

$$S(X) = \left|\log \frac{\sum_x P(\boldsymbol{X}_B, X = x)}{P(\boldsymbol{X}_B, X = x^{t^*})}\right| \tag{4.16}$$

We can simplify the above measure:

$$\begin{aligned} S(X) &= \left|\log \frac{\sum_x P(\boldsymbol{X}_B, X = x)}{P(\boldsymbol{X}_B, X = x^{t^*})}\right| \\ &= \left|\log \frac{P(\boldsymbol{X}_B)}{P(\boldsymbol{X}_B, X = x^{t^*})}\right| \\ &= \left|\log \frac{1}{P(X = x^{t^*}|\boldsymbol{X}_B)}\right| \\ &= \left|-\log P(X = x^{t^*}|\boldsymbol{X}_B)\right| \\ &= -\log P(X = x^{t^*}|\boldsymbol{X}_B) \end{aligned}$$

The last step is possible since $\log P(\cdot)$ is always negative.

In splash sampler block growth, they pick nodes with biggest $S(X)$, that means the smallest $\log P(X = x^{t^*}|\boldsymbol{X}_B)$. Since logarithm is monotonicly increasing function, it is really finding node with smallest conditional probability $P(X = x^{t^*}|\boldsymbol{X}_B)$. Thus, we renamed it to be `min_conditional`.

To find the conditional probability, we can use the following heuristics:

$$\begin{aligned} P(X = x^{t^*}|\boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}) &= \frac{P(X = x^{t^*}, \boldsymbol{X}_B = \boldsymbol{x}_B^{t^*})}{P(\boldsymbol{X}_B = \boldsymbol{x}_B^{t^*})} \\ &\propto P(X = x^{t^*}, \boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}) \end{aligned}$$

Since $P(\boldsymbol{X}_B)$ stays the same for all neighbors of $\boldsymbol{X}_B$, the denominator $P(\boldsymbol{X}_B = \boldsymbol{x}_B^{t^*})$ can be removed from the equation.

Also,

$$P(X = x^{t^*}, \boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}) = \sum_{t=1}^{R} I(\boldsymbol{x}_B^t = \boldsymbol{x}_B^{t^*}, x^t = x^{t^*})/R$$

Thus,

$$P(X = x^{t^*} | \boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}) \propto P(X = x^{t^*}, \boldsymbol{X}_B = \boldsymbol{x}_B^{t^*}) \tag{4.17}$$

$$\propto \sum_{t=1}^{R} I(\boldsymbol{x}_B^t = \boldsymbol{x}_B^{t^*}, x^t = x^{t^*})/R \tag{4.18}$$

$$\propto \sum_{t=1}^{R} I(\boldsymbol{x}_B^t = \boldsymbol{x}_B^{t^*}, x^t = x^{t^*}) \tag{4.19}$$

And this is the final equation we are using in ComputeScore for `min_conditional` measure.

- `min_marginal`: When a random variable X is in one of its unlikely state, *i.e.*, $P(X = x^{t^*})$ is very small, then probably it is because it is being "held back" by variables that it is highly correlated to and could not move to its more likely states. We would like to add such a node to a block so that the state can transition faster. To compute marginal probability, we can get approximations from existing samples, *i.e.*,

  $$P(X = x^{t^*}) = \sum_{t=1}^{R} I(x^t = x^{t^*})/R$$

  Thus, we can compare $\sum_{t=1}^{R} I(x^t = x^{t^*})$ for nodes in a block's neighbor set, and add one with minimum score.

- `min_coverage`: As discussed earlier, one symptom of high correlation or slow convergence, is that a node's state transition is slow. If a node has 10 potential states, but in the last 10 samples only on state appeared, it might be the reason of slow convergence.

  Denote the number of states of a variable X as $|X|$, and current sampling round as T, the score for `min_converage` is calculated as

  $$S(X) = \frac{|\text{unique}(\{x^t\}_{t=T-|X|}^{T})|}{|X|}$$

  We find out the number of unique states of X in the past $|X|$ existing samples, and divide by number of all possible states $|X|$. We are not using all past samples when finding unique states of X, because when the number of samples increase, all possible states of X will likely be visited. Thus, `min_coverage` will approach 1, and the distinguishing power of `min_coverage`

will slowly diminish. We use only limited number of past samples to counter such effect. We use $|X|$ past samples instead of some fixed number K, so that variables with different number of states can have a normalized score.

We will use an example to illustrate the block growing process in Algorithm 4.3. In Figure 4.13, we show the initial state of two blocks in an MRF, one block is dark blue, the other block is dark yellow. For the blue block, the neighbor set nodes are colored with light blue, and the computed scores for the nodes are shown as the number in each node. For the yellow block, the neighbor set nodes are colored light yellow. The computed score is assigned randomly, and is only for illustration purpose, they are not real result from any of the score measures discussed above.

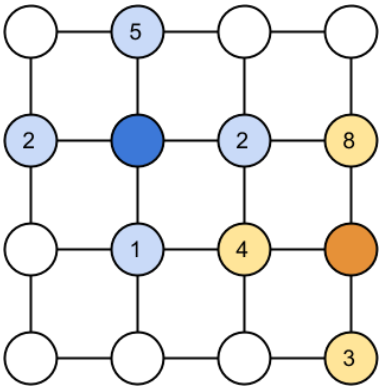
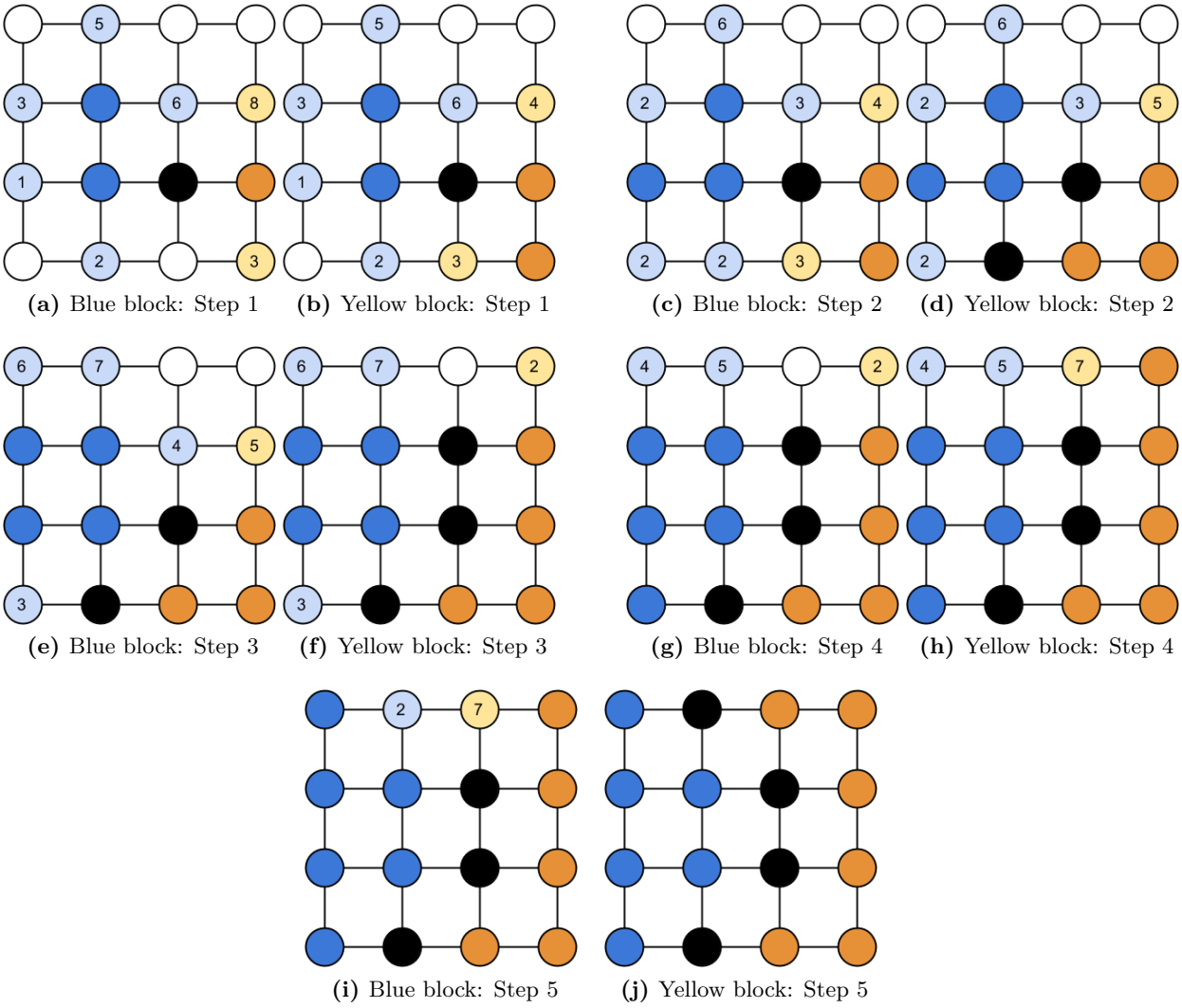


Figure 4.12: Block growth example: Initial blocks. Nodes in blocks are darker colored, and neighboring nodes are light colored. The numbers on the neighboring nodes represent the scores computed from ComputeScore (Algorithm 4.4).

In Figure 4.13, we show the growth of each block step by step. Note that block growth is done in rounds (as in Line 9 in Algorithm 4.3): Each block is given the chance to add a node from its neighbor set into the block sequentially, when all blocks are traversed, the next round of growth begins. This is done until there are no more nodes in any block's neighbor set.

In Figure 4.13a, blue block chose the node in neighbor set with smallest score, and added into the block. The neighbor set is updated, and scores for neighbor set nodes are re-calculated because the block nodes have changed. Note also that one neighbor node is changed to black, meaning it does not belong to any neighbor set and cannot be added to any block. It is because this node is neighbor to both blue and yellow block, and adding it to any block will make the two blocks adjacent. We discussed block adjacency rules in Section 4.3.7.



(a) Blue block: Step 1 **(b)** Yellow block: Step 1 **(c)** Blue block: Step 2 **(d)** Yellow block: Step 2

(e) Blue block: Step 3 **(f)** Yellow block: Step 3 **(g)** Blue block: Step 4 **(h)** Yellow block: Step 4

(i) Blue block: Step 5 **(j)** Yellow block: Step 5

Figure 4.13: Block growth example: Growing blocks. Blue and yellow blocks take turns to add a new node into the block. The new node is chosen based on score computed using one of the heuristic measure defined in Algorithm 4.4 ComputeScore. When a node is added to a block, all its neighbors that are adjacent to another block are marked black, so they cannot be added to another block. This process is continued until no more nodes can be added to blocks.

In Figure 4.13b, now the yellow block grows. It takes the same approach: Add node with minimum score, update neighbor set, and black out nodes that belong to multiple neighbor sets.

In the rest of figures, the blue and yellow block take turns to grow. In Figure 4.13d, we can see two nodes in blue block's neighbor set have the same score 2, in this case the node to be added is chosen randomly (Figure 4.13e).

In the end, in Figure 4.13j, all neighbor sets became empty, thus we stop growing the blocks. This block assignment is then used in later steps (after Line 16 in Algorithm 4.3).

4.4 Simulated Annealing

"**Simulated annealing** is a generic probabilistic metaheuristic for the global optimization problem of locating a good approximation to the global optimum of a given function in a large search space. It is often used when the search space is discrete (e.g., all tours that visit a given set of cities). For certain problems, simulated annealing may be more efficient than exhaustive enumeration – provided that the goal is merely to find an acceptably good solution in a fixed amount of time, rather than the best possible solution. The state of some physical systems, and the function E(s) to be minimized is analogous to the internal energy of the system in that state. The goal is to bring the system, from an arbitrary initial state, to a state with the minimum possible energy."

The above is an excerpt from Wikipedia[1], and it provides an excellent summary of the basics of **simulated annealing** [17, 27]. We will try to adapt simulated annealing to Gibbs sampling in probabilistic optical flow, described in Chapter 3.

In POF's Gibbs sampling, we are trying to find a sample (optical flow assignment), that has lowest energy (and thus highest likelyhood) among all possible flow assignments. We don't necessarily need to find a globally optimal solution - we only need a good enough optical flow assignment. In simulated annealing, the goal is also about bringing down the system's energy and finding a good approximation of the global optimum, which fits our goal in POF.

In simulated annealing, it tries to bring down the energy of the system by executing many rounds of state transitions. In POF, the "state" will be the current optical flow assignment $\boldsymbol{F}$.

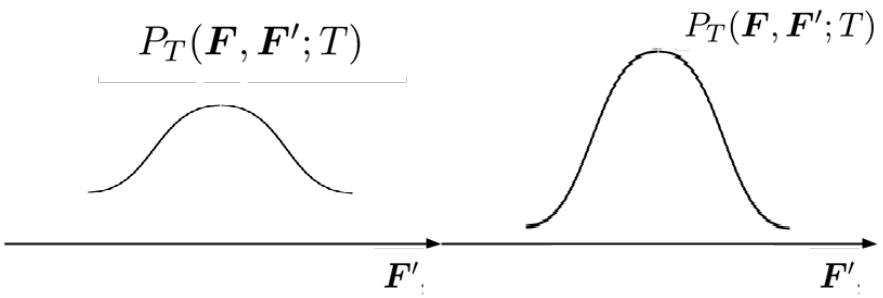
[1] http://en.wikipedia.org/wiki/Simulated_annealing

Each simulated annealing transition moves the current state from $\boldsymbol{F}$ to another state $\boldsymbol{F}'$, which is actually what Gibbs sampling is doing.

However, in simulated annealing, the state transition is guided by a temperature T, which controls the probability of transitioning from one state to another. Assume the current state is $\boldsymbol{F}$ and one possible state to transition into is $\boldsymbol{F}'$, and their respective energy is $E(\boldsymbol{F})$ and $E(\boldsymbol{F}')$. When T is high, the state transition probability from $\boldsymbol{F}$ to $\boldsymbol{F}$ is not affected by the relative quantity of $E(\boldsymbol{F})$ and $E(\boldsymbol{F}')$, *i.e.*, no matter how unlikely $\boldsymbol{F}'$ is, there is still a chance to transition to that state.

As annealing process goes on, the temperature T is gradually lowered. When T is low, the transition probability from a low energy state to a high energy state is lowered, making it more likely to transition to low energy state.

We denote the transition probability from $\boldsymbol{F}$ to $\boldsymbol{F}'$ given a temperature T as $P_T(\boldsymbol{F}, \boldsymbol{F}'; T)$. As an example, assume the possible states to transition into from $\boldsymbol{F}$ is one dimensional, and the hypothetical probability to transition into each of them when temperature T is high and low is shown in Figure 4.14.



(a) T is high **(b)** T is low

Figure 4.14: Transition probability in different temperatures. When T is lower, the probability of sampling a highly likely state is made even higher.

By introducing an additional temperature in POF's Gibbs sampling, we can speed up the algorithm. The lowering of temperature effectively does a re-scaling of probability space, increasing probability of the high probability region, and decreasing probability of low probability region.

In our implementation of simulated annealing in POF, we made a few changes to help Gibbs sampling find the optimum flow assignment faster. The modified algorithm changes temperature

in stages, hence we call it **Staged Annealing**. In Algorithm 4.5 we show the pseudocode of this process.

Algorithm 4.5 Computing Probabilistic Optical Flow with Gibbs Sampling using Staged Annealing

Input: Image $\boldsymbol{I}$, Image $\boldsymbol{I}'$, α, β, γ, observe rounds R, stage count S, temperature decay D
Output: Optical flow $\hat{\boldsymbol{f}}$

```
1: Initialize f^0: ∀i, f_i^0 = (0,0); f̂ = f^0; t̂ = 0; Round t = 1; Temperature T = 1
2: for all stage = 1...S do
3:   repeat
4:     for all F_i ∈ F do
5:       Sample f_i^t ~ P(F_i; T|f_{MB_i}) {Refer to Equation 4.20}
6:     end for
7:     if E(f^t) < E(f̂) then
8:       f̂ = f^t; t̂ = t
9:     end if
10:    t = t + 1
11:  until t − t̂ > R
12:  T = T × D; t̂ = t; f^t = f̂
13: end for
14: return f̂
```

Compare it with the Algorithm 3.1 (in page 30) that do not do annealing. In Line 2, we add an outer loop for stages of sampling. In each stage, sampling is done until no more improvements to energy is seen for R rounds. Then, we use the current optimal flow assignment $\hat{\boldsymbol{f}}$ as the input to next stage (Line 12). Also, the temperature is changed at the end of each stage (also in Line 12).

The sampling probability in Line 5 is modified to take annealing into account. It is defined as in Equation 4.20:

$$P(F_i; T|\boldsymbol{f}_{MB_i}) \propto \exp\left(-\frac{E(F_i|\boldsymbol{f}_{MB_i})}{T}\right) \tag{4.20}$$

Initially when T is 1, $P(F_i; T|\boldsymbol{f}_{MB_i}) = P(F_i|\boldsymbol{f}_{MB_i})$, thus it has no impact to the flow sampling. Later when temperature drops, the sampling will favor those states in high probability region. It can be seen more clearly when we break down 4.20 to the following:

$$
\begin{aligned}
P(F_i; T|\boldsymbol{f}_{MB_i}) &\propto \exp\left(-\frac{E(F_i|\boldsymbol{f}_{MB_i})}{T}\right) \\
&\propto (\exp(-E(F_i|\boldsymbol{f}_{MB_i})))^{\frac{1}{T}} \\
&\propto P(F_i|\boldsymbol{f}_{MB_i})^{\frac{1}{T}}
\end{aligned}
$$

Since $P(F_i|\boldsymbol{f}_{MB_i}) \in [0, 1]$, having an exponent $\frac{1}{T} > 1$ will make smaller probability even smaller, thus favoring states in high probability region.

The difference between our modified staged annealing to original simulated annealing is twofolds:

- The temperature T is changed in stages which include many state transitions, while in simulated annealing it changes after every transition. We do this to allow a more equal opportunity for states to be transitioned into within each stage.
- We use the best result in each stage as input to the next stage (Line 12), while simulated annealing always use the result from previous transition as input to the next transition. We do this so that we always have the best result within a stage to be used, not the latest result.

We will show how staged annealing helps speed up POF algorithm, in Section 6.4.

4.5 Conclusion of the Chapter

In this chapter we showed a few directions where we can speed up Gibbs sampling process, including 1) Doing better initialization, 2) Jointly sampling blocks of random variables, 3) Doing simulated annealing. We demonstrated the motivation of using these techniques, and showed how they can be incorporated into probabilistic optical flow algorithm we presented in Chapter 3. We will show experiment results of applying these speed up techniques in Chapter 6.

In the next chapter, we will discuss another important aspect of speeding up Gibbs sampling: Parallelization. We separated it into a single chapter since it deals with the implementation more than the algorithm itself, and sometimes it needs special computation framework or hardware.

Chapter 5

Parallelizing Gibbs Sampling

In this chapter, we dig into the world of parallelization, and see how we can use parallelization to speed up Gibbs sampling and POF algorithm. We will first see some compelling examples of why we need parallelization, then we will introduce parallelized computation in general. We will discuss various ways of parallelizing Gibbs sampling, and how we can apply these methods to our POF Gibbs sampling. The experiment results will be shown in Section 6.6.

5.1 Motivation

A lot of recent developments in machine learning comes with new probabilistic graphical models for various problem domains. For example, in [40], a way to model and diagnose the health of Electrical Power Systems (EPS) using Bayesian networks is described. The sensor readings of voltage, current and temperature of each component, as well as control devices in the EPS are modeled as random variables in the Bayesian network. An example Bayesian network from [40] is shown in Figure 5.1. The figure shows a small power system, for larger systems there can be easily tens of thousands of random variables.

Another example where graphical models are used is topic modeling. In [9], a method to model topics of text documents is proposed, namely Latent Dirichlet Allocation (LDA). It uses a custom MRF model to model the words, documents and topics. The graphical model can be seen in Figure 5.2.

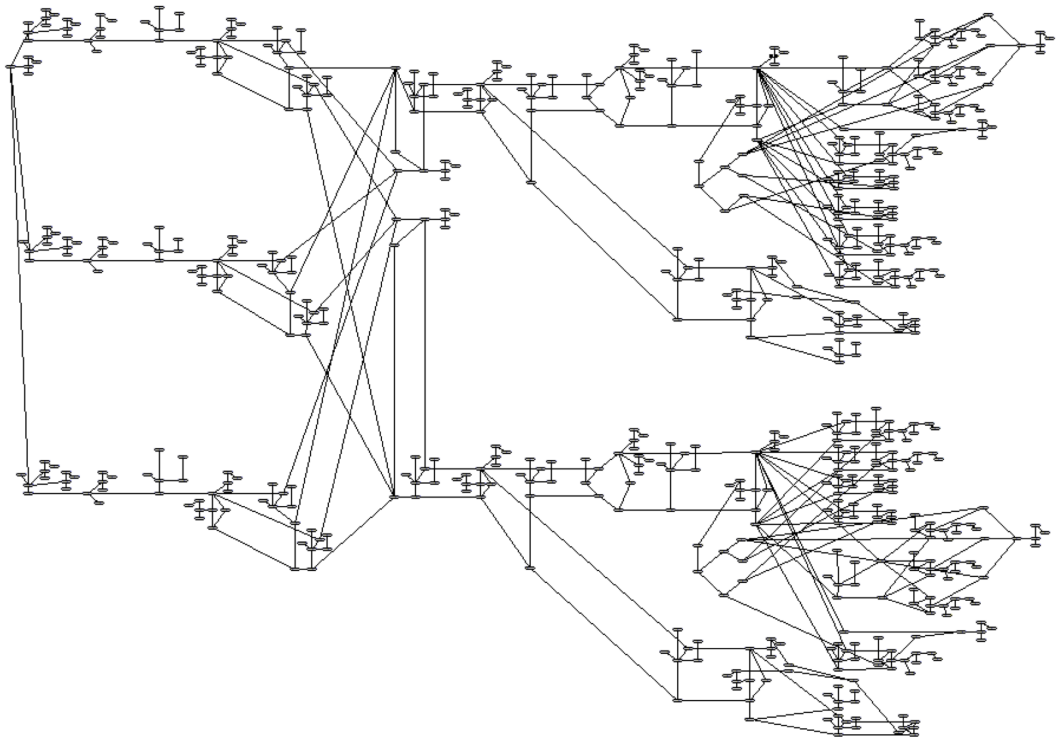


Figure 5.1: Bayesian network model for an EPS system

Here, the graphical model uses plate notation to remove duplications. See [11] for an introduction to plate notation. The rectangle box marked with D at bottom right corner represents documents, and we have D documents. Within each document, there are W words (the inner rectangle box), and each word has its own topic classification z, as well as the actual word itself w. The left rectangle box with K represents the topics we are trying to classify the words into, and ϕ represents each topic. In the entire model, only the words w are observed (thus shaded), and all other variables are up to the inference algorithm to infer their values.

As can be seen, the number of random variables in this LDA model is of size $O(WD)$. LDA and its various forms of derivatives are frequently used to model various documents for automatic topic modeling, and the size of the documents can be very huge. For example, let's say we try to model twitter's[1] tweets using LDA. As of today, there are almost 500 million tweets sent every day[2]. In one month, that will be 15 billion, which will be approximately 1.5TB in disk size, assuming the

[1] http://www.twitter.com
[2] http://www.internetlivestats.com/twitter-statistics/

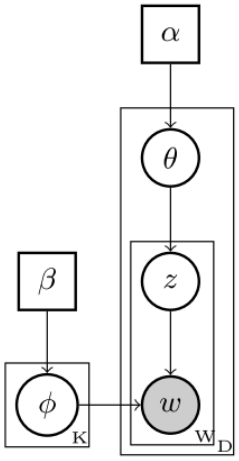


Figure 5.2: Graphical model for latent dirichlet allocation. Plate notation is used to show repeated graph structures.

average size of a tweet is 100 bytes. With this massive size, it is hard to put all data in a single computer, not to mention doing Gibbs sampling with it.

Since both the space and time complexity of Gibbs sampling is linear to the size of the input graph, we need a way to parallelize the algorithm into multiple computation units. We will discuss parallelization of algorithms in general, then see how Gibbs sampling can be parallelized.

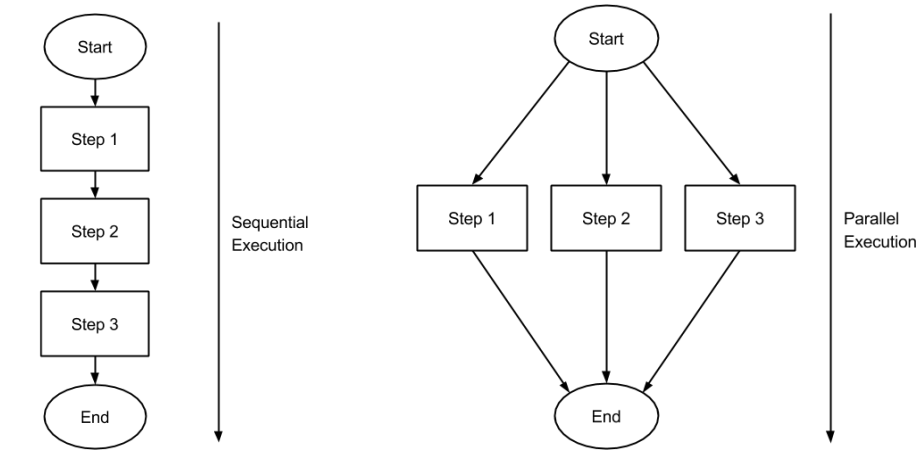
5.2 Parallelization in General

Traditionally, computers execute programs in a sequential manner. This is becuase a CPU can execute only one line of machine instruction at one time. Algorithm designers only needed to design an algorithm that runs sequentially, and try to make it run as fast as possible.

Things have changed. There are now multicore CPUs that enable multi-threaded execution of a single program. There are now GPUs that have thousands of computation units all of which can run simultaneously. There are distributed computer clusters that enable multiple computers to collaborate on a single task.

With all these new hardwares and new ways of organizing computers, algorithms don't have to run sequentially any more. We can spread the same task across multiple computation units, and make the algorithm run much faster than on a single computation unit.

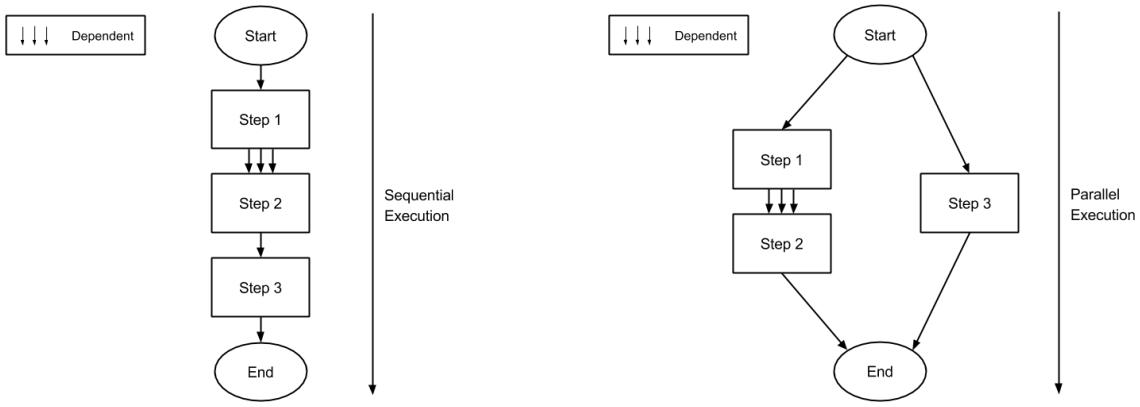
An algorithm typically contains many steps. If any of the later steps do not have to depend on the results from previous steps, they can be run on separate computation units in parallel. In Figure 5.3 we illustrated this possiblity.



(a) Sequential run **(b)** Parallel run

Figure 5.3: An algorithm ran sequentially and in parallel

However, if there are steps that depend on the results from earlier steps, they will have to be run in a sequential order, as seen in Figure 5.4.

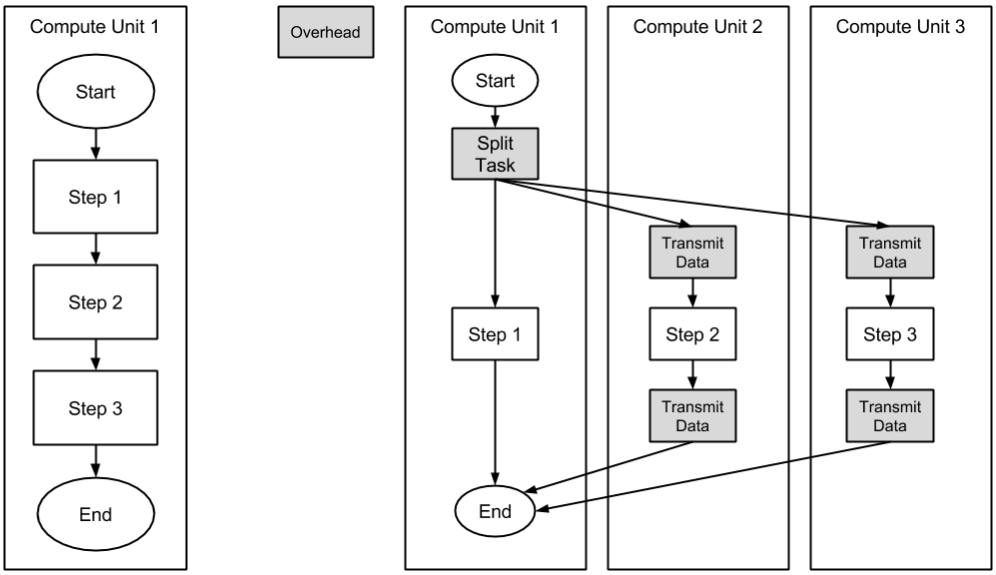


(a) Sequential run **(b)** Parallel run

Figure 5.4: An algorithm ran sequentially and in parallel, when there are dependent steps

When we are trying to run an algorithm in parallel, the first thing is to identify the steps that are dependent. We can then allocate the steps that are independent into individual computation units.

Another thing to note when designing a parallel algorithm, is the overhead associated with making the algorithm parallelized. When an algorithm is ran on a single computation unit, all the data is available in a single place. When it is parallelized, some of the data will need to be transmitted to the additional computation units. There is also overhead of splitting the task to be allocated to multiple computation units. This can be seen in Figure 5.5.



(a) Sequential **(b)** Parallelzed

Figure 5.5: Overhead with running parallel algorithm

Depending on the hardware used for transmitting the data, the time of transmission can range from negligible to significant. For example, if the computation units are different cores in a multi-core CPU, data transmission hardly takes any time, since the extra computation unit (another core) can access the data in the same way as the original core. If the computation units are separate computers connected through network, the data transmission occurs through network and will be significantly slower.

When choosing hardware architecture, we want to make sure that the speedup achieved by splitting work into multiple computation units is not shadowed by the overhead. If, for example, each step in Figure 5.3 takes 1 second to finish, then running them sequentially takes a total of 3 seconds. After splitting up the steps in 3 computation units, the time spent in splitting task and transmitting data should be less than 2 seconds for the parallelization to have any real benefit.

One way to evaluate whether parallelizing an algorithm will be beneficial or not, is to measure computation time to communication time ratio (CCR). Communication time is the time to send input of a step to the computation unit and receving output of the step from that unit. We can use CCR to roughly tell whether parallelization of a sequential algorithm is good or not. If CCR is too low, meaning there is too much overhead, we may want to adjust the parallelization strategy.

There are many more factors in evaluating whether a parallelized algorithm is beneficial, such as the total cost of running the algorithm (running on extra hardware may cause extra electricity bill, hardware rental bill, etc), time spent in developing parallel algorithm, etc. All these should be part of the consideration when trying to parallelize an algorithm.

5.3 Parallelizing Gibbs Sampling

5.3.1 Markov Chain Monte Carlo (MCMC)

Gibbs sampling is part of a larger family of sampling methods, called **Markov Chain Monte Carlo (MCMC)** methods [36]. MCMC's goal is to transition an initial probability distribution $P^0(\boldsymbol{X})$ to a target probability distribution $P(\boldsymbol{X})$ with a series of probabilistic transitions $T(\boldsymbol{X}'|\boldsymbol{X})$, each transition converting a previous probability distribution $P^t(\boldsymbol{X})$ to a new probability distribution $P^{t+1}(\boldsymbol{X})$, defined by

$$P^{t+1}(\boldsymbol{X}') = \int_{\boldsymbol{X}} P^t(\boldsymbol{X}) T(\boldsymbol{X}'|\boldsymbol{X}) d\boldsymbol{X}$$

MCMC is able to bring an arbitrary initial distribution $P^0(\boldsymbol{X})$ to the target distribution $P(\boldsymbol{X})$ as $t \to \infty$, as long as the target distribution $P(\boldsymbol{X})$ and the transition probability $T(\boldsymbol{X}'|\boldsymbol{X})$ satisfies the following properties:

- $P(\boldsymbol{X})$ is an **invariant distribution** of $T(\boldsymbol{X}'|\boldsymbol{X})$. It means $P(\boldsymbol{X})$ stays the same, after being transitioned by $T(\boldsymbol{X}'|\boldsymbol{X})$, in other words,

$$P(\boldsymbol{X}') = \int_{\boldsymbol{X}} P(\boldsymbol{X}) T(\boldsymbol{X}'|\boldsymbol{X}) d\boldsymbol{X} \tag{5.1}$$

- The Markov chain is **ergodic**. That is,

$$\forall P^0(\boldsymbol{X}), \lim_{t\to\infty} P^t(\boldsymbol{X}) = P(\boldsymbol{X}) \tag{5.2}$$

When we try to prove $P(\boldsymbol{X})$ is an invariant distribution of $T(\boldsymbol{X}'|\boldsymbol{X})$, we can prove $P(\boldsymbol{X})$ has detailed balance property instead. **Detailed balance** property means the probability of picking a random state $\boldsymbol{x}_a$ from the target distribution $P(\boldsymbol{X})$ and then transitioning it to a state $\boldsymbol{x}_b$ is equal to picking a state $\boldsymbol{x}_b$ from $P(\boldsymbol{X})$ and then transitioning it to state $\boldsymbol{x}_a$. In other words,

$$P(\boldsymbol{x}_a)T(\boldsymbol{x}_b|\boldsymbol{x}_a) = P(\boldsymbol{x}_b)T(\boldsymbol{x}_a|\boldsymbol{x}_b) \tag{5.3}$$

When $P(\boldsymbol{X})$ is detailed balanced with $T(\boldsymbol{X}'|\boldsymbol{X})$, we can easily prove it is an invariant distribution of $T(\boldsymbol{X}'|\boldsymbol{X})$:

$$\begin{aligned} P(\boldsymbol{X}') &= P(\boldsymbol{X}') \cdot 1 \\ &= P(\boldsymbol{X}') \cdot \int_{\boldsymbol{X}} T(\boldsymbol{X}|\boldsymbol{X}')d\boldsymbol{X} \\ &= \int_{\boldsymbol{X}} P(\boldsymbol{X}')T(\boldsymbol{X}|\boldsymbol{X}')d\boldsymbol{X} \\ &= \int_{\boldsymbol{X}} P(\boldsymbol{X})T(\boldsymbol{X}'|\boldsymbol{X})d\boldsymbol{X} \end{aligned}$$

The last step is possible because of detailed balance property, and the last equation is the definition of invariant distribution.

The sequential Gibbs sampling algorithm introduced in Chapter 2 Page 19 can be proved to satisfy detailed balance property, as follows.

Assume the two distinct random states we have chosen are $\boldsymbol{x}$ and $\boldsymbol{x}'$. There are two cases:

- $\boldsymbol{x}$ and $\boldsymbol{x}'$ differ by more than one variable. In sequential Gibbs sampling, only one variable is changed in a transition, so in this case $T(\boldsymbol{x}'|\boldsymbol{x}) = T(\boldsymbol{x}|\boldsymbol{x}') = 0$, meaning it is impossible to transition between $\boldsymbol{x}$ and $\boldsymbol{x}'$. Thus,

$$P(\boldsymbol{x})T(\boldsymbol{x}'|\boldsymbol{x}) = P(\boldsymbol{x}')T(\boldsymbol{x}|\boldsymbol{x}') = 0$$

- $\boldsymbol{x}$ and $\boldsymbol{x}'$ differ by only one variable, assume it is x_D. We will denote all other variables as $\boldsymbol{x}_{-D}$. Then, $\boldsymbol{x} = \{x_D, \boldsymbol{x}_{-D}\}$, $\boldsymbol{x}' = \{x'_D, \boldsymbol{x}_{-D}\}$.

  The transition probability from $\boldsymbol{x}$ to $\boldsymbol{x}'$ will be

$$\begin{aligned} T(\boldsymbol{x}'|\boldsymbol{x}) &= T(x'_D, \boldsymbol{x}_{-D}|x_D, \boldsymbol{x}_{-D}) \\ &= T(x'_D|x_D, \boldsymbol{x}_{-D}) \cdot T(\boldsymbol{x}_{-D}|x_D, \boldsymbol{x}_{-D}) \\ &= P(x'_D|x_D, \boldsymbol{x}_{-D}) \cdot 1 \qquad (5.4) \\ &= P(x'_D|\boldsymbol{x}_{-D}) \qquad (5.5) \end{aligned}$$

  In Equation 5.4, $T(x'_D|x_D, \boldsymbol{x}_{-D})$ is changed to $P(x'_D|x_D, \boldsymbol{x}_{-D})$, since by definition of Gibbs sampling, the state transition is made by sampling the new value of a node based on current value of all other variables. Also, since $\boldsymbol{x}_{-D}$ is not changing, the probability of transitioning from $\boldsymbol{x}_{-D}$ to the same values is 1. In Equation 5.5, x'_D only depends on $\boldsymbol{x}_{-D}$, thus x_D is removed from conditional term.

  Similarly, we can also get

$$T(\boldsymbol{x}|\boldsymbol{x}') = P(x_D|\boldsymbol{x}_{-D}) \qquad (5.6)$$

  Thus,

$$\begin{aligned} P(\boldsymbol{x})T(\boldsymbol{x}'|\boldsymbol{x}) &= P(x_D, \boldsymbol{x}_{-D})P(x'_D|\boldsymbol{x}_{-D}) && \text{Apply Equation 5.5} \\ &= P(x_D|\boldsymbol{x}_{-D})P(\boldsymbol{x}_{-D})P(x'_D|\boldsymbol{x}_{-D}) \\ &= P(x_D|\boldsymbol{x}_{-D})P(x'_D, \boldsymbol{x}_{-D}) \\ &= T(\boldsymbol{x}|\boldsymbol{x}')P(\boldsymbol{x}') && \text{Apply Equation 5.6} \end{aligned}$$

Since in both situations we have $P(\boldsymbol{x})T(\boldsymbol{x}'|\boldsymbol{x}) = P(\boldsymbol{x}')T(\boldsymbol{x}|\boldsymbol{x}')$, the detailed balance property is proved for sequential Gibbs sampling. Thus, Gibbs sampling can bring arbitrary initial distribution $P^0(\boldsymbol{X})$ to the target distribution $P(\boldsymbol{X})$, if $P(\boldsymbol{X})$ is ergodic.

5.3.2 Designing Correct Parallel Gibbs Sampling Method

When we are converting the sequential Gibbs sampling algorithm to parallelized version, it is important to make sure the parallelized algorithm still satisfies the properties required by MCMC. Specifically, detailed balance property needs to be satisfied.

One tempting way to parallelize sequential Gibbs sampling, is to simultaneously sample all variables in one round of sampling. The pseudocode is shown in Algorithm 5.1.

Algorithm 5.1 Hypothetic Fully Parallel Gibbs Sampling

```
1: for all t ∈ 1, 2, ..., N do
2:    for all X_i do in parallel
3:       Sample X_i^(t+1) ~ P(X_i | x^t_{MB_i})
4:    end for
5: end for
```

This method actually does not satisfy detailed balance property. We can see it from the following deduction. Since in Algorithm 5.1 we are sampling all variables at the same time, based on the current values of the variables,

$$T(\boldsymbol{x}'|\boldsymbol{x}) = \prod_i P(x'_i|\boldsymbol{x}_{-i})$$

$$T(\boldsymbol{x}|\boldsymbol{x}') = \prod_i P(x_i|\boldsymbol{x}'_{-i})$$

Thus,

$$P(\boldsymbol{x})T(\boldsymbol{x}'|\boldsymbol{x}) = P(\boldsymbol{x})\prod_i P(x_i'|\boldsymbol{x}_{-i})$$

$$P(\boldsymbol{x}')T(\boldsymbol{x}|\boldsymbol{x}') = P(\boldsymbol{x}')\prod_i P(x_i|\boldsymbol{x}'_{-i})$$

The two equations are not equal. We can use a simple example to verify their difference: Assume we have a two node probability distribution $P(A, B)$ with the following factor table:

A	B	$\phi(A, B)$
a	b	w
a	b'	x
a'	b	y
a'	b'	z

Assuming $\boldsymbol{x} = (a, b)$ and $\boldsymbol{x}' = (a', b')$, then,

$$\begin{aligned} P(\boldsymbol{x})T(\boldsymbol{x}'|\boldsymbol{x}) &= P(\boldsymbol{x})\prod_i P(x_i'|\boldsymbol{x}_{-i}) = P(a,b)P(a'|b)P(b'|a) \\ &= \frac{w}{w+x+y+z}\frac{y}{w+y}\frac{x}{w+x} \end{aligned}$$

$$\begin{aligned} P(\boldsymbol{x}')T(\boldsymbol{x}|\boldsymbol{x}') &= P(\boldsymbol{x}')\prod_i P(x_i|\boldsymbol{x}'_{-i}) = P(a',b')P(a|b')P(b|a') \\ &= \frac{z}{w+x+y+z}\frac{x}{x+z}\frac{y}{y+z} \end{aligned}$$

Which are apparently different.

5.4 Chromatic Gibbs Sampling

Joseph et al. [19] proposed a way to sample groups of variables in a PGM in parallel, called **Chromatic Gibbs sampling**. In this method, the nodes in PGM are colored so that no adjacent (neighboring) nodes have the same color, then in a Gibbs sampling round, we go through each color

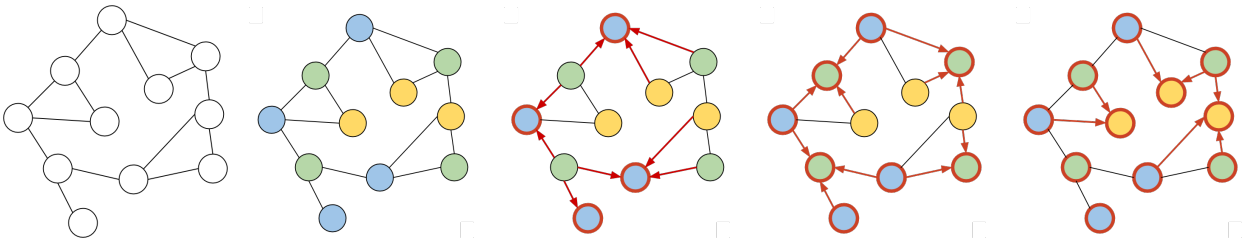
sequentially, and sample all nodes in that color in parallel. The pseudocode for this algorithm is shown in Algorithm 5.2.

Algorithm 5.2 Chromatic Gibbs Sampling

Input: PGM $G(\boldsymbol{X}, \boldsymbol{E}, \boldsymbol{\phi})$

$\{\boldsymbol{X}_i\}_{i=1}^{C} = \textsc{MinColor}(\text{G})$: Color nodes $\boldsymbol{X}$ into color groups $\{\boldsymbol{X}_i\}, i = 1, \cdots, C$, such that $\forall i \neq j, \boldsymbol{X}_i \cap \boldsymbol{X}_j = \varnothing$, $\cup_i \boldsymbol{X}_i = \boldsymbol{X}$, $\forall X_p, X_q \in \boldsymbol{X}_i, (X_p, X_q) \notin \boldsymbol{E}$, and C is minimum.
for all $t \in 1, 2, \cdots, N$ **do**
  for all $i \in 1, 2, \cdots, C$ **do**
    for all $X_p \in \boldsymbol{X}_i$ **do in parallel**
      Sample $X_p^{(t+1)} \sim P(X_p | \boldsymbol{x}_{MB_p}^{current})$
    end for
  end for
end for

We show an example of running chromatic Gibbs sampling, in Figure 5.6.



(a) Initial PGM **(b)** Minimum colored PGM **(c)** Sample blue nodes **(d)** Sample green nodes **(e)** Sample yellow nodes

Figure 5.6: Example of running chromatic Gibbs sampling. Nodes are first minimum colored (b), then we iterate through each color, and sample nodes with the same color in parallel, using their Markov blankets (red edges).

In Figure 5.6b, the original PGM is colored using three colors, where nodes with the same color are not adjacent to each other. Then, in the sampling round, we iterate over each color sequentially, and sample nodes in each color in parallel.

Since when we are sampling nodes in the same color, none of their Markov blanket neighbors contain any nodes with the same color, sampling all of them in parallel yields the same result as when sampling each of them sequentially, which can help us understand why chromatic Gibbs sampling is a correct way to parallelize Gibbs sampling.

Formally to prove chromatic Gibbs sampling is correct MCMC algorithm, we need to prove it satisfies detailed balance property. Assume we are sampling color group $\boldsymbol{X}_c$, then the collective

Markov blanket of $\boldsymbol{X}_c$, denoted as $\boldsymbol{X}_{MB_c}$, will be disjoint from $\boldsymbol{X}_c$, *i.e.*, $\boldsymbol{X}_c \cap \boldsymbol{X}_{MB_c} = \varnothing$.

Thus,

$$T_C(\boldsymbol{X}'|\boldsymbol{X}) = T_C(\boldsymbol{X}_c|\boldsymbol{X}_{MB_c}) = \prod_{X_i \in \boldsymbol{X}_c} T_S(X_i|\boldsymbol{X}_{MB_c})$$

We can see that the transition probability of chromatic Gibbs sampling, T_C, is a multiplication of transition probability of sequential Gibbs sampling T_S. We have proved $P(\boldsymbol{X})$ is detailed balanced with T_S in previous section. In MCMC, if a transition is a concatenation of other transitions that already satisfy detailed balance property, then it will satisfy detailed balance property itself. Thus, chromatic Gibbs sampling is valid as a parallel Gibbs sampling algorithm.

In sequential Gibbs sampling, the time complexity of running a single round of sampling is $O(N)$, with N being number of nodes. In chromatic sampling, given we have P processors and the graph is colored with C colors, each color containing N_c nodes, the time complexity becomes

$$O\left(\sum_c \lceil \frac{N_c}{P} \rceil\right) = O\left(\sum_c (\frac{N_c}{P} + 1)\right) = O\left(\frac{N}{P} + C\right) \tag{5.7}$$

Thus, as long as C is small, we can speed up sequential Gibbs sampling by a factor of P. This is also the reason why we want to color the graph with minimum colors (Line 1 in Algorithm 5.2).

Unfortunately, minimum coloring a general graph is NP-Complete. For many types of graphs, however, it is easy to color them with minimum colors. For example, the grid MRF for POF algorithm (Figure 3.3a on page 28) is trivially two colorable. Even for graphs without easy way to find minimum coloring, it is found that simple heuristic graph coloring method perform well in practice [19].

Now let's try to apply chromatic Gibbs sampling to our POF algorithm (Algorithm 3.1). The MRF for POF is a grid model that can be trivially two colored, thus if we have enough processors, and assuming graph data are instantly available to all of them (*i.e.*, no communication overhead), the original $O(N)$ complexity sampling in each round can be done in $O(1)$ complexity. In actual implementation, we don't have infinite processors, also transferring data between computation units do cost time, thus the real speedup won't be that dramatic. We will show comparisons in Section

6.6.

5.5 Parallel Block Gibbs Sampling

In Section 4.3 we discussed block Gibbs sampling as a way to speed up Gibbs sampling convergence. We devised way to detect highly correlated nodes, and sample them jointly after caliberating the block's clique tree.

Here we will first prove the block Gibbs sampling algorithm satisfies detailed balance property, then try to find a way to parallelize it.

In block Gibbs sampling (Algorithm 4.1), we decide on the blocks $\{\boldsymbol{X}_b\}$ before each round of sampling starts, and nodes within each block will be sampled jointly. The blocks are gone through sequentially, so we only need to prove one transition, *i.e.*, sampling one block, satisfies detailed balance property.

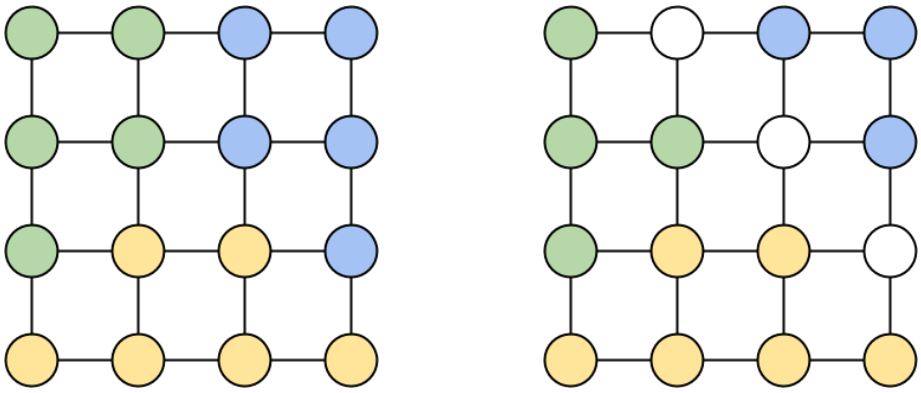
Assume the current block to sample is $\boldsymbol{X}_B$, and all other nodes is $\boldsymbol{X}_{-B}$. We have the following proof:

$$\begin{aligned} P(\boldsymbol{x})T(\boldsymbol{x}'|\boldsymbol{x}) &= P(\boldsymbol{x}_B, \boldsymbol{x}_{-B})P(\boldsymbol{x}'_B|\boldsymbol{x}_{-B}) \\ &= P(\boldsymbol{x}_B|\boldsymbol{x}_{-B})P(\boldsymbol{x}_{-B})P(\boldsymbol{x}'_B|\boldsymbol{x}_{-B}) \\ &= P(\boldsymbol{x}_B|\boldsymbol{x}_{-B})P(\boldsymbol{x}'_B, \boldsymbol{x}_{-B}) \\ &= T(\boldsymbol{x}|\boldsymbol{x}')P(\boldsymbol{x}') \end{aligned}$$

It is almost identical to the proof of sequential Gibbs sampling, except that now we replaced a single node with a block of nodes.

Now let's think about how to parallelize block Gibbs sampling. Take a look at Figure 5.7. The natural way will be trying to sample all the blocks in parallel. Recall Algorithm 5.1 where we sample all nodes in parallel, and we proved that it doesn't satisfy detailed balance property. When the blocks are generated so that they are adjacent to each other (Figure 5.7a), and we sample all

of them in parallel, it will have the same problem. However, if we leave a margin among the blocks (Figure 5.7b), we will be able to sample these blocks in parallel.



(a) Blocks adjacent, cannot sample blocks in parallel

(b) Blocks with margins, can sample blocks in parallel

Figure 5.7: How different blocking affects parallelization. Only blocks with nodes that are not adjacent can be sampled in parallel. For example, blue and yellow block can be sampled together, or blue and green, but not green and yellow, or all three together.

The parallelized version of block Gibbs sampling algorithm (Algorithm 4.1) is shown in Algorithm 5.3. In Line 7, the sequential sampling of individual nodes can also incorporate chromatic sampling to have more speed up.

Algorithm 5.3 Parallel Block Gibbs Sampling General Framework

1: **for all** $t \in 1, 2, \ldots, N$ **do**
2: Decide on blocks $\boldsymbol{B} = \{B_i\}, B_i = \boldsymbol{X}_{B_i}, \cup_i \boldsymbol{X}_{B_i} \subseteq \boldsymbol{X}$ so that no blocks are adjacent
3: **for all** B_i **do in parallel**
4: Jointly sample $\boldsymbol{X}_{B_i}^{(t+1)} \sim P(\boldsymbol{X}_{B_i} | \boldsymbol{x}_{MB_{B_i}}^{current})$
5: **end for**
6: **for all** $X_i \in \boldsymbol{X} - \cup_j \boldsymbol{X}_{B_j}$ **do**
7: Sample $X_i^{(t+1)} \sim P(X_i | \boldsymbol{x}_{MB_i}^{current})$
8: **end for**
9: **end for**

In [19], a way to dynamically generate "splashes" of nodes are proposed, and the splashes are sampled in parallel. The splashes are the same as blocks we have been discussing. In their work, the splashes are generated so that each splash's treewidth is always bounded, which can guarantee reasonable runtime for jointly sampling nodes in a splash using junction tree algorithm.

When we parallellize block Gibbs sampling, another aspect of consideration is hardware and

architecture choice. As we discussed in Section 5.2, the overhead of splitting work into extra computation units should be less than the speedup we try to achieve. In our situation, the part that's being sent to individual computation unit, is the sampling of individual blocks.

The size of blocks depend roughly on the size of the entire graph and the number of blocks we are trying to generate. Also, the workload of sampling each block is associated with the treewidth of the block. Before we send any blocks to compute, we can filter them by their size, such as number of nodes and treewidth. We do this to control the amount of computation involved for sampling each block.

Then, depending on the size we have chosen for each block, we can decide on the hardware architecture. For example, if each block is huge, we can choose to distribute the workload into multiple computers across network, since the communication time over network is negligible compared to sampling time for each block. However, when block is small, distributing them over network can be costly, and we may opt to use multicore architecture to parallelize the algorithm.

We will experiment with different block sizes, number of computation units, and test out the speedup in experiment Section 6.6.

Chapter 6

Experiments

In this Chapter, we present experimental results for methods presented in earlier chapters. First we will show how the Probabilistic Optical Flow (POF) algorithm compares with other state-of-the-art optical flow algorithms. Then we will apply various speedup techniques, such as doing clever initialization, using staged annealing, doing block Gibbs sampling and parallelization, and see how they can improve algorithm speed.

Various concepts and their corresponding experimental sections are summarized in Table 6.1.

Section Content	Concept Introduced	Experiment Section
Compare POF algorithm with others	Chapter 3	Section 6.1
Initialization	Section 4.2	Section 6.3
Staged annealing	Section 4.4	Section 6.4
Block Gibbs sampling	Section 4.3	Section 6.5
Parallelization	Chapter 5	Section 6.6

Table 6.1: Overview of experiments.

6.1 POF Compared With Other Optical Flow Methods

In this section, we first present the metrics we will use to evaluate optical flow algorithms in Section 6.1.1. One metric uses interpolated image of the original image, thus we will discuss how to do the interpolation in Section 6.1.2. Discussion of experimental datasets is in Section 6.1.3.

Experiments will be split into several categories, to evaluate different aspects of the algorithms.

Using synthetic datasets, we evaluate optical flow algorithms' ability to handle different shapes (Section 6.1.4) and different movement patterns (Section 6.1.5). Experiments with real-world images will be presented in Section 6.1.6 and Section 6.1.7.

6.1.1 Evaluation Metrics and Methods

The "Middlebury" dataset[1] has became very popular in the optical flow research field as the standard evaluation dataset, and we will be using it in our experiments. The Middlebury dataset has an accompanying survey paper [5], which presents a few metrics that measure the quality of optical flow algorithms. In their website[2], they allow optical flow algorithm programs to be submitted, and use a test set with hidden ground truth to evaluate submitted algorithms. The up-to-date rankings of algorithms are presented on the website, sorted by different metrics.

Four metrics are discussed in [5], *i.e.*, **Endpoint Error (EE), Angular Error (AE), Interpolation Error (IE) and Normalized Interpolation Error (NE)**. We implemented the first three - EE, AE and IE - as our evaluation metrics. We now describe each metric in detail.

- **Endpoint Error (EE)**: Originally used by Otte et al. [37], endpoint error measures the length of flow vector difference. Denote a flow vector for a pixel as $f(p) = (u, v)$ for 2-D images, and the ground truth flow vector as $f_{GT} = (u_{GT}, v_{GT})$, then

$$\mathrm{EE}(p) \equiv \sqrt{(u - u_{GT})^2 + (v - v_{GT})^2} \tag{6.1}$$

  The endpoint error for the entire flow field can be summarized by a few statistics, including mean, standard deviations, and RX statistics. RX denotes the percentage of pixels that have an error measure above X. For example, $R2.5$ means the following:

$$\mathrm{EE}_{R2.5} \equiv \frac{\sum_p I(\mathrm{EE}(p) > 2.5)}{N} \tag{6.2}$$

  We will use EE to denote the mean of all $\mathrm{EE}(p)$, and EE_{SD} as the standard deviation. In Figure 6.1 we show part of the EE ranking on the Middlebury website.

[1] http://vision.middlebury.edu/flow/data/
[2] http://vision.middlebury.edu/flow/eval/

Average endpoint error	avg. rank	Army (Hidden texture) GT im0 im1 all		disc		untext		Mequon (Hidden texture) GT im0 im1 all		disc		untext		Schefflera (Hidden texture) GT im0 im1 all		disc		untext	
NNF-Local [87]	2.7	**0.07**	1	0.20	2	**0.05**	1	**0.15**	1	0.51	3	0.12	5	**0.18**	1	**0.37**	1	**0.14**	1
OFLAF [77]	6.9	0.08	7	0.21	3	0.06	5	0.16	5	0.53	4	0.12	5	0.19	2	**0.37**	1	**0.14**	1
MDP-Flow2 [68]	7.9	0.08	7	0.21	3	0.07	14	**0.15**	1	**0.48**	1	**0.11**	1	0.20	4	0.40	4	**0.14**	1
NN-field [71]	8.7	0.08	7	0.22	14	**0.05**	1	0.17	7	0.55	6	0.13	10	0.19	2	0.39	3	0.15	6
ComponentFusion [96]	10.0	**0.07**	1	0.21	3	**0.05**	1	0.16	5	0.55	6	0.12	5	0.20	4	0.44	7	0.15	6
WLIF-Flow [93]	14.7	0.08	7	0.21	3	0.06	5	0.18	9	0.55	6	0.15	19	0.25	13	0.56	14	0.17	12
TC/T-Flow [76]	15.0	**0.07**	1	0.21	3	**0.05**	1	0.19	14	0.68	27	0.12	5	0.28	18	0.66	23	**0.14**	1
Layers++ [37]	16.5	0.08	7	0.21	3	0.07	14	0.19	14	0.56	9	0.17	26	0.20	4	0.40	4	0.18	18
LME [70]	17.0	0.08	7	0.22	14	0.06	5	**0.15**	1	0.49	2	**0.11**	1	0.30	26	0.64	18	0.31	69
IROF++ [58]	17.5	0.08	7	0.23	19	0.07	14	0.21	26	0.68	27	0.17	26	0.28	18	0.63	17	0.19	30
nLayers [57]	17.6	**0.07**	1	**0.19**	1	0.06	5	0.22	32	0.59	12	0.19	45	0.25	13	0.54	11	0.20	38
FC-2Layers-FF [74]	19.7	0.08	7	0.21	3	0.07	14	0.21	26	0.70	31	0.17	26	0.20	4	0.40	4	0.18	18
Correlation Flow [75]	20.1	0.09	28	0.23	19	0.07	14	0.17	7	0.58	11	**0.11**	1	0.43	47	0.99	50	0.15	6
AGIF+OF [85]	21.1	0.08	7	0.22	14	0.07	14	0.23	44	0.73	35	0.18	36	0.28	18	0.66	23	0.18	18
Classic+CPF [83]	22.8	0.08	7	0.23	19	0.07	14	0.22	32	0.73	35	0.17	26	0.30	26	0.70	26	0.18	18
FESL [72]	22.9	0.08	7	0.21	3	0.07	14	0.25	54	0.75	41	0.19	45	0.27	15	0.61	15	0.18	18
ALD-Flow [66]	23.2	**0.07**	1	0.21	3	0.06	5	0.19	14	0.64	21	0.13	10	0.30	26	0.73	29	0.15	6
TC-Flow [46]	23.4	**0.07**	1	0.21	3	0.06	5	**0.15**	1	0.59	12	**0.11**	1	0.31	31	0.78	34	**0.14**	1
COFM [59]	23.4	0.08	7	0.26	36	0.06	5	0.18	9	0.62	17	0.14	15	0.30	26	0.74	31	0.19	30
Sparse-NonSparse [56]	23.7	0.08	7	0.23	19	0.07	14	0.22	32	0.73	35	0.18	36	0.28	18	0.64	18	0.19	30
Efficient-NL [60]	23.9	0.08	7	0.22	14	0.06	5	0.21	26	0.67	25	0.17	26	0.31	31	0.73	29	0.18	18
LSM [39]	25.3	0.08	7	0.23	19	0.07	14	0.22	32	0.73	35	0.18	36	0.28	18	0.64	18	0.19	30
Ramp [62]	25.8	0.08	7	0.24	26	0.07	14	0.21	26	0.72	33	0.18	36	0.27	15	0.62	16	0.19	30
Classic+NL [31]	27.6	0.08	7	0.23	19	0.07	14	0.22	32	0.74	39	0.18	36	0.29	23	0.65	22	0.19	30

Figure 6.1: Partial ranking of optical flow algorithms on the Middlebury website, ranked by the average ranking of endpoint errors (EE) among different input images.

- **Angular Error (AE)**: Originally used in a survey by Barren et al. [6], the angular error of a pixel AE(p) measures the angle difference between a flow vector and ground truth vector:

$$\mathrm{AE}(p) \equiv \arccos\left(\frac{1.0 + u \cdot u_{GT} + v \cdot v_{GT}}{\sqrt{1.0 + u^2 + v^2}\sqrt{1.0 + u_{GT}^2 + v_{GT}^2}}\right) \tag{6.3}$$

It is defined as the angle between 3-D vector $(u, v, 1.0)$ and $(u_{GT}, v_{GT}, 1.0)$. We are using 3-D vector with an extra Z-dimension length as 1.0, to deal with situations when one of the 2-D vector has zero length. However, the AE error penalizes longer flow vectors less than shorter flow vectors, and the choice of Z-dimension length is a bit arbitrary [5].

Similar to EE, we use AE to denote the mean of all angular errors, and AE_{SD} for standard deviation.

- **Interpolation Error (IE)**: Given an image $\boldsymbol{I}$ and optical flow $\boldsymbol{f}$, if we move all pixels in $\boldsymbol{I}$ based on $\boldsymbol{f}$, the result image is called an interpolation of $\boldsymbol{I}$ using $\boldsymbol{f}$. We denote the interpolated image as $\boldsymbol{I}_{INT(\boldsymbol{f})}$.

  Given first image $\boldsymbol{I}$, second image $\boldsymbol{I'}$ and calculated flow $\boldsymbol{f}$, interpolation error IE represents the difference between second image $\boldsymbol{I'}$ and interpolated image $\boldsymbol{I}_{INT(\boldsymbol{f})}$:

$$\text{IE} \equiv \sqrt{\frac{1}{N}\sum_{p}\left(\boldsymbol{I'}(p) - \boldsymbol{I}_{INT(\boldsymbol{f})}(p)\right)^2} \tag{6.4}$$

  We will describe the interpolation algorithm we used in Section 6.1.2.

When selecting optical flow algorithms to compare with the POF algorithm, we would like to use algorithms that both have good performances in the Middlebury evaluation dataset, and have open source code available. In the Middlebury ranking page,[3] algorithms can be ranked by any of the metrics presented above - AE, EE or IE. We chose the following algorithms: **Classic+NL (C+NL) [44]**[4] (Highly ranked in AE and EE), **Correlation Flow (CF) [13]**[5] (Highly ranked in AE and EE), **CLG-TV [12]**[6] (Highly ranked in IE), as well as one classic optical flow algorithm, the **Horn Schunck method (HS) [20]**[7].

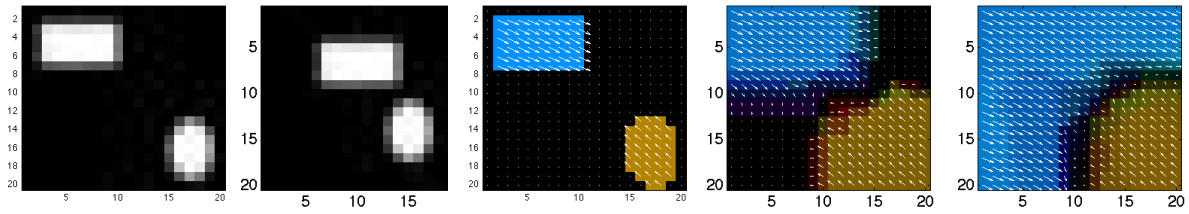
For each algorithm, we use the default settings provided in the respective open source code. For the POF algorithm, we use $\alpha = 3, \beta = 100, \gamma = 1$ for most experiments, unless otherwise noted. For the α parameter, a smaller value enforces a tighter control on intensity conformity. The β parameter is set to a large value for less restriction on movement distances. A small value of the γ parameter makes neighboring pixels more conformant with each other, thus making flow fields smoother.

Some optical flow algorithms are probabilistic, such as the POF algorithm, meaning they generate different results on different runs. For such algorithms, we run the same experiment multiple times and take the average for the metrics measured.

When implementing error metrics AE and EE, we only consider areas where we know the ground

[3] http://vision.middlebury.edu/flow/eval/results/results-e1.php
[4] http://www.cs.brown.edu/~dqsun/research/software.html
[5] http://cv.utcluj.ro/optical-flow.html
[6] http://www.cv.utcluj.ro/optical-flow.html
[7] http://www.mathworks.com/matlabcentral/fileexchange/22756-horn-schunck-optical-flow-method

truth flow. The reason can be seen in Figure 6.2. Here, the rectangle moved in the bottom right direction, and the ellipse moved in the top left direction. We only know the ground truth flow for the two objects, but not the background. Figure 6.2d and Figure 6.2e shows results from the POF and CLG-TV algorithms. They differ quite a lot, but in the area where we know the ground truth, both of them have the same results, thus they should have the same error result. For this reason, we will only compute AE and EE in areas where we know the ground truth flows.

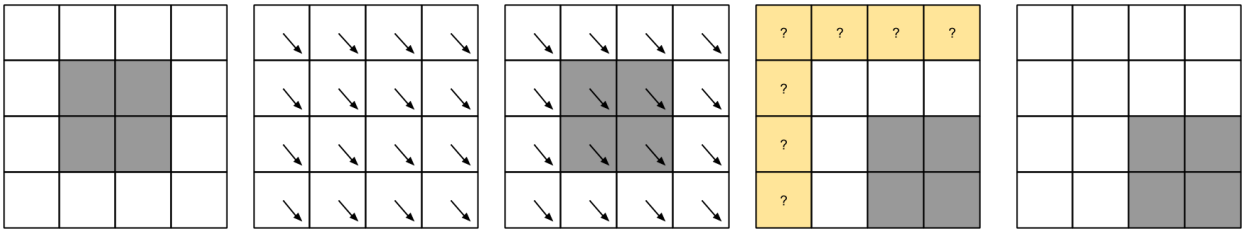


(a) Image 1 **(b)** Image 2 **(c)** Ground truth **(d)** POF result **(e)** CLG-TV result

Figure 6.2: Error metric implementation. Here ground truth is known for the objects (rectangle and ellipse), but not for the black background. When measuring AE and EE of different flow results (Figure (d) and (e)) only flow values in the area of known ground truth are measured.

6.1.2 Interpolation

Given an image $\boldsymbol{I}$ and an optical flow $\boldsymbol{f}$, an interpolation algorithm $f_{INT}(\boldsymbol{I}, \boldsymbol{f})$ maps the image $\boldsymbol{I}$ to an interpolated image $\boldsymbol{I}_{INT(\boldsymbol{f})}$, based on the optical flow $\boldsymbol{f}$.



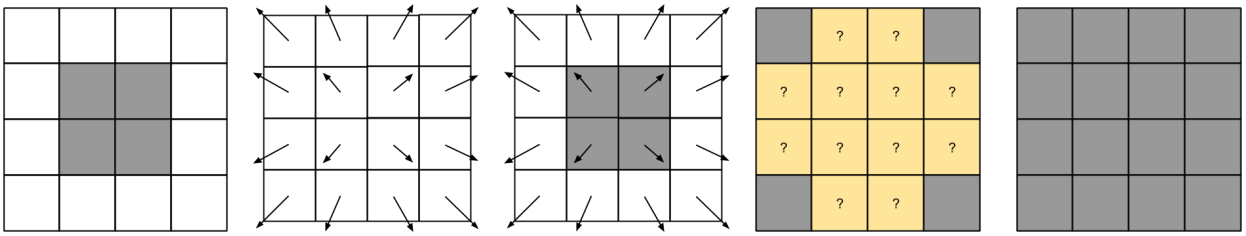
(a) Input image $\boldsymbol{I}$ **(b)** Optical flow $\boldsymbol{f}$ **(c)** Optical flow applied on input image **(d)** Interpolated image $\boldsymbol{I}_{INT(\boldsymbol{f})}$ **(e)** Ideal interpolation

Figure 6.3: Straightforward (naive) optical flow interpolation with rigid movement flow. Edge pixels in interpolated image are left unknown. Ideally they are filled as original edge pixels.

A straightforward method for interpolating $\boldsymbol{I}$ with $\boldsymbol{f}$ is to simply apply flow vector $f(p)$ on pixel p in $\boldsymbol{I}$, and place the resulting pixel in the interpolated image. An example is given in Figure 6.3. In this example, we have a small dark square in the center of the 4x4 input image, and the

optical flow is given such that all pixels move one pixel in the bottom right direction. When we apply the flow to the image, the black box is moved to the bottom right corner. However, since we don't know what should be filled in at the top and left edges, they are marked with a "?" (Figure 6.3d). This problem can potentially be solved by filling the unknown edge pixels with the original edge pixel values. The ideal interpolated image is shown in Figure 6.3e.

However, the unknown pixels can also appear in the middle of an interpolated image. In Figure 6.4, we show such an example.



(a) Input image $\boldsymbol{I}$ **(b)** Optical flow $\boldsymbol{f}$ **(c)** Optical flow applied on input image **(d)** Interpolated image $\boldsymbol{I}_{INT(\boldsymbol{f})}$ **(e)** Ideal interpolation

Figure 6.4: Straightford (naive) optical flow interpolation with expanding flow. The black box in the center is expanded by the given flow, and the pixels inside are left unknown. Ideally the object is expanded with inside filled.

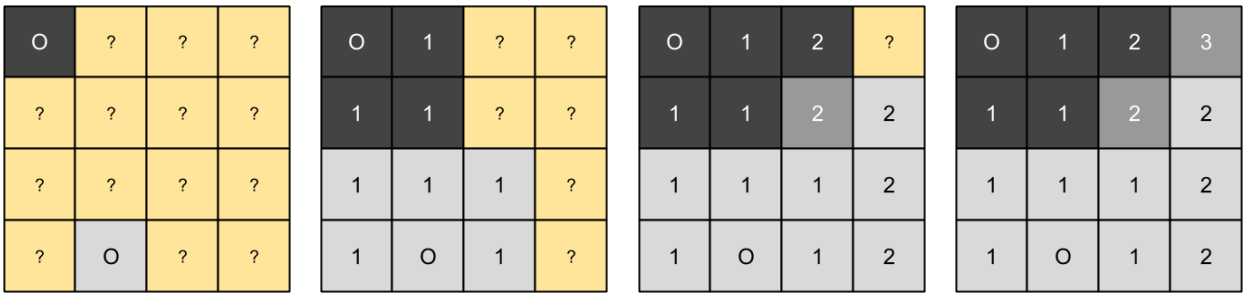
Here, the input image is the same as last example, but the flow is changed to be expanding from the center. Ideally, as an object expands, it will occupy all areas inside of it, as shown in the ideal interpolation Figure 6.4e. However, the straightforward (naive) interpolation is not able to fill in the unknown pixels (Figure 6.4d).

We can fill in the "holes" by "guessing" their values from their surrounding pixels. In our modified interpolation algorithm, we fill in the holes by interpolating the unknown pixels using their neighbors. An illustrated example is given in Figure 6.5.

Assume the interpolated image has only two known pixels, marked with "O", and one is darker than the other (Figure 6.5a). All the remaining pixels are unknown (holes).

We fill the holes that are nearest to the known pixels, and gradually to the ones that are further away. In Figure 6.5b, all holes one pixel away from known pixels are filled by their closest known pixels.

In Figure 6.5c, we keep filling holes that are two pixels away. For the hole with equal distance



(a) Interpolated image with holes

(b) Filling holes within 1 pixel

(c) Filling holes within 2 pixels

(d) Filling holes within 3 pixels

Figure 6.5: Filling in the holes of interpolated image. Intensity values of original pixels (marked by "O") are propagated to neighboring pixels with unknown values (marked by "?"), level by level. When multiple intensity values are propagated to the same pixel, average value is used.

to the dark and light pixel, we fill it with the average intensity of the pixels marked with "O". In Figure 6.5d, we fill the last hole using the same process.

Using this process, the previous interpolated images in Figure 6.3d and Figure 6.4d can be easily filled to the ideal interpolations (Figure 6.3e and Figure 6.4e).

Pseudocode for this interpolation algorithm is presented in Algorithm 6.1.

Algorithm 6.1 InterpolateImage: Image interpolation algorithm

Input: Image $\boldsymbol{I}$, optical flow $\boldsymbol{f}$
Output: Interpolated image $\boldsymbol{I}_{INT(\boldsymbol{f})}$

1: Temporary image $\boldsymbol{I}' \leftarrow$ Apply $\boldsymbol{f}$ on $\boldsymbol{I}$, mark unknown pixels with "?". If multiple pixels in $\boldsymbol{I}$ move to the same position in $\boldsymbol{I}'$, then use the maximum value.
2: $\boldsymbol{I}_{INT(\boldsymbol{f})} \leftarrow$ FillHoles($\boldsymbol{I}'$) {Algorithm 6.2}
3: **return** $\boldsymbol{I}_{INT(\boldsymbol{f})}$

Algorithm 6.2 FillHoles: Filling unknown pixels by interpolating from known pixels

Input: Image with holes $\boldsymbol{I}$
Output: Image without holes $\boldsymbol{I}^*$

1: Initialize $\boldsymbol{I}^* \leftarrow \boldsymbol{I}$
2: **for all** unknown pixel p in $\boldsymbol{I}^*$ **do**
3: $D \leftarrow$ Distance from p to nearest known pixel in $\boldsymbol{I}$
4: $\boldsymbol{I}_D \leftarrow$ All known pixels D pixels away from p, in $\boldsymbol{I}$
5: $\boldsymbol{I}^*(p) \leftarrow$ average($\boldsymbol{I}_D$)
6: **end for**
7: **return** $\boldsymbol{I}^*$

In Figure 6.6, we show a few examples of image interpolation.

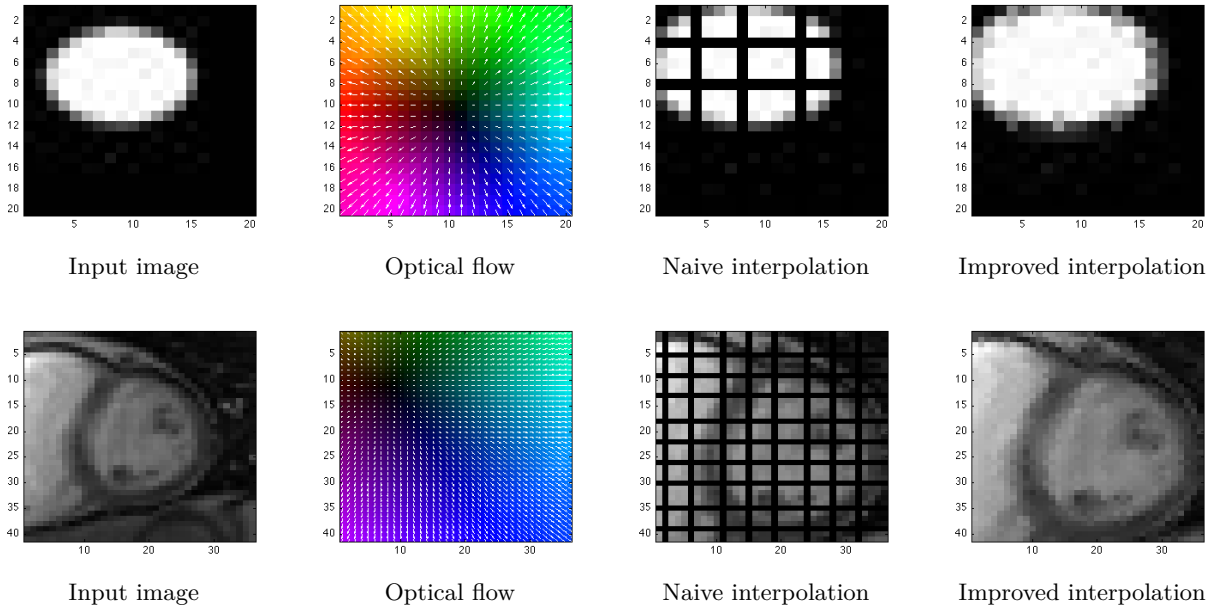


Figure 6.6: Image interpolation examples. With an expanding optical flow, naive interpolation leaves "holes" (black grids), while improved algorithm fills the holes nicely.

6.1.3 Input Dataset

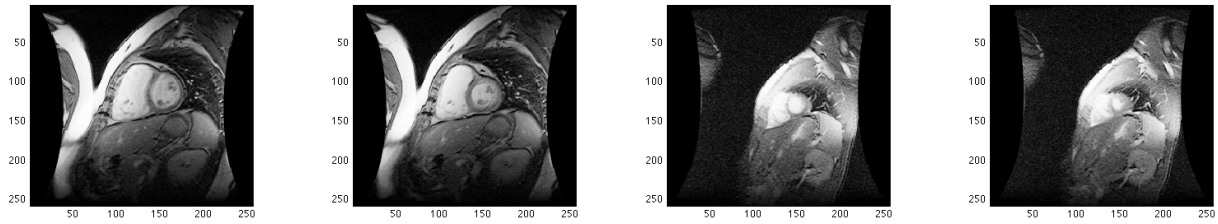
We use both synthetic images and real images for testing. The benefit of using synthetic images is the ability to control ground truth optical flow and input image, which allows fine grained testing of optical flow algorithms. Using real images allows testing algorithms in real world situations.

When using synthetic images, we first generate the first image $\boldsymbol{I}$ and the ground truth flow $\boldsymbol{f}_{GT}$. Then we generate the second image $\boldsymbol{I}'$ by interpolating $\boldsymbol{I}$ with $\boldsymbol{f}_{GT}$. The exact shapes of synthetic images will be shown in the following experimental sections. The synthetic images have a size of 20x20 pixels, unless otherwise noted.

For real images, one set of images used comes from York University's Cardiac MRI dataset[8] [3]. The dataset contains cardiac MR images acquired from 33 patients, each patient's sequence consisting of 20 frames and 8-15 slices along the long axis, for a total of 7980 images. A sample of the dataset is shown in Figure 6.7.

We also use the Middlebury dataset[9] [5]. The dataset contains both synthetic images with

[8] http://www.cse.yorku.ca/~mridataset/
[9] http://vision.middlebury.edu/flow/data/



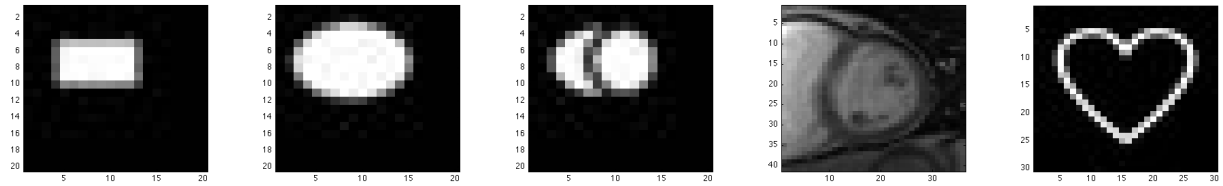
Patient 1 slice 5 frame 1 | Patient 1 slice 5 frame 5 | Patient 2 slice 10 frame 1 | Patient 2 slice 10 frame 5

Figure 6.7: York dataset sample cardiac MRI images.

known ground truth, as well as camera captured image sequences with ground truth measured by special texture-tracking device. We will use part of this dataset for comparison.

6.1.4 Impact of Object Shape

In this set of experiment, we investigate how the shape of an input object affects the resulting optical flows. The input objects are shown in Figure 6.8.



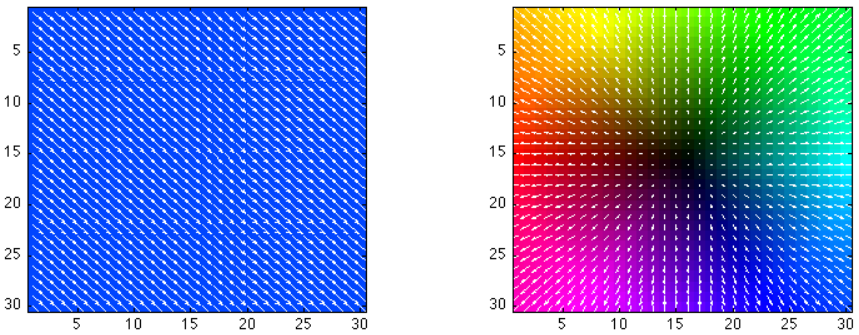
(a) Rectangle **(b)** Ellipse **(c)** Ventricles **(d)** Heart **(e)** Heart2

Figure 6.8: Input shapes used in determining impact of object shape on flow result

The input objects $\boldsymbol{I}$ include the following:

- **Rectangle and ellipse** - These are basic geometric shapes, with inside filled.
- **Ventricles** - This is a simplified image of a person's left ventricle (LV) and right ventricle (RV). The actual shape of LV & RV varies for different people, while two examples can be seen in Figure 6.7a and Figure 6.7c.
- **Heart** - This image is a crop of Figure 6.7a, focusing on the ventricles area.
- **Heart2** - This is an illustrated heart, to add some irregularity to input shapes. It is also made hollow to differentiate it from the other shapes.

We applied two different types of transformations to the input images $\boldsymbol{I}$ to generate their sequel images $\boldsymbol{I}'$:



(a) Optical flow 1 **(b)** Optical flow 2

Figure 6.9: Optical flow used in determining impact of object shape on flow result

- **Transformation 1 (Moving in the bottom right direction)**: Apply optical flow shown in Figure 6.9a. It moves all pixels towards bottom right direction for 2 pixels.
- **Transformation 2 (Expansion and dimming)**: Apply optical flow shown in Figure 6.9b, which expands the image from the center to 130% of the original size, then dim all pixels' intensity values to 80% of the original values, to mimic changing intensity situation.

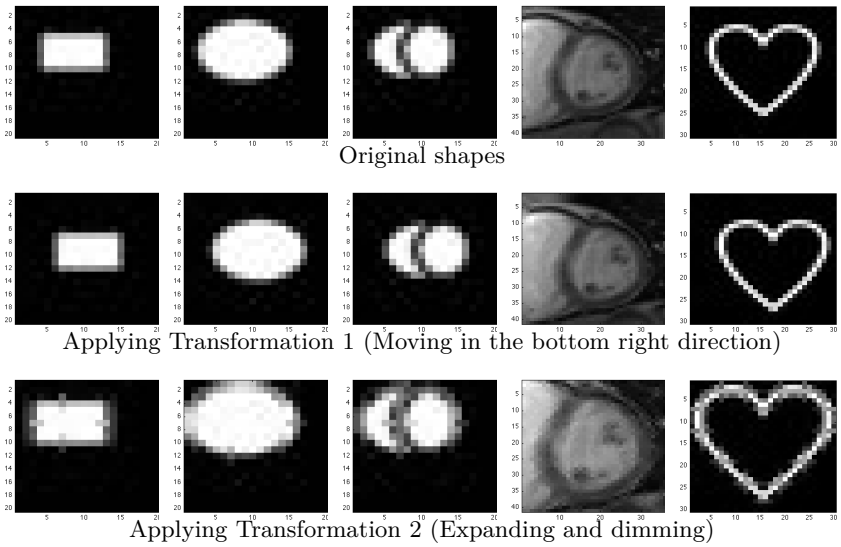


Figure 6.10: Applying transformations on five different input shapes.

Note that the optical flow applied (Figure 6.9a and Figure 6.9b) will be the ground truth flow

we will be comparing the results of optical flow algorithms with. The result of applying the two transformations on the five input shapes are shown in Figure 6.10.

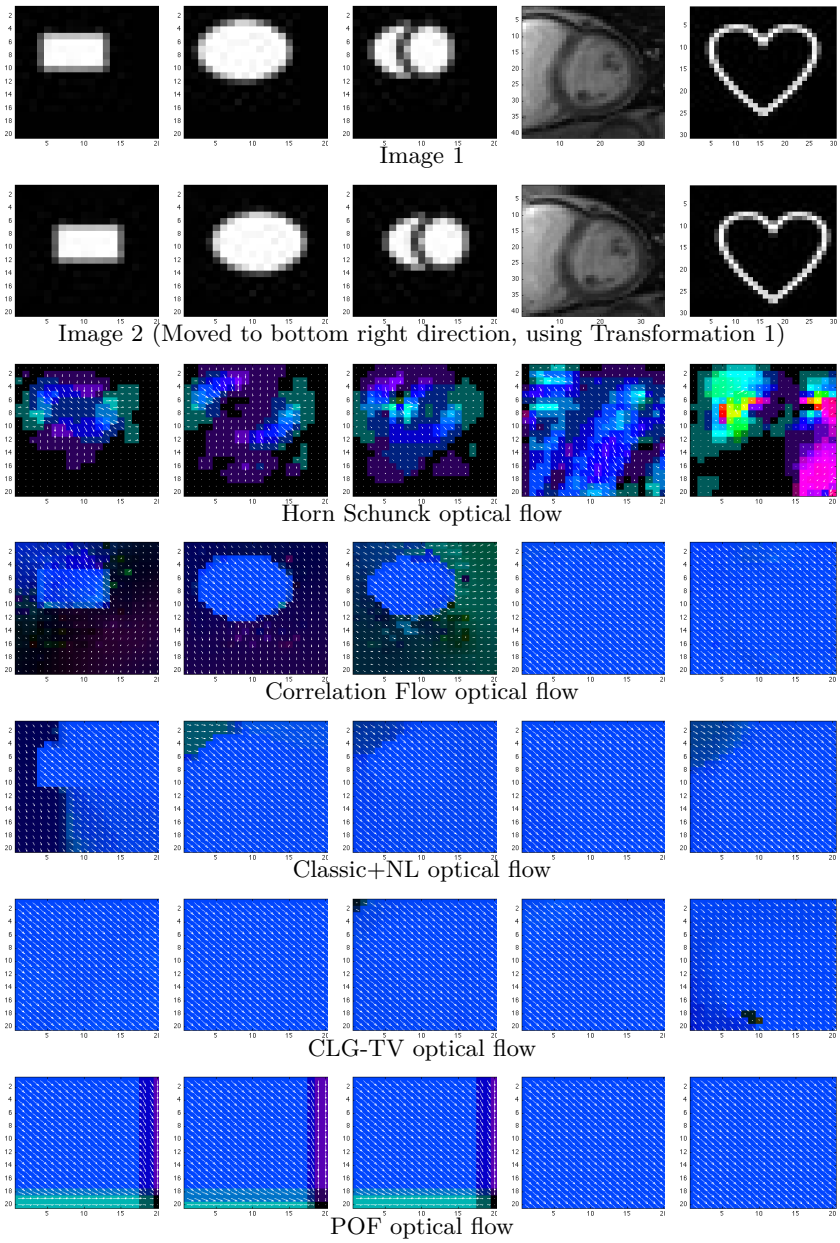


Figure 6.11: Optical flow result on changing shapes, with ground truth flow in Figure 6.9a.

We will first investigate the results of applying Transformation 1. In Figure 6.11, we show the optical flow results of using $\boldsymbol{I}$ from Figure 6.8, and $\boldsymbol{I}'$ generated using Transformation 1 (Movement, second row in Figure 6.10).

Since the ground truth flow we used is rigid movement in the bottom right direction, most

algorithms generated correct result, with most, if not all flow vectors the same as ground truth (Classic+NL, CLG-TV, POF, and part of Correlation Flow). For Correlation Flow, it detected the shape of the simpler input objects (rectangle, ellipse and ventricles). This doesn't affect error metrics since AE and EE are only computed in the object area, but this feature may be useful when using optical flow result to track objects. For the Horn Schunck algorithm, the overall result is worse than other algorithms.

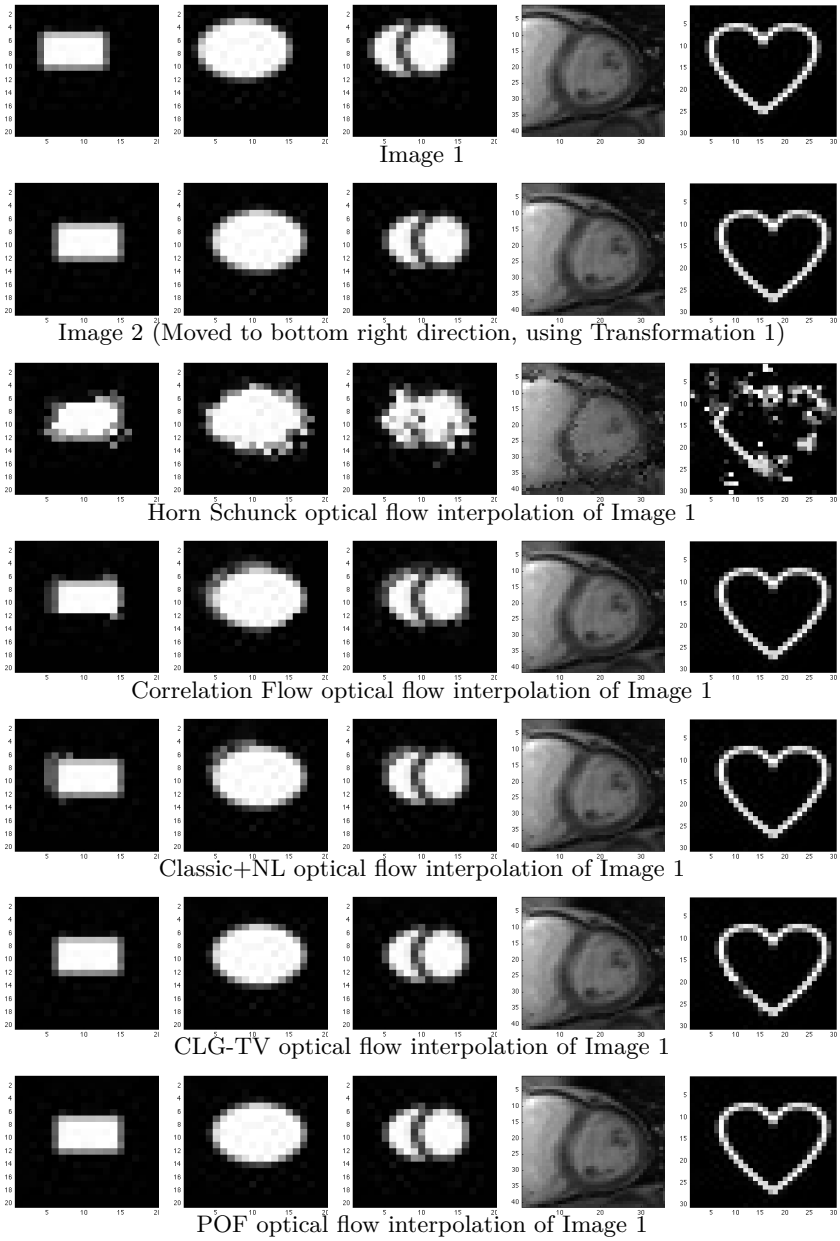


Figure 6.12: Optical flow result applied on input images. Here Image 2 is achieved by applying Transformation 1 to Image 1.

In Figure 6.12, we show the results of interpolating Image 1 with computed optical flow, and compare with ground truth Image 2. Since most algorithms generated flow results close to ground truth, the interpolated images are very close to input Image 2. For the HS method (third row in Figure 6.12), the result is not ideal.

With both optical flow results and interpolated images shown, we can go ahead and calculate the error metrics - AE, EE and IE. Results are shown in Figure 6.13. Generally, the CLG-TV and POF algorithms performed the best, with basically no errors for most shapes. However, both algorithms showed a much better performance on the simpler object shapes (rectangle, ellipse and ventricles) than the more complicated ones (hollow heart and real heart image). Interestingly, for Correlation Flow and Classic+NL algorithm, it is the opposite. They performed better on the complicated shapes than on simple geometric shapes. This result suggests that these two algorithms may be able to handle more complicated inputs.

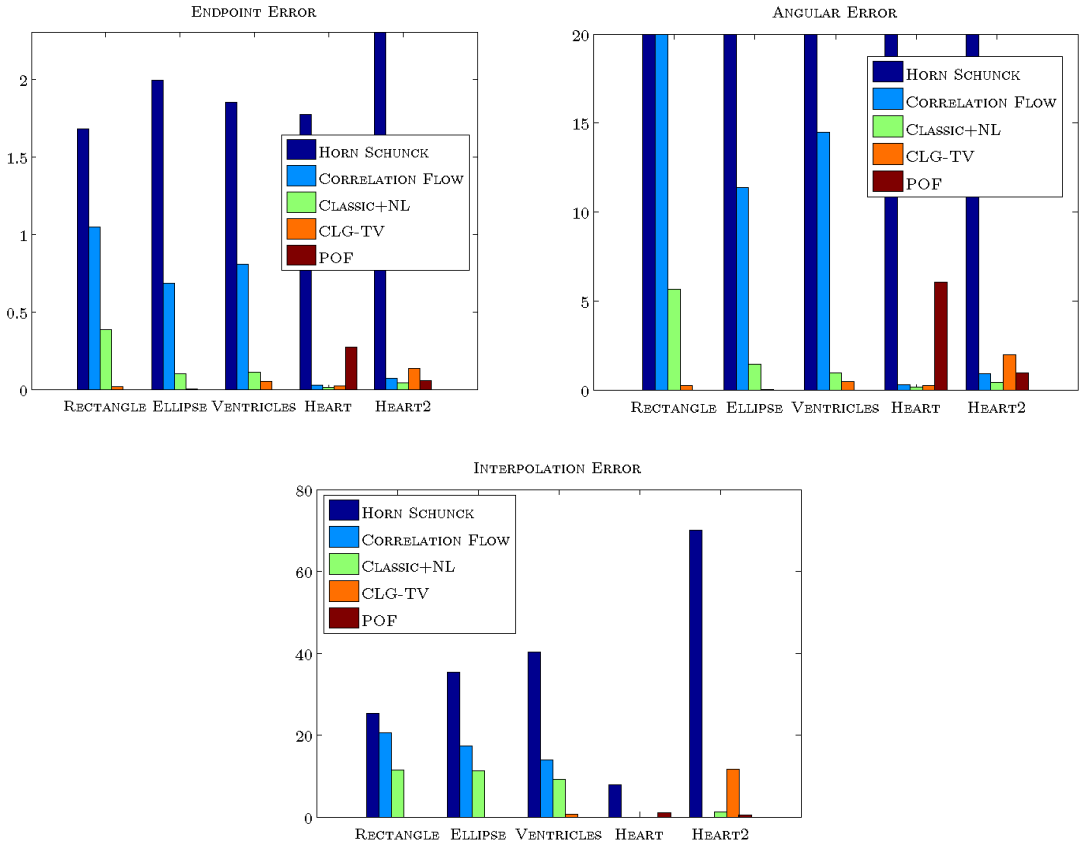


Figure 6.13: Error measures of shape test (Transformation 1). The input shapes affect flow results a lot. Generally, CLG-TV and POF performed the best.

Now we analyze the results of applying Transformation 2. Figure 6.14 shows the optical flow results. As can be seen, most of the result are quite confusing, since ideally the flow is an expansion from the center (as in ground truth in Figure 6.9b). Interestingly, many of the algorithms detected the shapes of geometric objects (rectangle, ellipse, ventricles, hollow heart) (column 1, 2, 3 and 5).

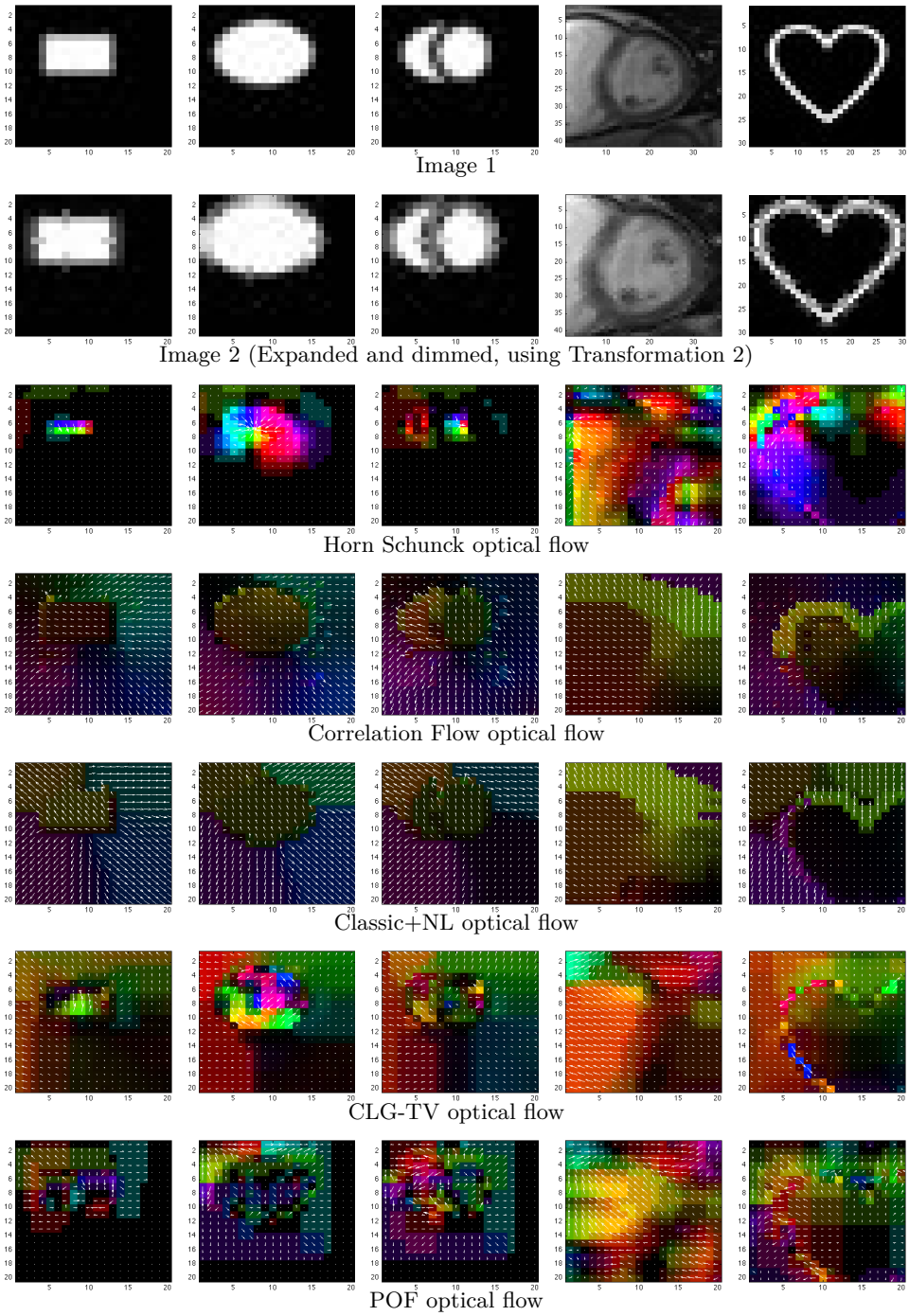


Figure 6.14: Optical flow result on changing shapes, with ground truth flow in Figure 6.9b.

In Figure 6.15, we show the results of interpolating Image 1 with the computed optical flows, and compare with ground truth Image 2. Intuitively, the interpolation result is worse than the first experiment (Transformation 1 - Movement).

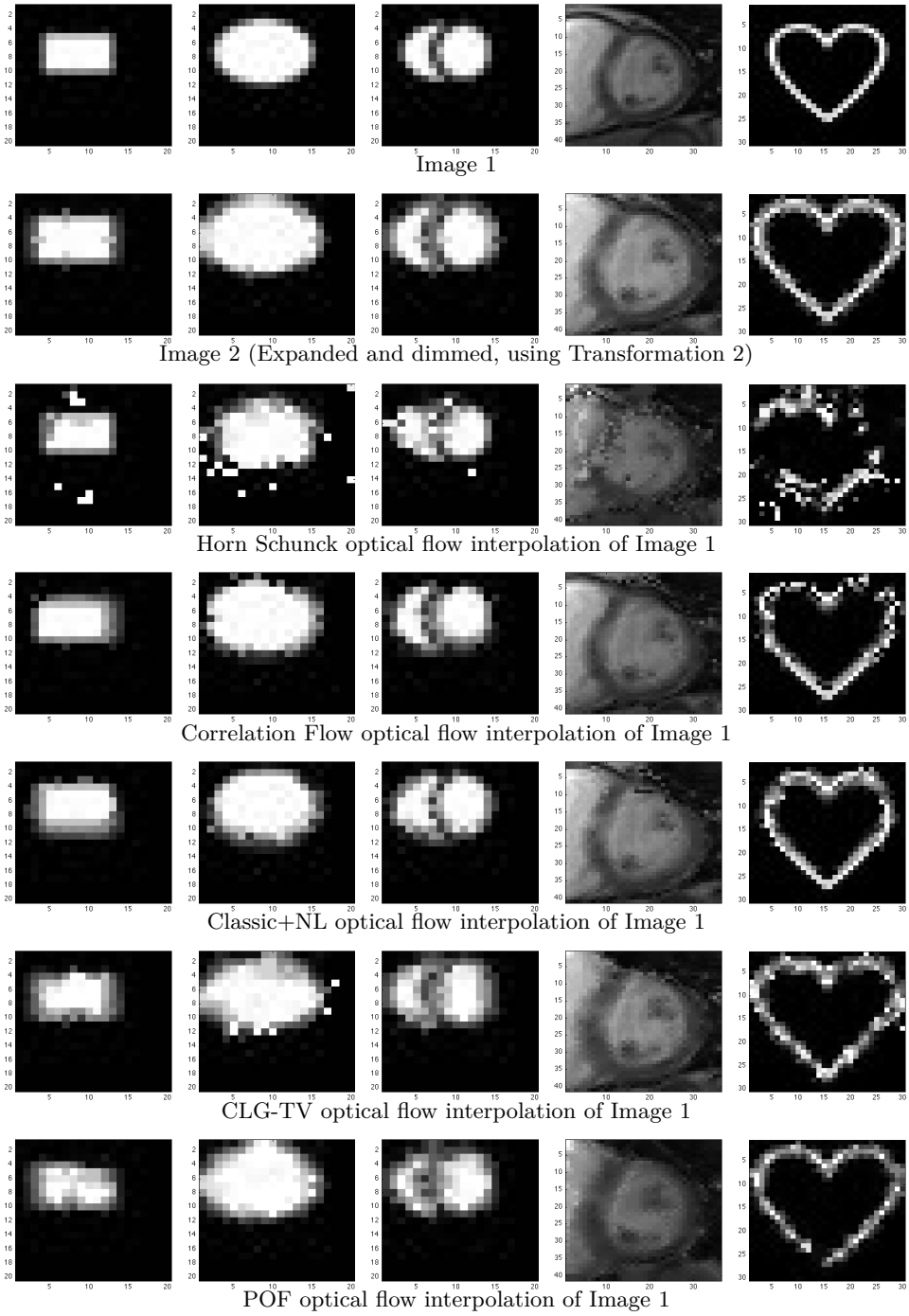
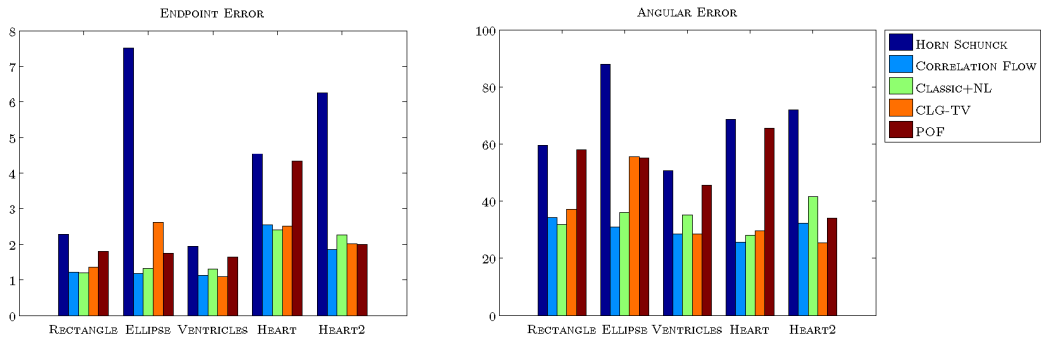


Figure 6.15: Optical flow result applied on input images. Here Image 2 is obtained by applying Transformation 2 to Image 1.

In Figure 6.16, we show the results for different metrics. Here, since the intensity is changed after the transformation, measuring interpolation error will not make sense any more. The reason is, when we apply computed optical flow to Image 1, the interpolated image will keep the original intensity of Image 1, thus it will be dramatically different from Image 2 whose intensity is dimmed during transformation. The AE and EE results across different shapes are very similar, with errors from the two heart images slightly higher for AE. Also, compared with errors in the previous movement test, the errors are much higher.



(a) Endpoint Error (EE) **(b)** Angular Error (AE)

Figure 6.16: Error measures of shape test (Transformation 2, expansion and dimming). Compared to errors in the first test (Transformation 1) which has a rigid movement, here the errors are much higher. Shape didn't affect error results that much. Also, errors are pretty consistent among different algorithms (except HS algorithm).

6.1.5 Impact of Movement

In this section, we study how movement patterns affect the results of optical flow algorithms. The following types of movements are used.

- **Single object moving in varying distances**: We will use two types of objects - rectangle and ventricles. They will each move towards bottom right direction for 1 pixel to 5 pixels.
- **Two objects moving toward each other**
- **Two objects moving away from each other**

The input Image 1, Image 2 and corresponding ground truth flow are shown in Figure 6.17.

The optical flow result on the single object moving test is shown in Figure 6.18 and Figure 6.19.

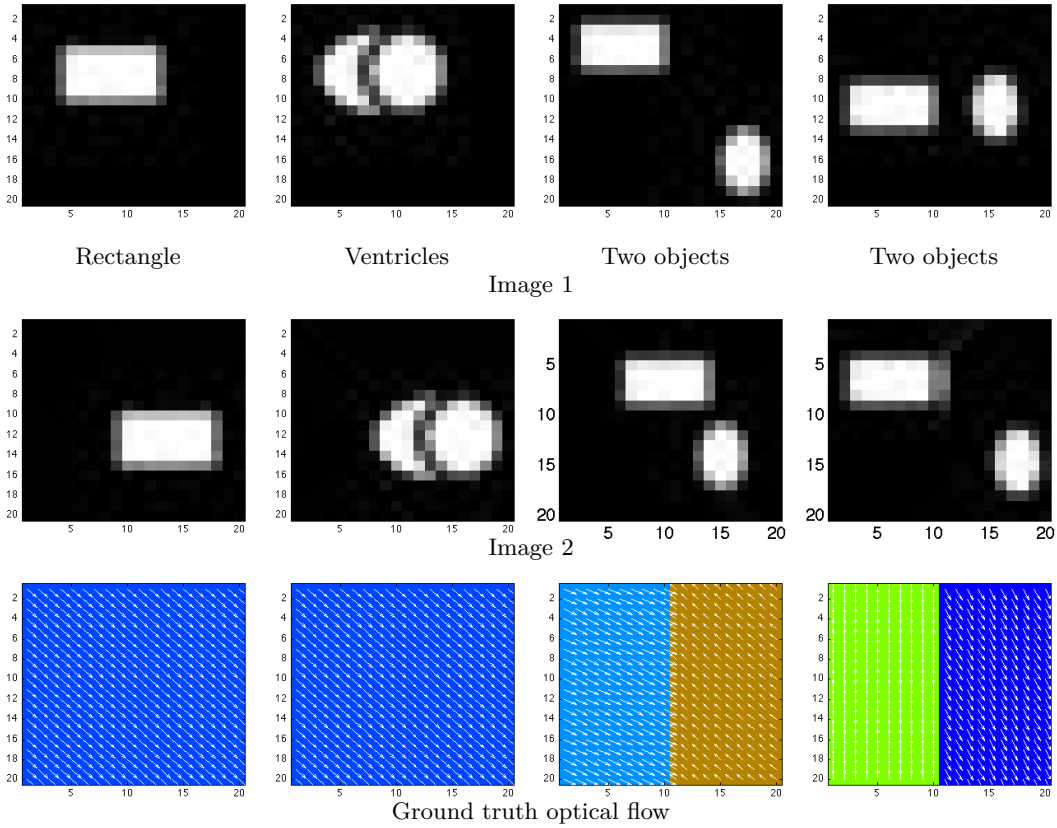


Figure 6.17: Input to the movement experiments. The left two columns (with rectangle and ellipse) are inputs to movement distance test. We only showed the result of moving Image 1 by 5 pixels. The right two columns are two objects moving closer to and away from each other.

For the rectangle movement (Figure 6.18), similar to results in the shape test (Figure 6.11), both CLG-TV and POF algorithm generated consistent flows, for movement distances 1 pixel to 4 pixels. However when distance increased to 5 pixel, both of them started generating non-ideal results. For CF and C+NL, as distance increases, both of them showed more explicit contours of the objects, though the flow of C+NL on 5 pixel distance is clearly wrong. The HS algorithm again generated confusing result.

When we changed the shape to ventricles (Figure 6.19), the results are generally worse than for the rectangle. Both CLG-TV and POF generated worse result from shorter distances (4 pixels), and for CF and C+NL, the results are absolutely incorrect, starting from 4 pixels (CF) and 3 pixels (C+NL) respectively.

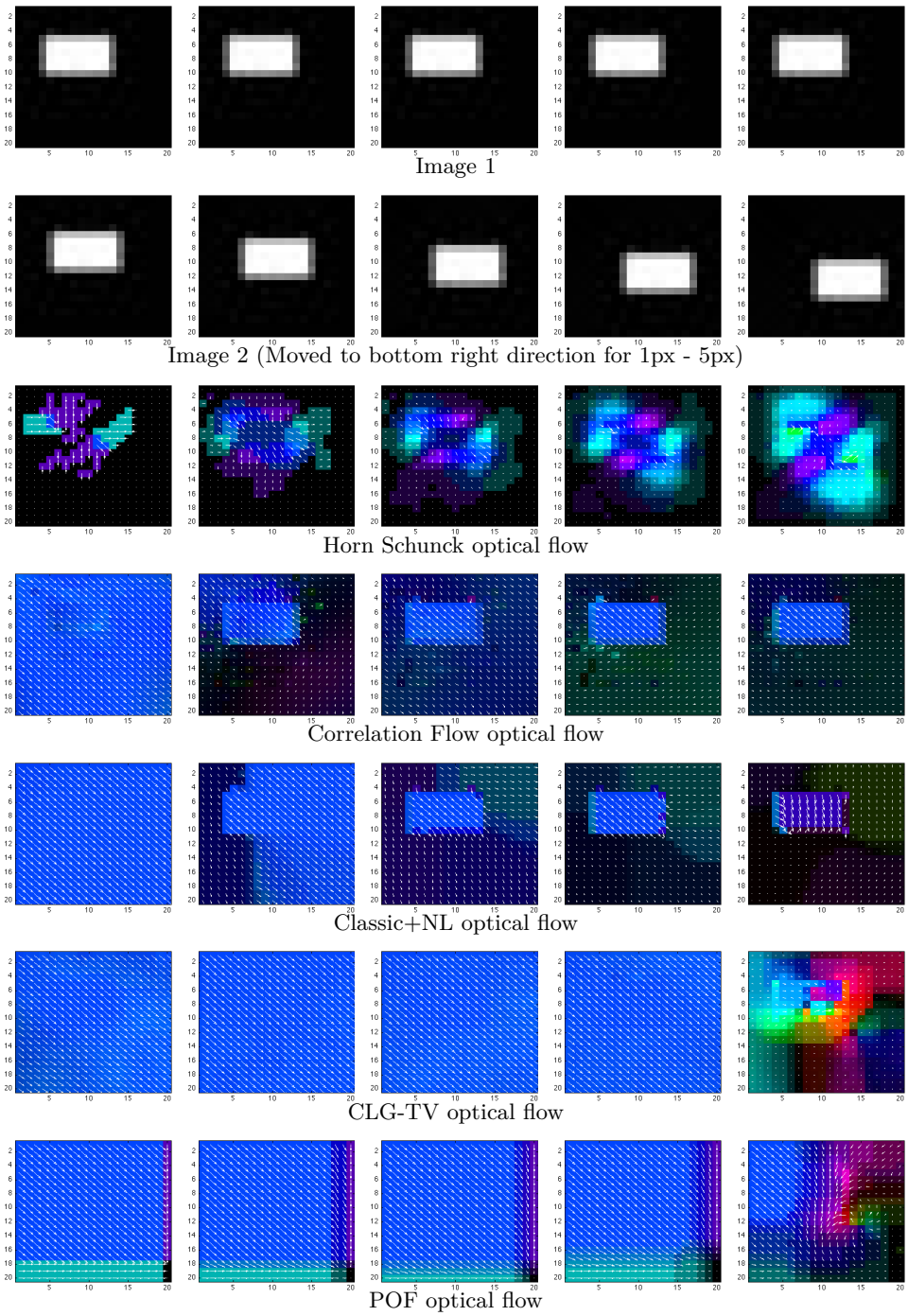


Figure 6.18: Optical flow result for the movement test (Single object, rectangle). As the movement distance increases, results have decreasing quality. Interestingly, CF and C+NL flows showed the contours of input objects, though C+NL's flow result is completely incorrect when distance is 5 pixels.

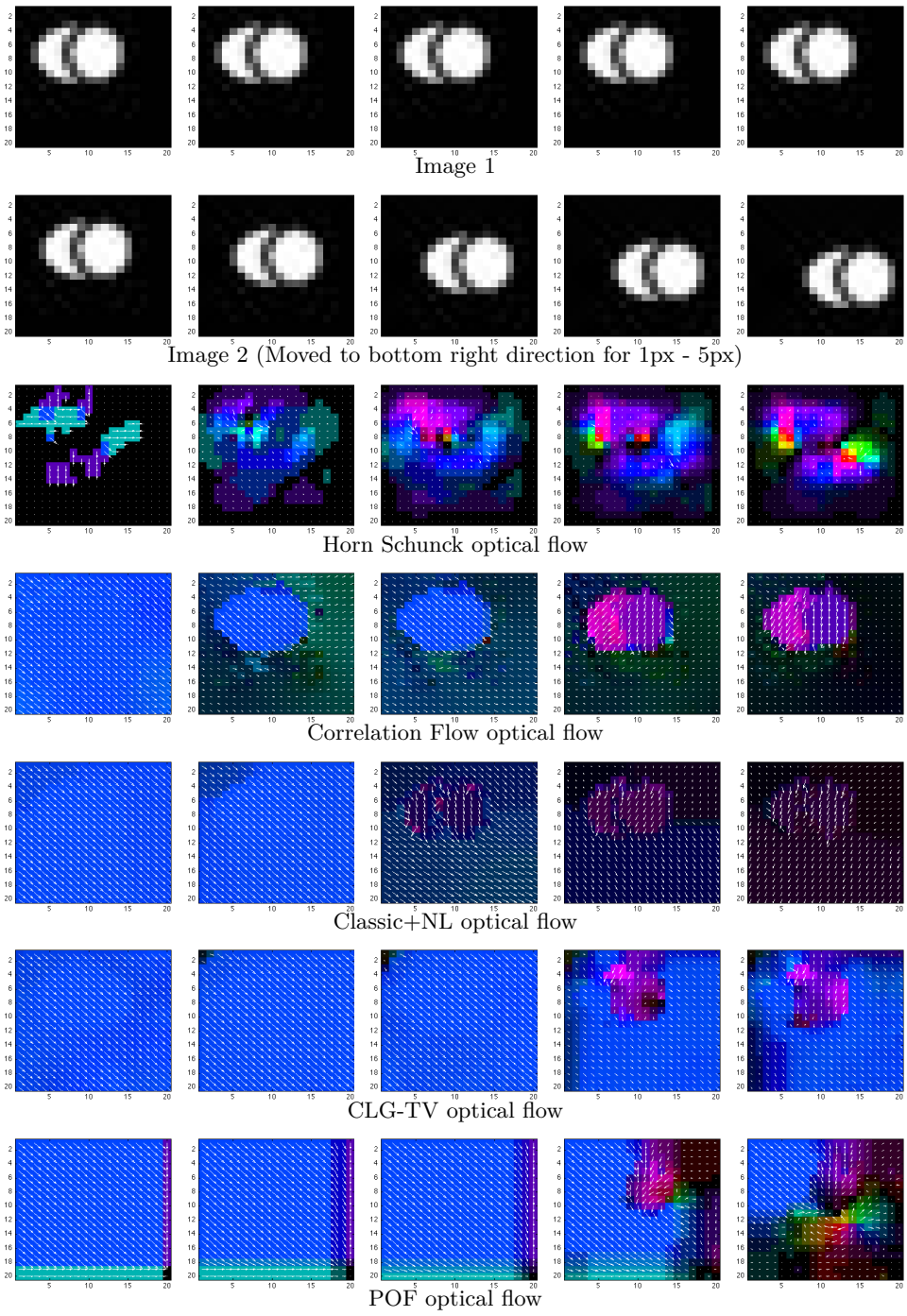


Figure 6.19: Optical flow result for the movement test (Single object, ventricles). As movement distance increases, results have decreasing quality. Again, CF and C+NL flows showed the contours of input objects, but this time both CF and C+NL's flow results are wrong when distance is large.

In Figure 6.20 and 6.21, we show the interpolated images using optical flow results.

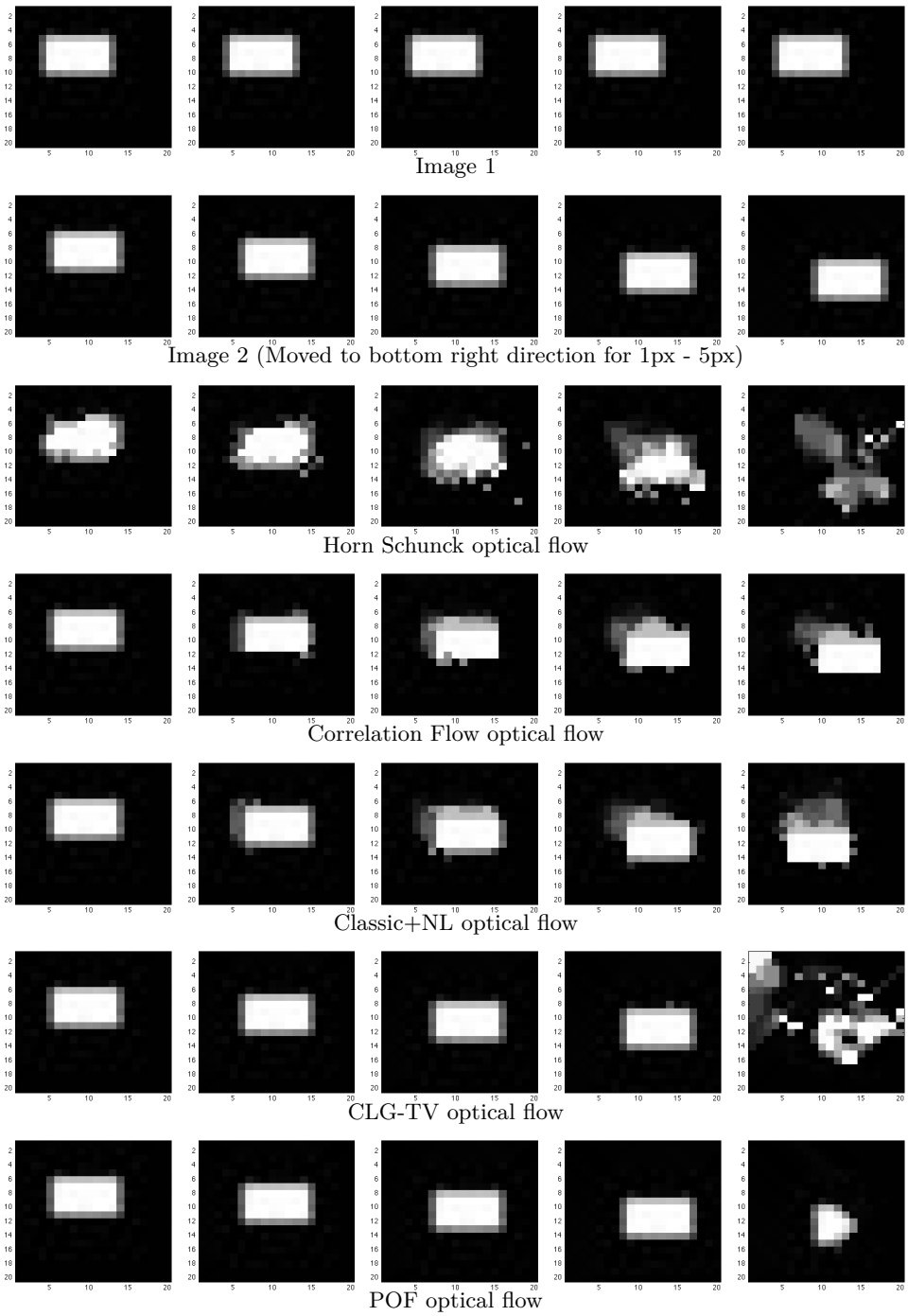


Figure 6.20: Interpolated images using optical flow result of movement test (Single object, rectangle). As movement distance increases, results have decreasing quality.

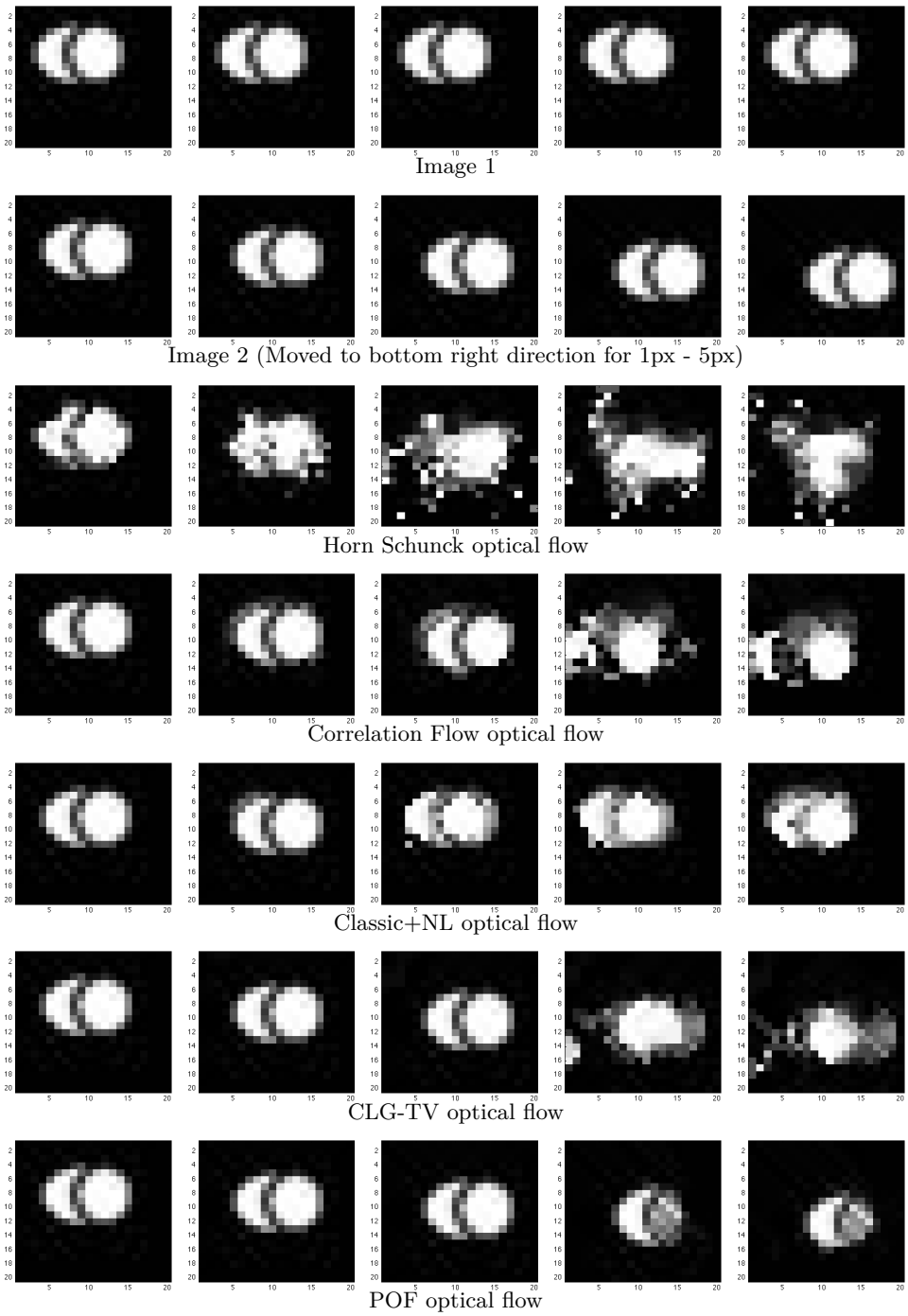


Figure 6.21: Interpolated images using optical flow result of movement test (Single object, ventricles). As movement distance increases, results have decreasing quality.

The error metrics (AE, EE, IE) of the single object moving test are shown in Figure 6.22. Generally, errors increase when moving distance increases. CF algorithm has the best performance when the input is a rectangle, across all three error metrics, but degraded a lot when the shape is

changed to ventricles. CLG-TV and POF had very similar errors, and both are less susceptible to changes of shapes.

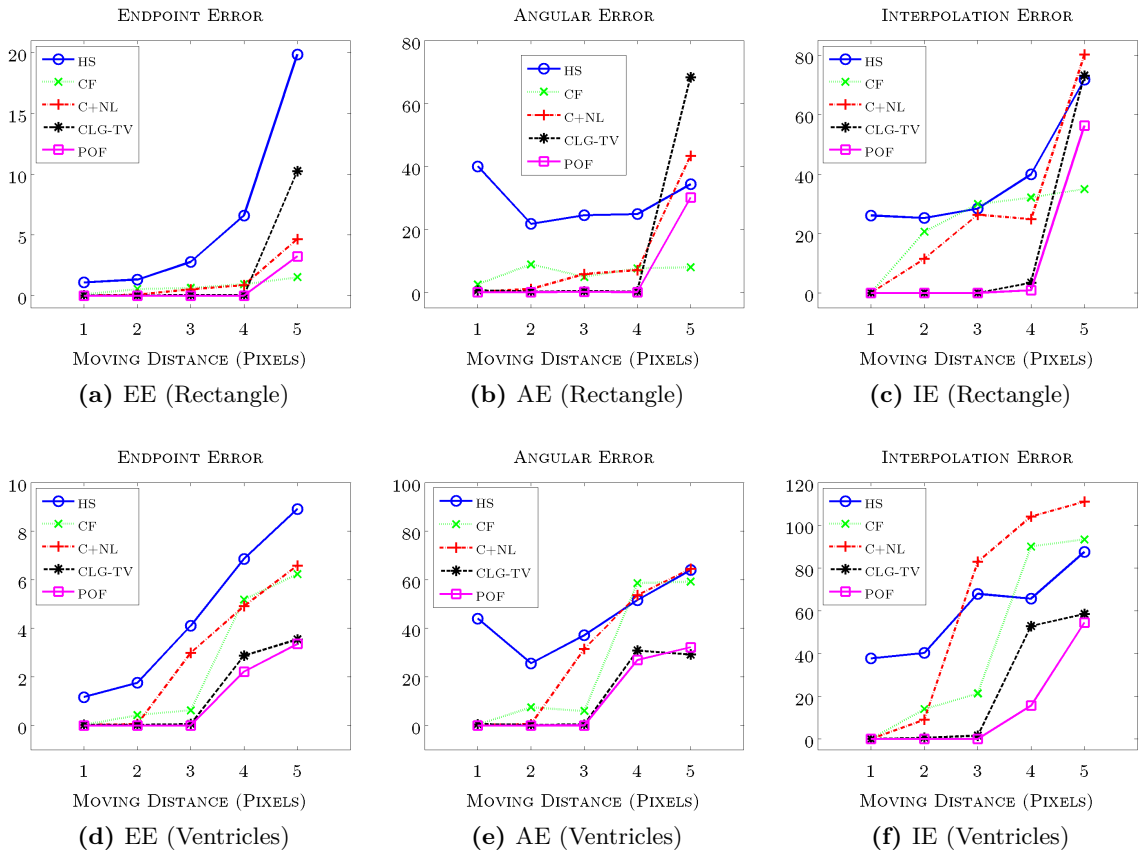
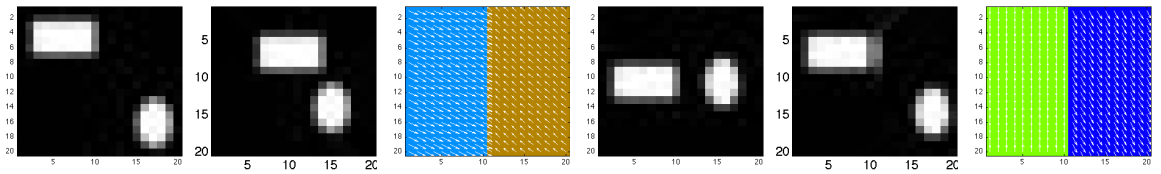


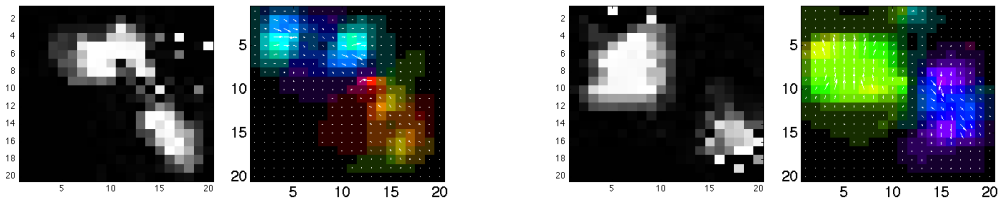
Figure 6.22: Error measures of movement test (Single object). As distance increases, errors increase. Generally CLG-TV and POF performed best when input shape is ventricles, while CF is better when the shape is rectangle.

We now go on to see how movement of two objects affects optical flow algorithms. In Figure 6.23, we show both the optical flow result and interpolated image for the two objects movement test. Image 1 is previously shown in Figure 6.17. For Image 2, the first one is the result of two objects moving closer (the ground truth flow on left and right side points toward each other), and the second one is the result of two objects moving away (ground truth flow pointing in opposite direction). CF and C+NL again generated flows that showed boundaries around the input objects, though for C+NL the rectangle's flow is incorrect when the objects are moving away. For CLG-TV and POF, both generated flow values correctly on top of the objects. The difference is in how much neighboring areas are also included in the flow.

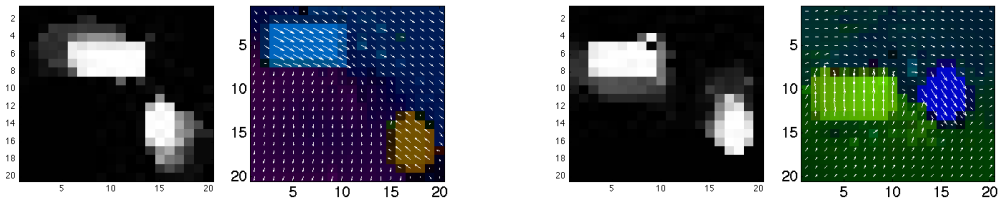


Moved closer Moved away

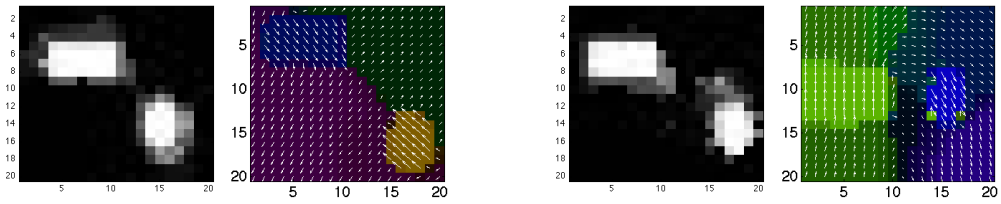
Image 1, Image 2 and ground truth optical flow



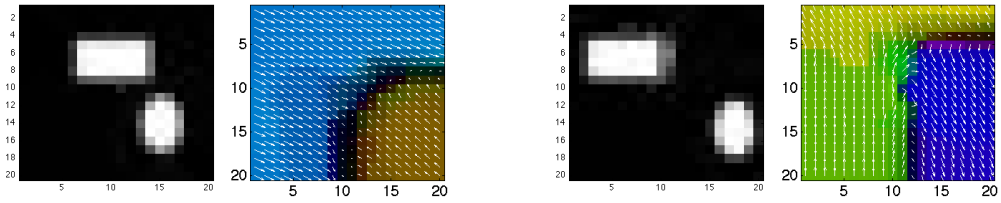
Horn Schunck interpolation and flow



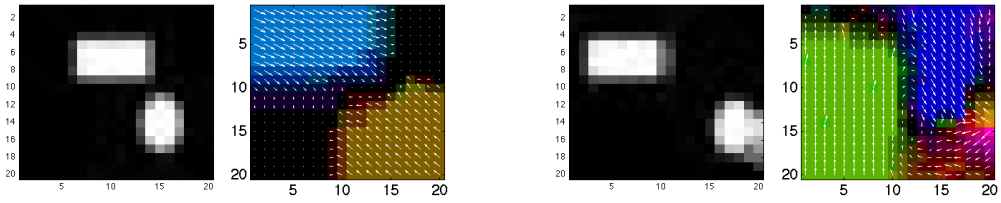
Correlation Flow interpolation and flow



Classic+NL interpolation and flow



CLG-TV interpolation and flow



POF interpolation and flow

Figure 6.23: Interpolated images using optical flow result of movement test, two objects moving closer and away. From the second row to bottom, the 1st and 3rd columns show interpolated images using optical flow results shown in 2nd and 4th columns.

When objects moved away (right two columns), again CF and C+NL's flow results showed object boundaries.

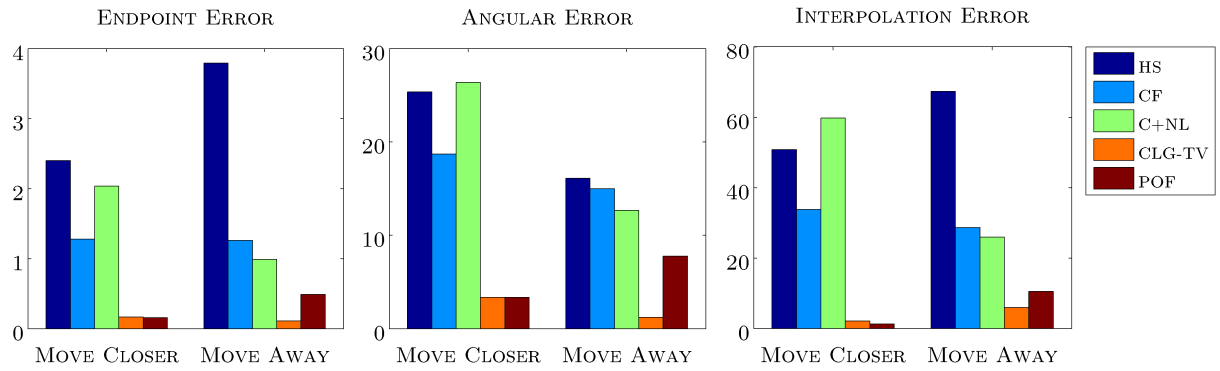


Figure 6.24: Error measures of movement test (Two objects). CLG-TV and POF are the winners here, showing their adaptabiity to non-rigid movements.

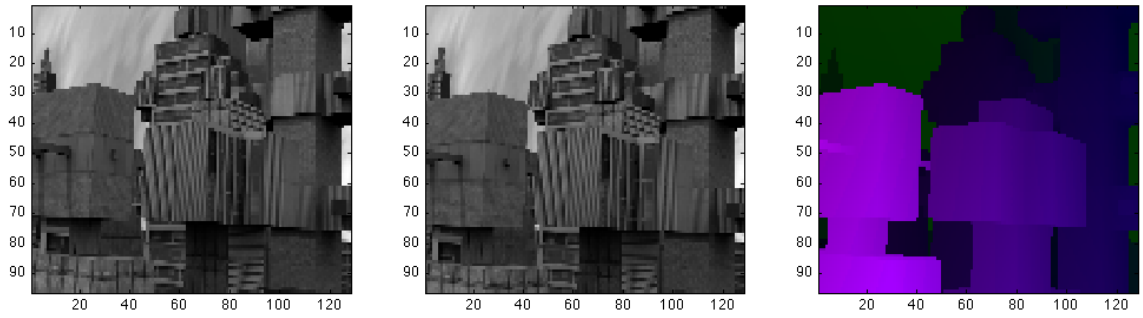
The error metrics of the two objects moving test is shown in Figure 6.24. For CF, C+NL and CLG-TV, they all performed better when objects move away, while POF performed better when objects move closer. Also, CLG-TV and POF (the two algorithms that generated a larger area of flows) performed a lot better than CF and C+NL. It shows CLG-TV and POF has a better adaptability to non-rigid movements (movements with different directions).

6.1.6 Using Middlebury Images

From the Middlebury dataset,[10] we used **Urban3** image input, which can be seen in Figure 6.25.

The flow and interpolation results are shown in Figure 6.26, and the errors are shown in Table 6.2. From the flow result, we can tell that CLG-TV and POF have more fragmented flow fields, while CF and C+NL are more smooth. The EE and AE for CF and C+NL are better than for CLG-TV and POF, while CLG-TV and POF have better IE result. The reason is that CLG-TV and POF algorithms allow more flexibility of flows in local small image patches, while CF and C+NL enforce a tighter global control. Thus, CLG-TV and POF will generate flows that give better interpolation result (since small image patches can be more flexibily flown to desired endpoint), while this flexibility often hurts the global EE and AE which requires flow to be more consistent,

[10] http://vision.middlebury.edu/flow/data/



(a) Image 1 **(b)** Image 2 **(c)** Ground truth flow

Figure 6.25: Input of Middlebury data (Urban3)

like ground truth flow.

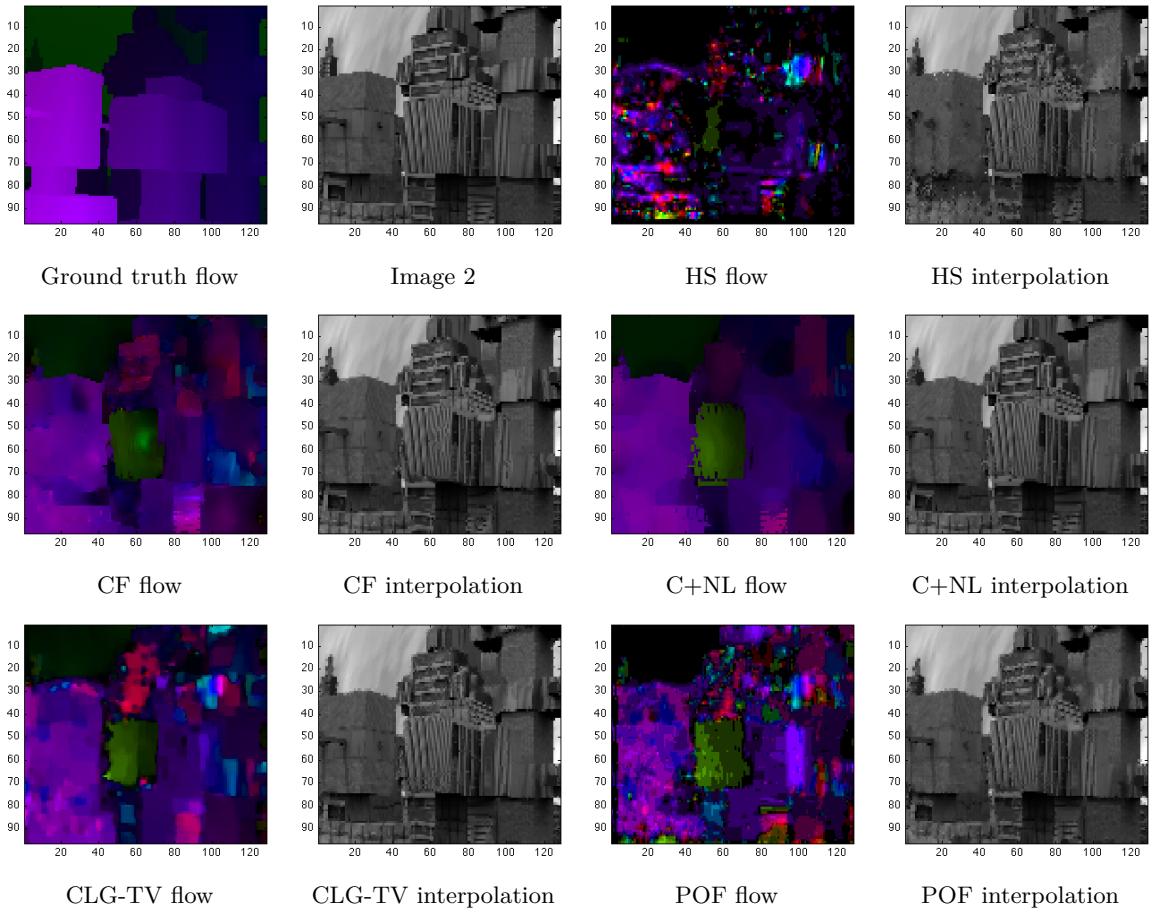


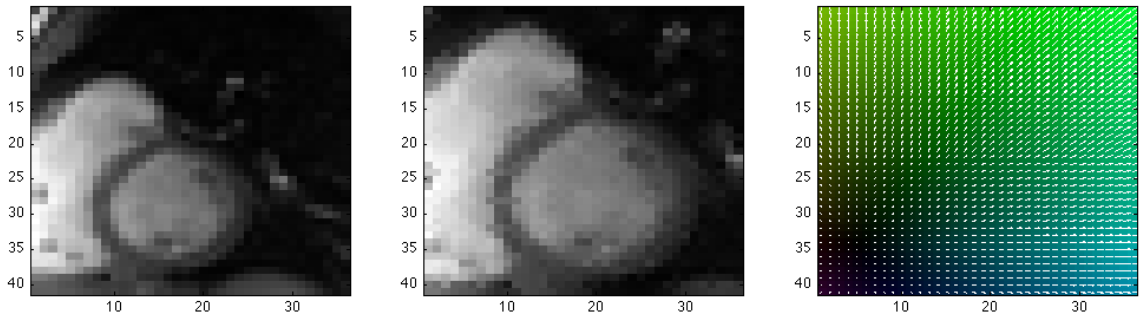
Figure 6.26: Optical flow result on Urban3. POF result is more fragmented compared with CF, C+NL and CLG-TV method.

	EE	AE	IE
HS	1.4	40.0	15.8
CF	0.8	24.0	14.1
C+NL	**0.6**	**20.4**	13.6
CLG-TV	0.9	25.9	**9.7**
POF	1.2	37.2	10.2

Table 6.2: Errors of optical flow results on Urban3 input

6.1.7 Using Cardiac Images

Here we compare OF algorithms on cardiac images. As mentioned in Section 6.1.3, we will be using York University's Cardiac MRI dataset[11] [3]. For input Image 1, we used patient 3's images, with slice number 5 at time frame 1. Since we don't have ground truth flow between these MRI images, we opted to generate our own ground truth flow, and apply it on Image 1 to generate the input Image 2. The input images can be seen in Figure 6.27. The ground truth flow shown in Figure 6.27c is an expanding flow, centered on the bottom left corner.



(a) Image 1 **(b)** Image 2 **(c)** Ground truth flow

Figure 6.27: Input of cardiac image

The resulting flows and interpolated images with the flows can be seen in Figure 6.28, and resulting errors can be seen in Table 6.3.

[11] http://www.cse.yorku.ca/∼mridataset/

	EE	AE	IE
HS	3.4	40.9	14.8
Correlation Flow	2.8	31.2	14.4
Classic+NL	2.1	17.5	12.6
CLG-TV	**1.2**	**10.0**	4.3
POF	2.0	22.3	**3.6**

Table 6.3: Errors of optical flow results on cardiac MRI input

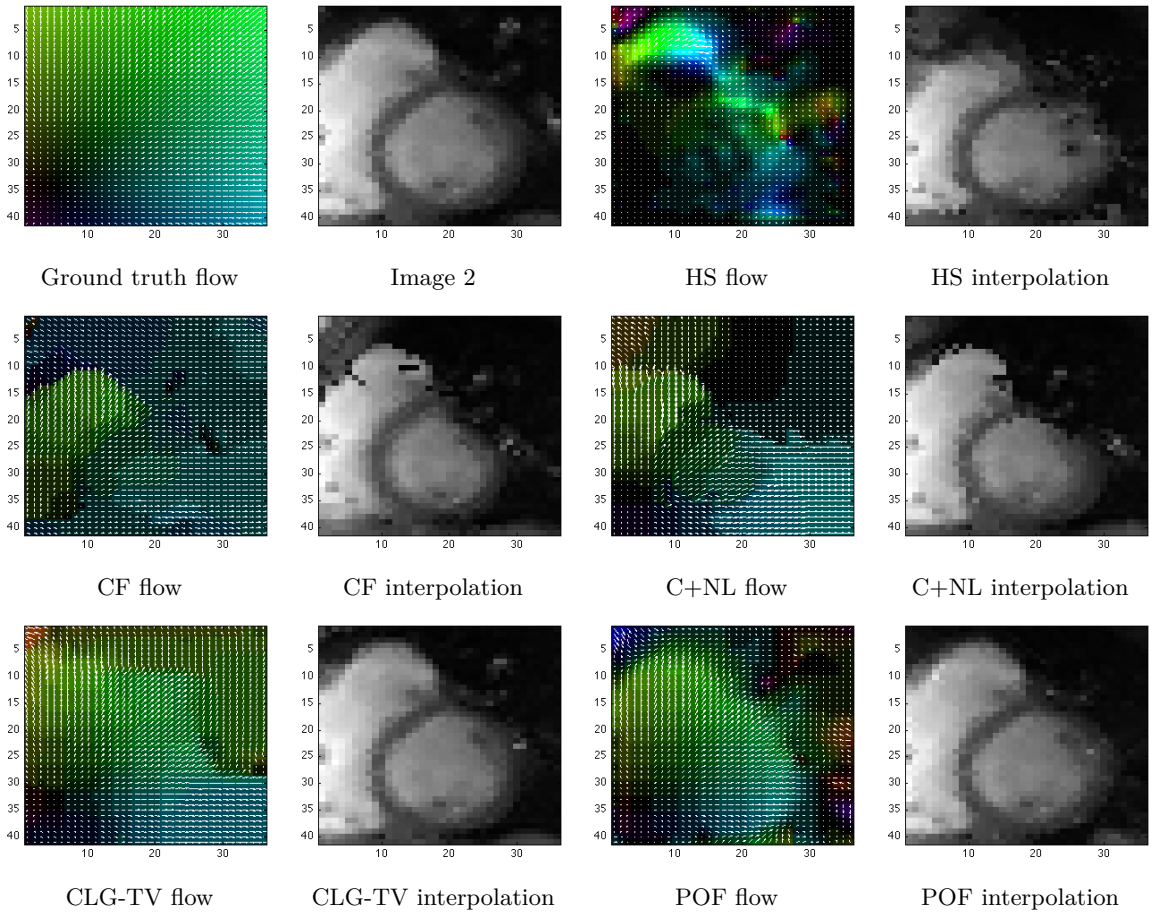


Figure 6.28: Optical flow result on cardiac MRI images.

Both CLG-TV and POF generated optical flows close to ground truth flow, while POF's result had lower interpolation error. This result shows that POF is able to deal with non-rigid motions nicely. It also shows POF's ability to lower interpolation error compared to the other algorithms. A low interpolation error means that pixels in Image 1 are moved to positions in Image 2 with similar intensity values. In POF, each pixel's intensity error is lowered individually via intensity energy

(Equation 3.7). It is more flexible than other algorithms in which pixels are moved in groups, and that is the reason for the superior IE performance of POF algorithm.

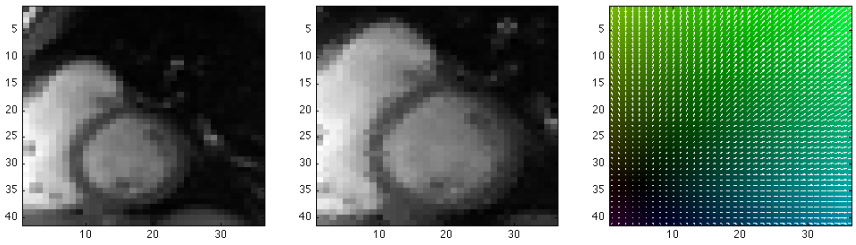
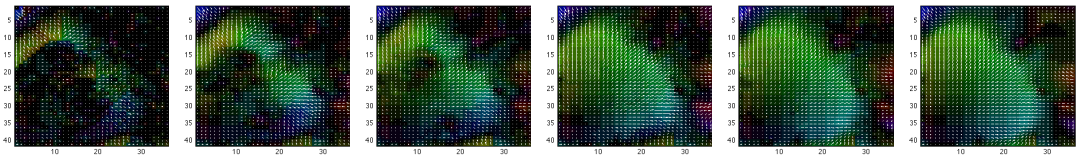
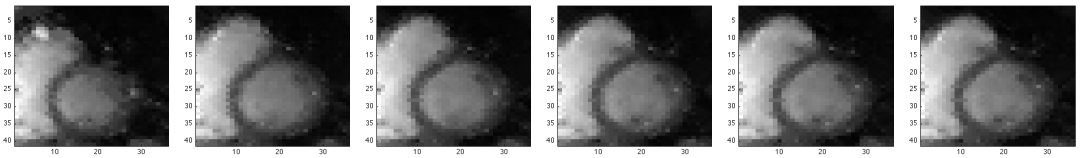


Image 1 Image 2 Ground truth flow



$t = 0.3$ $t = 2.5$ $t = 6.7$ $t = 21.3$ $t = 85.2$ $t = 216$ (Final)

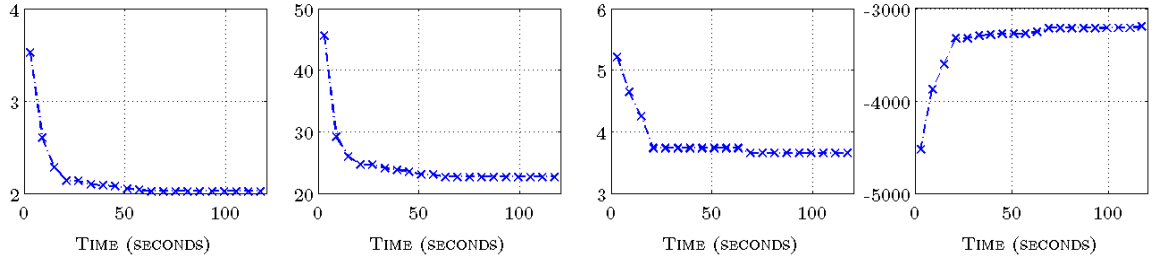
Optical flow result from POF at different time steps, from $t = 0.3$ sec to $t = 216$ sec.



$t = 0.3$ $t = 2.5$ $t = 6.7$ $t = 21.3$ $t = 85.2$ $t = 216$ (Final)

Interpolated image using OF result from POF at different time, from $t = 0.3$ sec to $t = 216$ sec.

Figure 6.29: Evolvement of optical flow result on cardiac MRI images at different time.



(a) Median Best EE **(b)** Median Best AE **(c)** Median Best IE **(d)** Median Best Log-Likelihood

Figure 6.30: Median best errors and log likelihoods of POF algorithm on cardiac image input. OF samples' performance increase quickly at the beginning, then the improvement speed slows down.

In the end, we show the evolution of POF's optical flow samples over time, as computed by the

POF Gibbs sampler, in Figure 6.29. The evolution of various error measures and log likelihoods are shown in Figure 6.30. We can see that initially the log-likelihood of the flow samples increases quickly, then it slows down. Similar pattern can be found for the error measures. This kind of behaviour is very typical for the POF algorithm. It takes very short time (compared to the entire sampling time) to find a reasonably good flow assignment, then finding a better flow assignment becomes harder and takes longer time.

6.1.8 Conclusion

In this section of experiments, we tested the performance of various optical flow algorithms, on different types of testing inputs, including varying shapes, changing moving distance on single object, two objects moving closer and away, and real images.

Notably, POF algorithm is often able to generate flows that have a better IE error result than other algorithms, which shows the strong adaptability of the algorithm to fine-grained flow variations. Also, compared to other OF algorithms, POF is less susceptible to changing input shapes, nor non-rigid movement patterns. However, when movement distance of an object increases, POF, together with all other algorithms, has decreasing flow result quality. Also, when the input image becomes larger, the result of the POF algorithm starts to become fragmented, which may not be desired in real-world images. The reason for such behaviour is that in the definition of the POF algorithm, we used neighbor energy (Equation 3.9) that only involves adjacent nodes to control smoothness of generated flow. While in small images this is enough, in larger images the conformity among neighbors may not propagate far enough to cover a larger section of the image.

There are several directions to improve the POF algorithm.

- **Use higher level factors.** Currently, the only restriction in POF on having conformant flow among adjacent pixels is the neighbor factor defined on adjacent flow values. We can extend the neighborhood to enforce a higher level of conformity.
- **Use pyramid method, *i.e.*, incremental multi-resolution technique.** This technique is a common method used in optical flow algorithms (see [8], [10] and the Correlation Flow [13] and CLG-TV [12] algorithms), to determine flow values in a coarse to fine process. First,

images are scaled down to enable a fast and coarse optical flow calculation, then this result is used as guidance on higher-resolution image optical flow calculations. Using this method, conformity can be enforced among larger image patches, since the same flow value (generated from a coarse image) is used as guidance on many pixels on fine grained image.

- **Speed up POF.** Using pyramid method can potentially speed up POF. Also, since most time of POF algorithm is spent on Gibbs sampling, any method that speeds up Gibbs sampling will be able to speed up POF. We will be discussing these methods in the following experimental sections.

6.2 Measuring Gibbs Sampling Convergence

We define a few metrics to measure the speed of convergence for Gibbs sampling. Since Gibbs sampling process is probabilistic, we wish to reduce the variation in our results. Thus, for any Gibbs sampling process with the same input, we run it L times (typically $L = 30$). Denote $\boldsymbol{S}$ as all samples collected from all L runs for one Gibbs sampling experiment, and $\boldsymbol{S}_l$ as the samples collected from the l-th run, then $\boldsymbol{S} \equiv \{\boldsymbol{S}_l\}, l = 1, \cdots, L$. Denote the number of samples in run $\boldsymbol{S}_l$ as R, then, $\boldsymbol{S}_l = \{S_l^r\}, r = 1, \cdots, R$. The entire set of samples we collected from one experiment can be denoted as

$$\boldsymbol{S} = \{\boldsymbol{S}_l\}_{l=1}^{L} = \left\{\{S_l^r\}_{r=1}^{R}\right\}_{l=1}^{L} \tag{6.5}$$

For each sample S_l^r, we record its log likelihood $\log P(S_l^r)$ and total elapsed time since algorithm started $t(S_l^r)$. With these data we can measure **Average Best Log-Likelihood**, **Median Best Log-Likelihood**, **Convergence Time Distribution** and **Cumulative Convergence Time Distribution**, defined as follows:

- **Average Best Log-Likelihood (ABL)** and **Median Best Log-Likelihood (MBL)**: Average / median best log-likelihood at time t for an experiment measures the average / median

of highest log-likelihoods from all sample chains, up to time t:

$$\mathrm{ABL}(t) = \mathrm{mean}\left[\max_{r=1,\cdots,r^*}\{\log P(S_l^r)\}\right], \tag{6.6}$$

$$\mathrm{MBL}(t) = \mathrm{median}\left[\max_{r=1,\cdots,r^*}\{\log P(S_l^r)\}\right], \tag{6.7}$$

where

$$r^* = \underset{r}{\mathrm{argmax}}\,\{t(S_l^r) < t\} \tag{6.8}$$

ABL(t) and MBL(t) provides an estimate of how well an average sample chain will perform at a given time. Also, we can plot ABL(t) / MBL(t) with t ranging from $[0, T]$ where T is the total sampling time.

- **Convergence Time Distribution (CTD)** and **Cumulative Convergence Time Distribution (CCTD)**: Convergence time distribution is the probability distribution of the time of convergence of a sampling chain. Denote T_{conv} as the time of convergence of one sampling chain $\boldsymbol{S}_l$, since T_{conv} is a random variable, it will follow a certain probability distribution $P_{CTD}(T_{conv})$. We can approximate this distribution by producing many sampling chains and recording their convergence times. Similar approach of measuring random processes can be found in [34, 35].

  For convergence time, we will simply use the first time for a sample chain to reach a certain log likelihood threshold $\log P_{conv}$, *i.e.*, $T_{conv} = \min\{t(S_l^r)\}, \forall \log P(S_l^r) \geq \log P_{conv}$. With CTD, we can have an estimate of how likely a sample chain can converge at time t, and use it to compare convergence speed of different sampling chains.

  Cumulative convergence time distribution is simply the cumulative form of CTD.

6.3 Doing Clever Initialization

6.3.1 Preparation

As discussed in Section 4.2, we will experiment with using results from different OF algorithms as input to POF's Gibbs sampling. We wish to use a $\boldsymbol{f}_0$ that is close to $\boldsymbol{f}^* = \mathrm{argmax}_{\boldsymbol{f}}\, P(\boldsymbol{f})$ as

a starting point, so that the sample chain can quickly find an optimal or approximately optimal result. Using results from other OF methods can provide such initial samples, since these results are "believed" by the other algorithms to be optimum solutions, and thus may do relatively well in POF's probability evaluation.

We denote a candidate OF algorithm as **XOF**, and our hybrid method as **POF-XOF**. The pseudocode of the hybrid approach can be seen in Algorithm 6.3.

Algorithm 6.3 POF-XOF: Hybrid Probabilistic Optical Flow

Input: Images $\boldsymbol{I}$, $\boldsymbol{I'}$, an OF method XOF, runtime limit T, observation rounds R
Output: Optical flow $\hat{\boldsymbol{f}}$

```
t ← 0; t̂ ← 0; f^0 ← XOF(I, I'); f̂ ← f^0
while t − t̂ < R AND runtime < T do
  t ← t + 1
  f^t ← POF_SAMPLE_ROUND(I, I, f^{t−1})
  if P(f^t) > P(f̂) then
    t̂ ← t; f̂ ← f^t
  end if
end while
return f̂
```

Here one round of POF Gibbs sampling is denoted as POF_SAMPLE_ROUND. The difference between POF-XOF and the original POF algorithm is that instead of initializing $\boldsymbol{f}^0$ randomly, we initialize it using the result of XOF. The previous POF algorithm is thus a special case of POF-XOF, namely the case where XOF generates a random flow assignment.

As of the choice of XOF algorithms, we will keep using the ones we used to compare with POF algorithm in Section 6.1, *i.e.*, **Classic+NL (C+NL) [44]**[12], **Correlation Flow (CF) [13]**[13] and **CLG-TV [12]**[14]. **Horn Schunck** method is not used because of its unsatisfactory performance.

The following factors should be taken into consideration when deciding whether an XOF algorithm is a good candidate for POF:

- **Speed of XOF**: XOF should be relatively fast in generating an initial result $\boldsymbol{f}^0$, so that the time saved by starting POF with $\boldsymbol{f}^0$ directly is more than the time spent on generating $\boldsymbol{f}^0$.
- **"Compatibility" of XOF with POF**: Different OF methods use different models and

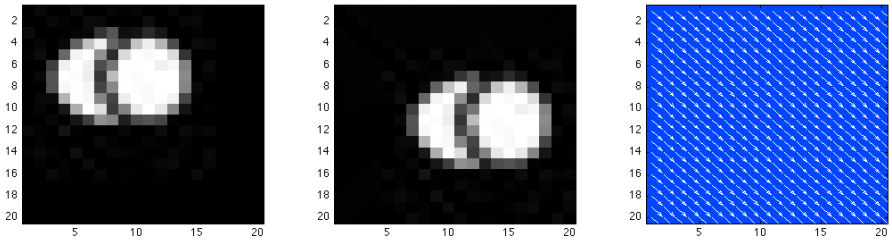
[12] http://www.cs.brown.edu/~dqsun/research/software.html
[13] http://cv.utcluj.ro/optical-flow.html
[14] http://www.cv.utcluj.ro/optical-flow.html

target functions to compute their optimized OF results. These models may not align with POF's model nicely, *e.g.*, they may not generate an $\boldsymbol{f}^0$ that has high likelihood in $P(\boldsymbol{f}^0)$, and thus may not help in speeding up the Gibbs sampling. We want to choose an XOF that produces flow result with high likelihood in POF.

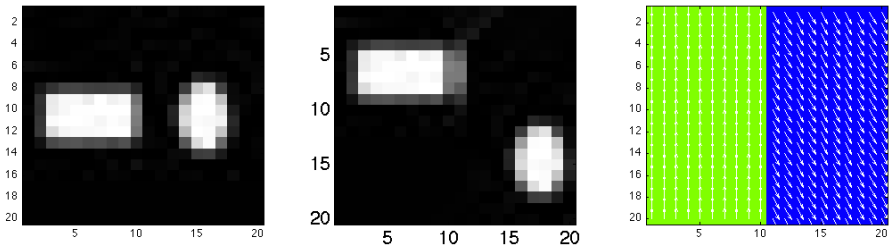
We will keep the settings used in Section 6.1 for the XOF algorithms and POF algorithms. For each algorithm, we use the default settings provided in the respective open source code package. For POF algorithm, we use $\alpha = 3, \beta = 100, \gamma = 1$.

6.3.2 Input Data

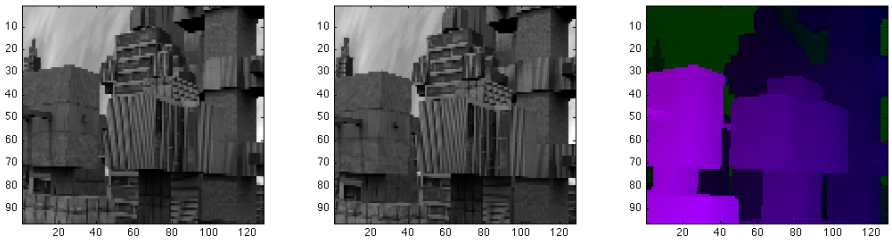
We will use both synthetic images and real world images as input to our algorithms. They are summarized in Figure 6.31.



Ventricles (Image 1, Image 2 & Ground truth flow)



Two objects (Image 1, Image 2 & Ground truth flow)



Urban3 (Image 1, Image 2 & Ground truth flow)

Figure 6.31: Input images for better POF initialization experiments

"Ventricles" and "Two objects" are both synthetic images. "Ventricles" is a simplified illustra-

tion of human cardiac images, and the entire image moved in bottom right direction. We use this image to test OF algorithms' ability to handle rigid movements for irregular shapes. "Two objects" shows movements of two objects in opposite directions, we use this since many OF algorithms do not handle such movements well. "Urban3" is a real world image taken from the Middlebury dataset [5].

To compare the speed of POF-XOF and POF algorithm, we will keep using the convergence measures defined in 6.2, *i.e.*, **Median Best Log-Likelihood (MBL)**, **Convergence Time Distribution (CTD)** and **Cumulative Convergence Time Distribution (CCTD)**. Additionally, we measure three additional values: **Median Best Endpoint Error (MBEE)**, **Median Best Angular Error (MBAE)** and **Median Best Interpolation Error (MBIE)**. The errors for a flow result is defined in Section 6.1.1. These are median best errors measured in essentially the same way as median best log-likelihood, except that here "best" means lowest error (whereas best log-likelihood is the maximum). For example, MBEE is defined as follows:

$$\text{MBEE}(t) = \text{median}\left[\min_{r=1,\cdots,r^*}\{\text{EE}(S_l^r)\}\right], \tag{6.9}$$

where

$$r^* = \underset{r}{\text{argmax}}\,\{t(S_l^r) < t\} \tag{6.10}$$

For each experiment, we run it L times where $L = 30$.

6.3.3 Comparing XOF and POF

We first compare flow results between POF and the candidate XOF algorithms. We let the POF algorithm terminate when no log likelihood improvements are made in the most recent 200 sampling rounds. The errors and runtime (denoted as RT) of all OF methods on all the input images are measured in Table 6.4. The runtimes are measured in seconds, and for POF, all values are medians taken from 30 independent sampling runs.

We can see that for both "Ventricles" and "Two objects", POF algorithm's results have much

	Ventricles				Two Objects				Urban3			
	EE	AE	IE	RT	EE	AE	IE	RT	EE	AE	IE	RT
CF	5.2	58.5	90.0	3.2	1.3	15.0	28.6	3.2	0.8	24.0	14.1	4.1
C+NL	4.9	53.6	104.1	**0.6**	1.0	12.7	25.9	**0.7**	**0.6**	**20.4**	13.6	5.1
CLG-TV	2.9	30.8	52.8	3.2	0.1	1.2	**5.9**	3.5	0.9	25.9	**9.7**	**2.4**
POF	**0.0**	**0.1**	**0.9**	71.3	**0.0**	**0.3**	10.5	52.4	1.3	37.0	10.4	291.8

Table 6.4: Comparing XOF and POF. POF generates best OF results for "Ventricles" and "Two objects", but the runtime is much longer than other OF methods.

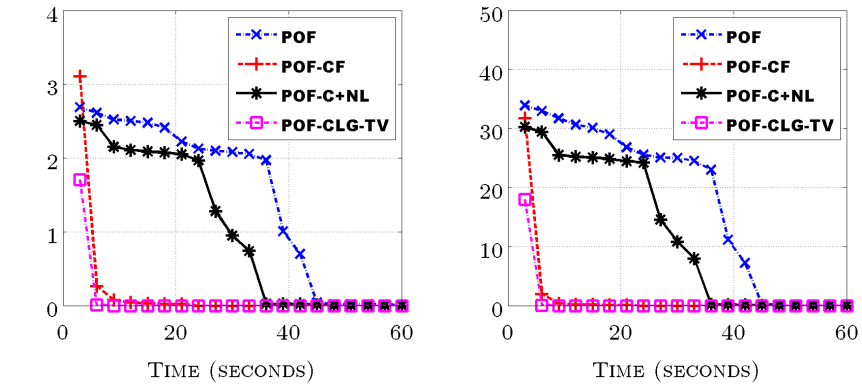
smaller errors (EE, AE and IE) than other XOF algorithms, while for the "Urban3" input it has mediocre results. However, for all these methods, POF method's runtimes (RT) are far longer than the other algorithms. Depending on the application, the superior error performance of POF may be overshadowed by the long runtime of the POF algorithm.

6.3.4 Comparing POF-XOF and POF

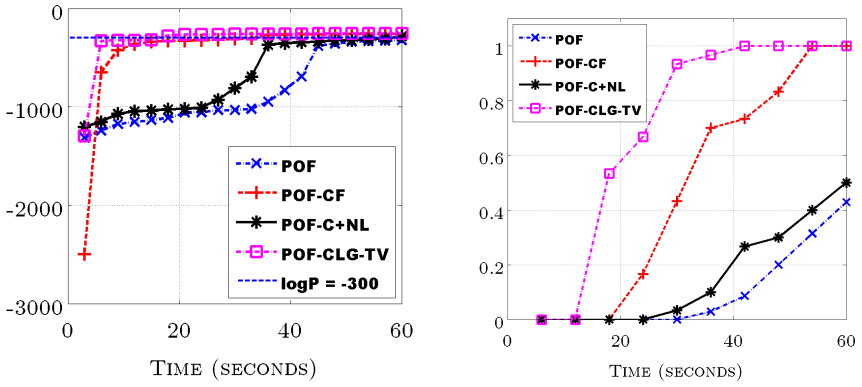
We run hybrid OF methods: **POF-CF, POF-CLG-TV** and **POF-C+NL** on the same inputs, and measure the convergence speed of these methods, comparing with POF method. The runtime is set to 60 seconds for "Ventricles" and "Two objects", and 120 seconds for "Urban3". The results are shown in Figure 6.32, Figure 6.34 and Figure 6.35. Note that all curves of POF-XOF are shifted along the time axis by the amount of time spent on XOF.

Figure 6.32 shows results on "Ventricles" input. In both MBEE and MBAE we see that using POF-XOF brings down the EE and AE errors much faster than POF method. For example, POF-CF and POF-CLG-TV were able to reach MBEE = 0 at around $t = 10$, while POF reached the same result at $t = 45$. In Figure 6.32c, we can also verify that the median best log likelihood of POF-CF and POF-CLG-TV reached the convergence threshold $\log P_{conv} = -300$ much faster than POF. The cumulative convergence time distribution plot in Figure 6.32d shows that at $t = 30$, almost all sample chains in POF-CLG-TV have reached the convergence threshold, while for POF, even at $t = 60$, only around 40% of sample chains were able to reach the convergence threshold.

Another thing to note is how the choice of XOF affected the convergence of POF-XOF. Even though POF-C+NL and POF-CLG-TV started at around the same log likelihood (Figure 6.32c), the MBL curves were drastically different. Also, though POF-CF had a very low starting log



(a) Median Best EE **(b)** Median Best AE



(c) MBL **(d)** CCTD

Figure 6.32: Comparing POF-XOF with POF, using "Ventricles" as input. For CCTD (Cumulative Convergence Time Distribution), the convergence threshold is set to $\log P_{conv} = -300$, as shown in the MBL plot. POF-CLG-TV converged the fastest, with POF-CF trailing. POF-C+NL was slower, but still faster than running POF alone.

likelihood, it was able to catch up fast with POF-CLG-TV and converged much faster than POF-C+NL. To better understand these behaviours, we plot the optical flow results from XOF and POF algorithms in Figure 6.33.

In Figure 6.33, we can see that though CLG-TV, C+NL and CF all performed badly in error metrics (see Table 6.4), their flow results are quite different. The flow result from CLG-TV is much closer to POF compared with C+NL and CF. Thus, when using result of CLG-TV as input to POF, POF was able to quickly start from what is provided, and reach convergence. This shows the importance of choosing the right initial sample for Gibbs sampling. As for C+NL, the result has similar log likelihood as CLG-TV in POF's evaluation (Equation 3.4), but the convergence path in Figure 6.32c is drastically different. That's because the probability space defined by Equation 3.4

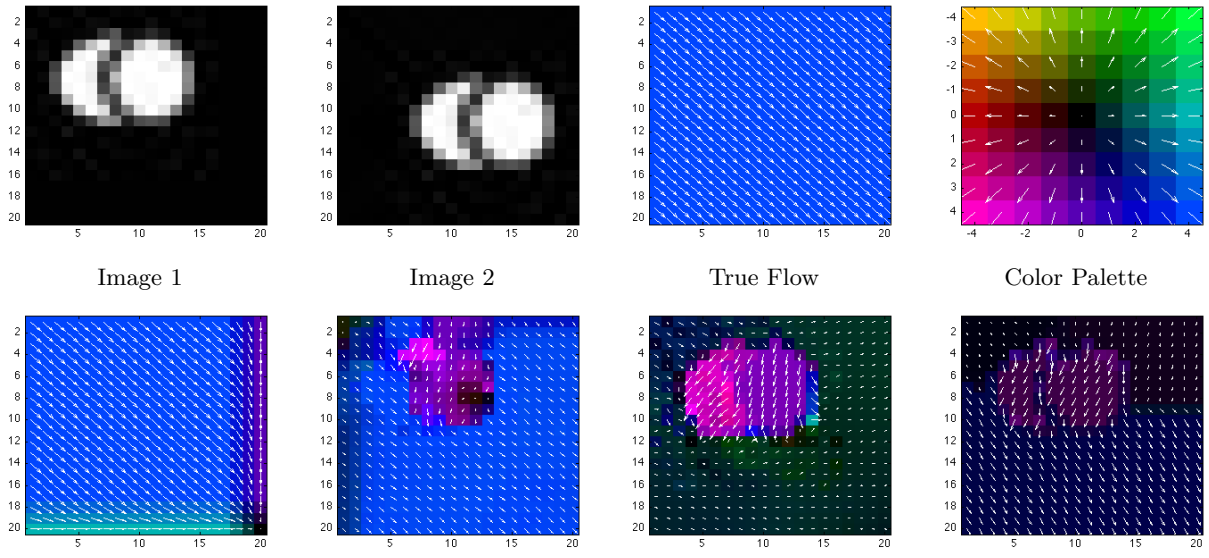
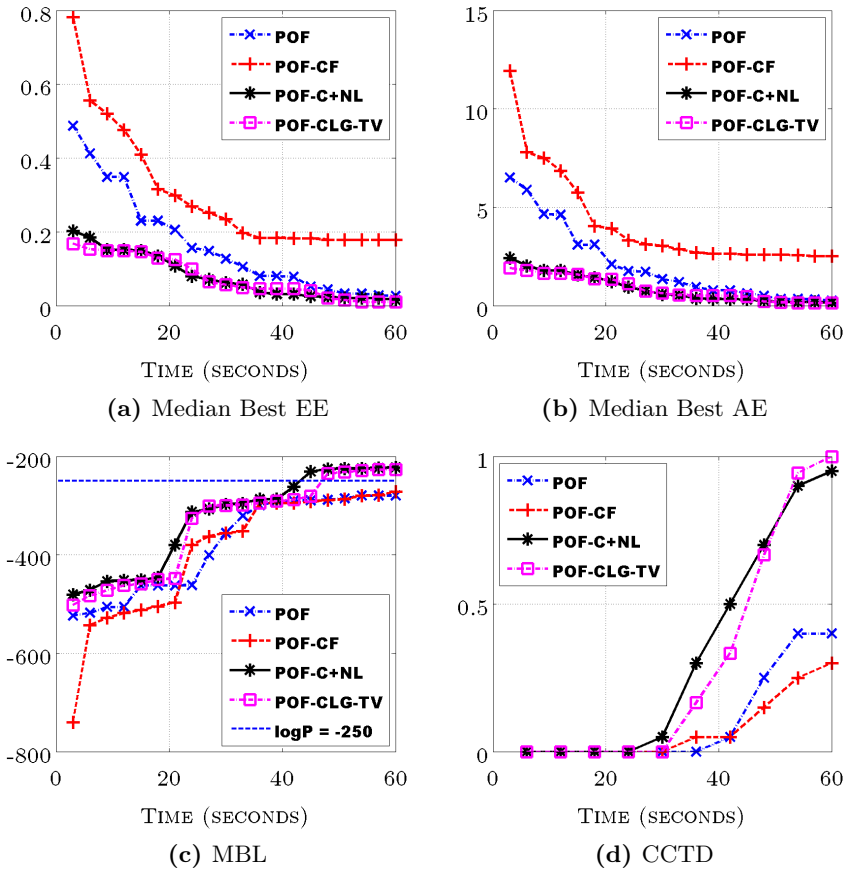


Figure 6.33: Optical Flow Results of XOF and POF on "Ventricles" input. CLG-TV's result is closest to POF, thus POF-CLG-TV was able to converge fast. CF and C+NL's results are both very different from POF, but CF's result was corrected easily by POF, while C+NL's result was further away from global optimal $\boldsymbol{f}^*$.

is multi-modal, with many locally optimal but globally non-optimal peaks. C+NL's result appears to be in one of the locally optimal peaks, but to reach the globally optimal result, it had to go through a longer sampling path than CLG-TV. This may also explain why CF algorithm's result had a lower likelihood, but was able to catch up fast.

Figure 6.34 shows the convergence on "Two objects" input images. Here both POF-CLG-TV and POF-C+NL performed better than POF, while POF-CF performed worse. Comparing with Figure 6.32, where POF-CF performed well, we can see that even the same algorithm may not always generate a flow that "fits" into the POF algorithm.

Figure 6.35 shows convergence on "Urban3" input images. The input images are much larger and complicated than the previous synthetic images, thus POF's performance was not outstanding (see Table 6.4). However, using our hybrid approach, POF-XOF was able to improve the IE measure (Figure 6.35b), when using CF and C+NL as XOF, from the original IE = 10.4 (in Table 6.4) to IE $\sim$ 9.7 at $t = 120$. Using POF alone, to be able to reach such result, it could have taken much longer sampling time. The MBL and CCTD curves also suggest that POF-XOF has much faster



(a) Median Best EE **(b)** Median Best AE

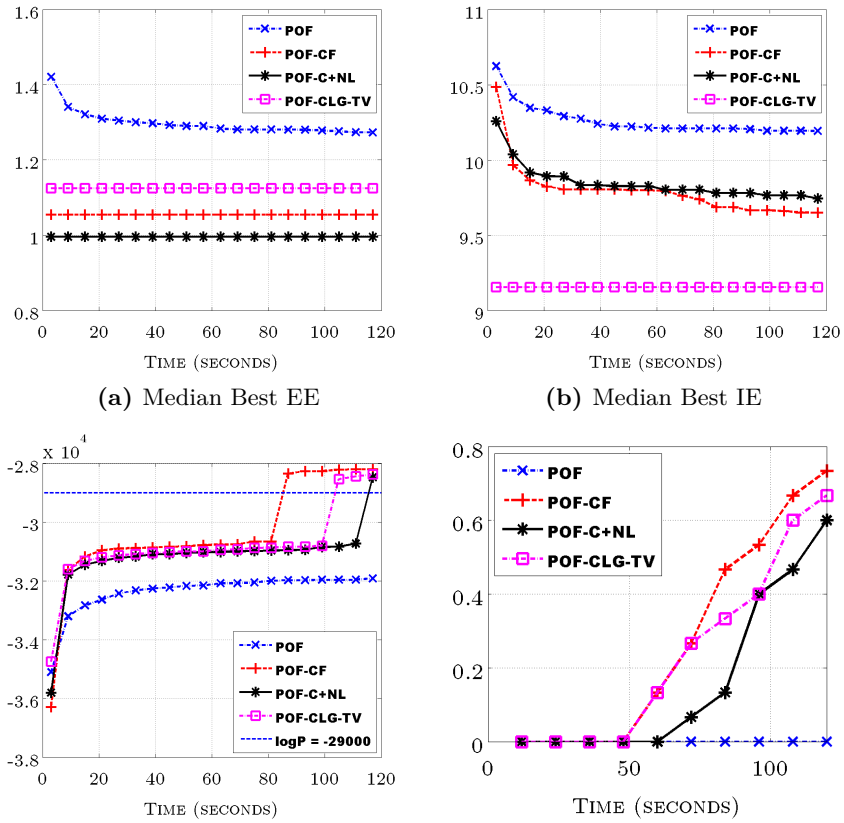
(c) MBL **(d)** CCTD

Figure 6.34: Comparing POF-XOF with POF, using "Two objects" as input. Convergence threshold in CCTD is set to $\log P_{conv} = -250$, as shown in MBL plot. Here, both POF-CLG-TV and POF-C+NL converged faster than POF, while POF-CF was slower.

convergence than POF.

Comparing Figure 6.32, Figure 6.34 and Figure 6.35, we see that the convergence pattern is drastically different. The reason is that different OF algorithms generate completely different flow results for the same input images, and it greatly affects the starting sample of the POF algorithm. The other reason is, the ability of the same OF algorithm to handle different input images is quite different. Determining the "best" XOF algorithm for POF-XOF is thus still a trial and error process.

In the end, we show the runtime comparisons of POF-XOF with POF in Figure 6.36. Here we set termination condition $T = \infty$ and $R = 200$ for both POF-XOF and POF in Algorithm 6.3. All runtimes are medians from 30 runs.



(a) Median Best EE **(b)** Median Best IE

(c) MBL **(d)** CCTD

Figure 6.35: Comparing POF-XOF with POF, using "Urban3" as input. Convergence threshold in CCTD is set to $\log P_{conv} = -2.9 \times 10^4$, as shown in MBL plot.

As expected, POF-XOF were able to speed up the original POF. For "Ventricles" and "Urban3" inputs, all 3 candidate XOF algorithms were able to generate a speedup. For "Two objects", only POF-C+NL had noticable speedup. It may be due to the fact that these XOF algorithms do not perform well for objects with complete opposite moving directions (see errors in Table 6.4), and thus could not help POF to get to a better starting position.

6.3.5 Conclusion

When we are using sampling to solve probability based OF methods, the initial sample needs to be chosen carefully to have a fast sampling process. We showed that significant speedups in POF method can be achieved by applying result from other OF methods, *i.e.*, using the hybrid POF-XOF approach. We also showed that the choice of input OF algorithm affects the convergence of

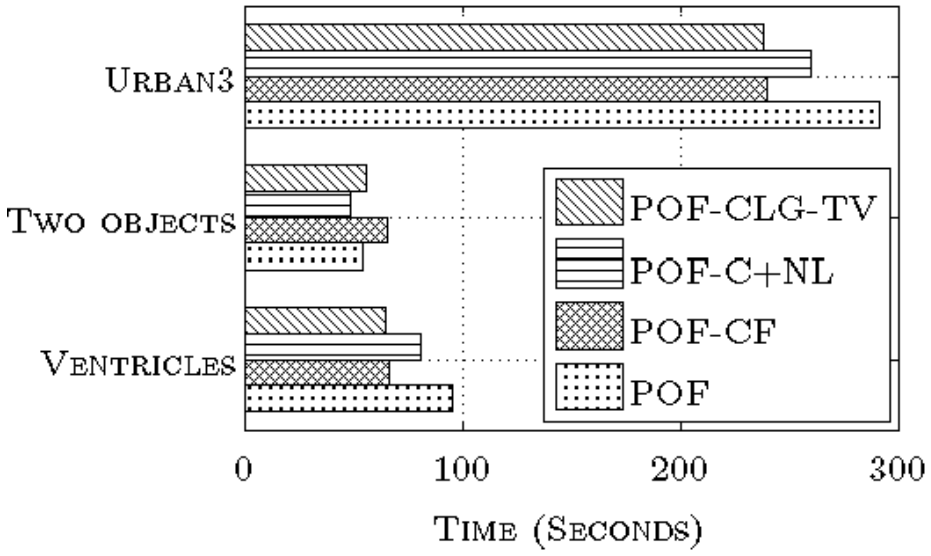


Figure 6.36: Runtime comparison of POF-XOF and POF

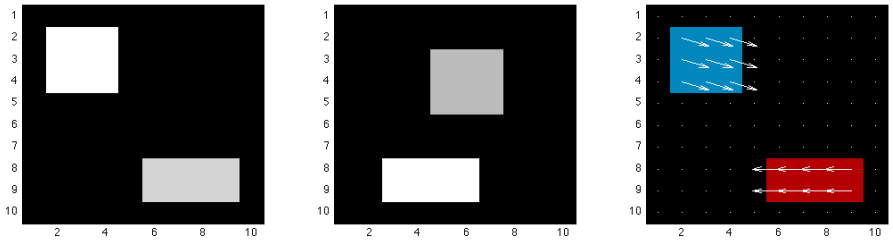
the hybrid algorithm in various degrees, depending on the input images given and how close the result of input OF is to that of the probability-based OF method. As of now, there is no good way to determine which OF method is the best to be used in POF-XOF, and it remains as a future research direction.

6.4 Using Staged Annealing

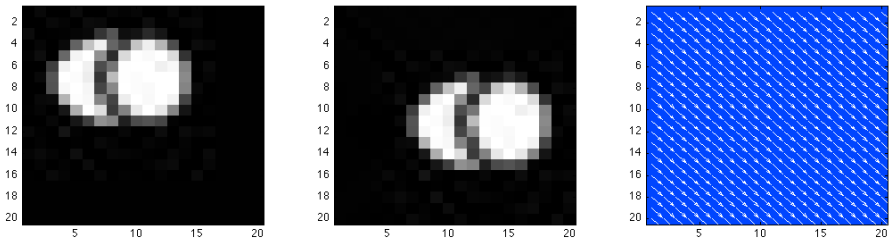
In this section we will evaluate how staged annealing improves POF algorithm. In Section 4.4 we discussed how simulated annealing is able to speed up POF algorithm. We adapted simulated annealing into staged annealing in POF, as in Algorithm 4.5. In Algorithm 4.5, instead of continously lowering temperature T every round as in simulated annealing, we keep it consistent within a stage, and lower it only between stages. The sample with highest likelihood from each stage is passed as input to next stage, instead of using the sample from previous sampling round.

In our experiment, we run POF algorithm with and without staged annealing on each input for $L = 30$ times. For stage-annealed POF, we set observation round $R = 100$, and stage count $S = 3$. The temperature decay D is set to 0.5. We will use two inputs, as shown in Figure 6.37. "Two rectangles" shows two rectangles moving positions and changing intensities, and "Ventricles" shows

synthetic ventricles moving in bottom right direction. For "Two rectangles" input, each sample run is bounded to 30 seconds, and for "Ventricles" it is 60 seconds.



Two rectangles (Image 1, Image 2 & Ground truth flow)



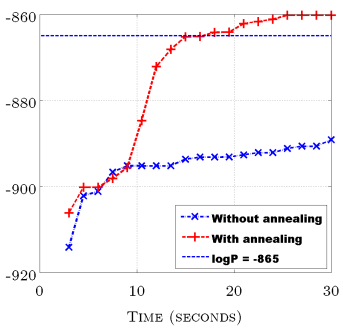
Ventricles (Image 1, Image 2 & Ground truth flow)

Figure 6.37: Input images for better POF initialization experiments

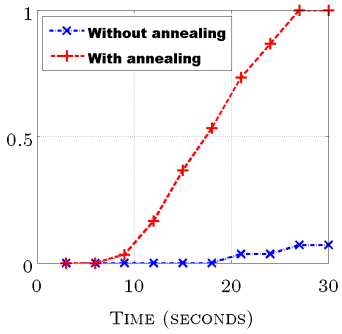
The resulting errors and log likelihoods are shown in Figure 6.38. Here, we use the convergence metrics **Median Best Log-Likelihood (MBL)** and **Cumulative Convergence Time Distribution (CCTD)** defined in Section 6.2.

Figure 6.38a and Figure 6.38b are results on "Two rectangles", and Figure 6.38c and Figure 6.38d are results on "Ventricles". In both MBL curves (Figure 6.38a and Figure 6.38c), the log likelihoods with and witout staged annealing are basically the same up to certain point (in Figure 6.38a it is $\sim$9 second, and in Figure 6.38c it is aroud 30 second), which shows that the staged annealing were in the first stage. In the first stage, the temperature is $T = 1$, and it has no impact on the original POF sampling (see Equation 4.20).

Then, log likelihood starts to differ for annealed / non-annealed POF, when the temperature begins to lower in stage-annealed POF. For the "Two rectangles" example (Figure 6.38a), the MBL of stage-annealed POF outgrew the non-annealed POF starting from $t = 10$, and the difference in log likelihood remained for the entire testing duration. For the "Ventricles" example (Figure 6.38c), the difference started at $t = 30$, and the non-annealed POF started to catch up at around $t = 45$.

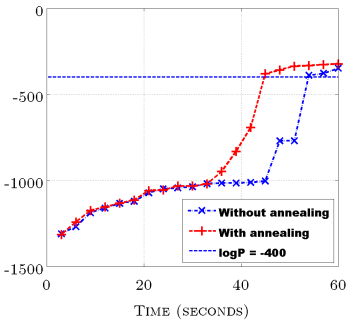


(a) Median Best Log-Likelihood (MBL)

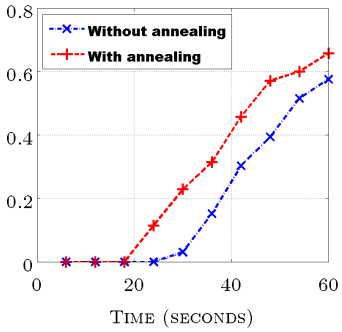


(b) Cumulative Convergence Time Distribution (CCTD). The convergence threshold is set to $\log P_{conv} = -865$, as shown in the MBL plot.

Using "Two rectangles" as input



(c) Median Best Log-Likelihood (MBL)



(d) Cumulative Convergence Time Distribution (CCTD). The convergence threshold is set to $\log P_{conv} = -400$, as shown in the MBL plot.

Using "Ventricles" as input

Figure 6.38: Comparing POF with and witout staged annealing. For both "Two rectangles" and "Ventricles", using stage-annealed POF allows faster increase in best log likelihoods.

The CCTD curves also show similar findings. For "Two rectangles" (Figure 6.38b), stage-annealed POF were able to reach $\log P_{conv} = -865$ at around $t = 27$ for 100% of sample chains, while only about 10% of non-annealed POF reached same threshold at that time. For "Ventricles" (Figure 6.38d), at each time step after $t = 20$, the percentage of conversion for annealed POF was roughly 10-15% higher than the non-annealed version.

From these experiments we can see staged annealing does help POF algorithm to find samples with higher log likelihood faster, and thus speed up POF algorithm. The convergence speed improvement from doing staged annealing, however, will differ by input images.

We show the log likelihood curve for one typical stage-annealed POF sampling chain, for the "Two rectangles" input, in Figure 6.39. The log likelihood in different stages are marked. An earlier stage has bigger variance in log likelihoods than later stage (*e.g.*, Stage 1 vs Stage 2), showing that when temperature is high (in earlier stage), the Gibbs sampler has more flexibility in choosing from possible states than when temperature is low. When temperature is lowered, the sampler is more inclined to sample from higher-likelihood states.

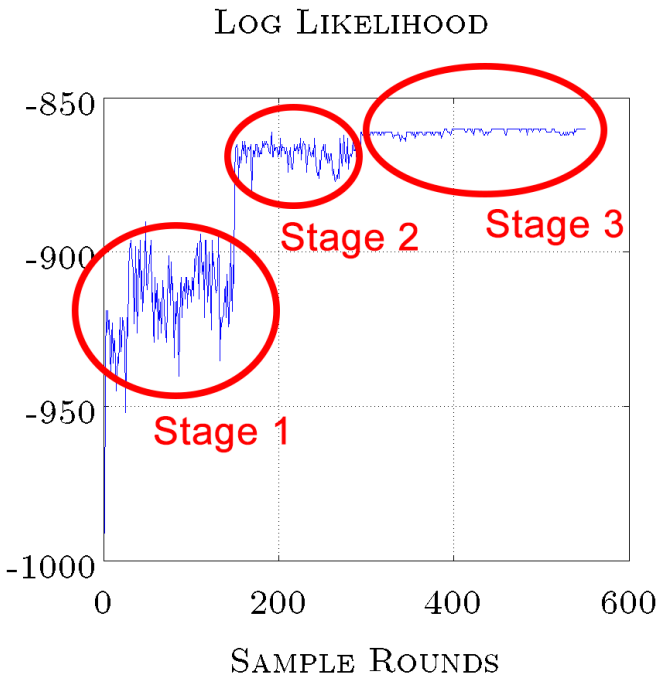


Figure 6.39: Log likelihood curve for stage-annealed POF on "Two rectangles" input. Stages are marked. In an earlier stage, temperature T is higher, which allows more flexible sampling and bigger variance in log likelihoods from samples.

6.5 Block Gibbs Sampling

In this section, we will study the impact of doing blocking in Gibbs sampling. The details of block Gibbs sampling is previously discussed in Section 4.3. We will experiment on different methods of generating blocks, and test on different block sizes and block counts.

6.5.1 Preparation

As input to the block Gibbs sampling, we will use UAI benchmark PGM dataset[15]. UAI benchmark PGM dataset is a publicly accessible graphical model dataset for the evaluation of various graphical model related algorithms. The dataset includes a variety of PGM from a range of application domains. It has both Bayesian networks and Markov random fields, and the size of graphs span from small (∼50 nodes) to large (∼76,000). Some of the graphs are designed to have huge treewidth (∼60) so that exact inference algorithms are not applicable.

The graph we will be using is shown in Figure 6.40. It is chosen from **Students** graph set, which is a set of "Relational Bayesian networks constructed from the Primula tool". It is a Bayesian network with 376 binary variables, 376 factors with an average factor size of 2.72 (mostly 3-node factors). It has a relatively large treewidth (∼20), and the nodes are highly correlated. This graph is also used in [45] for their blocking algorithm. We will later add another graph for the experiment, *i.e.*, image denoising grid MRF.

We are not experimenting with the POF MRF described in Section 3.2, since in the POF MRF, each node (*i.e.*, flow vector) has a large number of discrete states (*i.e.*, possible directions and lengths). In block Gibbs sampling, before sampling each block, the blocks need to be caliberated using message passing algorithm. The time complexity of message passing is exponential to the treewidth of nodes in the block, *i.e.*, the largest clique size minus one, in the factor graph created from the block. If there are N nodes in a clique, to be able to pass the message from this clique to another one, there needs to be $O(S^N)$ additions. Say we are allowing a pixel to move 5 pixels away, then each flow vector has $(2 \times 5 + 1)^2 = 121$ possible states. When a clique has (say) 5 nodes, the number of additions is in the order of $121^5 \approx 10^{10}$, which is overly expensive for computation. Unless there are ways to generate blocks with controlled treewidth, doing block Gibbs sampling on graphs with high-state count nodes is computationally infeasible.

We implemented the block Gibbs sampling described in Algorithm 4.3 in Scala programming language[16], which is a highly efficient language with performance on par with Java language. It also compiles to Java byte code, and can be run on any Java Virtual Machine (JVM). Our experiments

[15] http://graphmod.ics.uci.edu/uai08/Evaluation/Report/Benchmarks
[16] http://www.scala-lang.org/

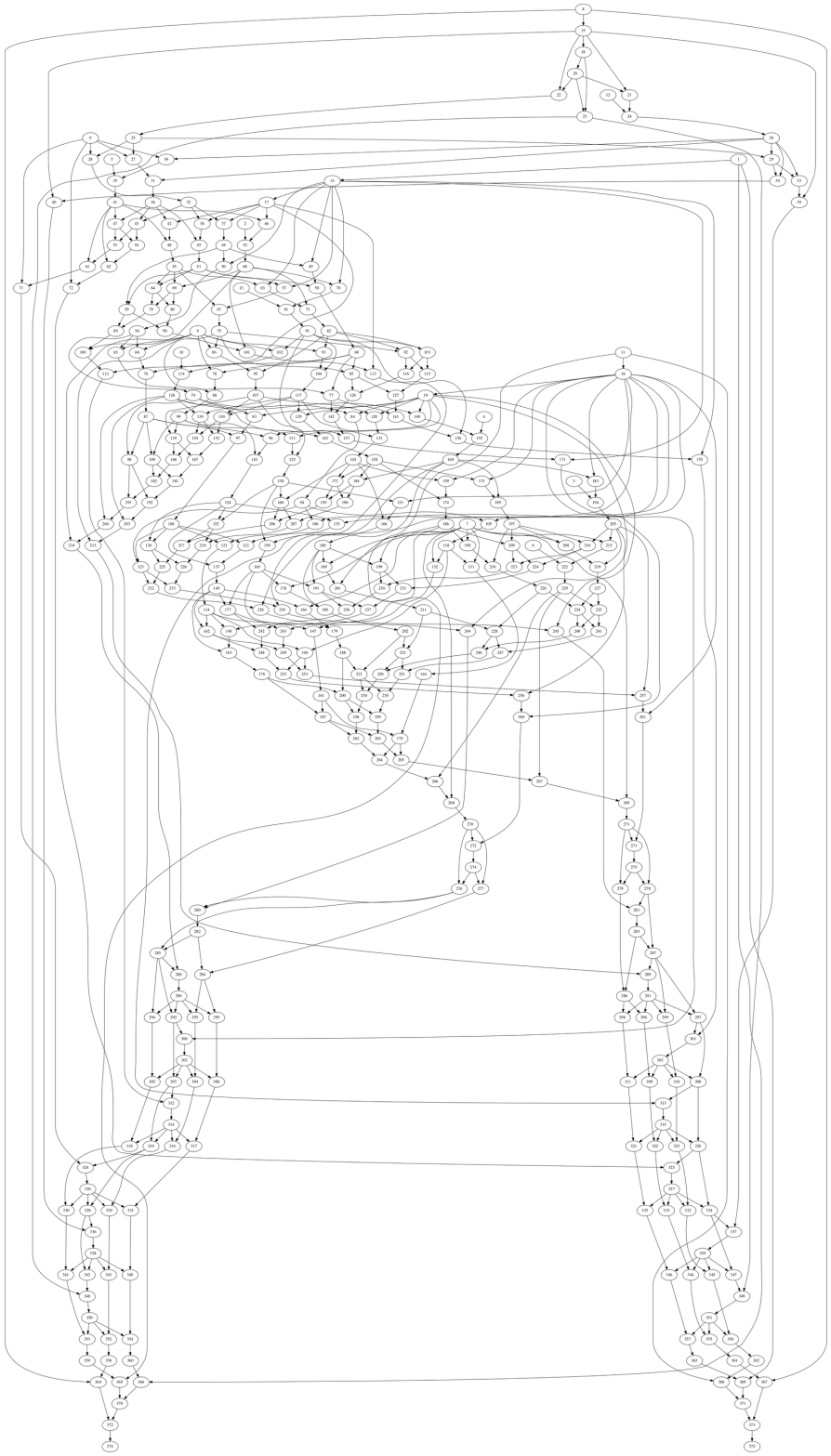


Figure 6.40: "Students" graph. It is a Bayesian network with 376 binary variables, 376 factors with an average factor size of 2.72.

are conducted on a MacBook Pro with 2.6GHz Intel Core i7 (quad core) and 16GB RAM.

In Agorithm 4.3, the inputs to the algorithm include the following: **Input graph** G, **Gibbs sampling duration** T, **Number of blocks** B, **max treewidth** Tw, **minimum block size** S_{min}, **maximum block size** S_{max}, **minimum samples before blocking** N_S and **block growth method** $\mathcal{M}$. We denote all experiments with the same setting as E, which is characterized by the inputs, *i.e.*, $E \equiv E(G, T, B, Tw, S_{min}, S_{max}, N_S, \mathcal{M})$.

We will use **Average Best Log-Likelihood (ABL)**, **Convergence Time Distribution (CTD)** and **Cumulative Convergence Time Distribution (CCTD)** to compare the convergence speed of different sampling processes, as described in Section 6.2. For each experiment, we run it L times where $L = 100$. For each sample S_l^r, in addition to log likelihood $\log P(S_l^r)$ and current sampling time $t(S_l^r)$, we also record these statistics: Block counts, average block size $\bar{B}(S_l^r)$, maximum treewidth of the blocks $Tw(S_l^r)$, and round duration $t_R(S_l^r)$.

6.5.2 Impact of Block Growth Method

In ComputeScore algorithm (Algorithm 4.4), we discussed four types of heuristics when growing a block: `max_correlation, min_conditional, min_marginal` and `min_coverage`. Here `min_conditional` is a heuristic metric used in [19], and the other three are proposed by us.

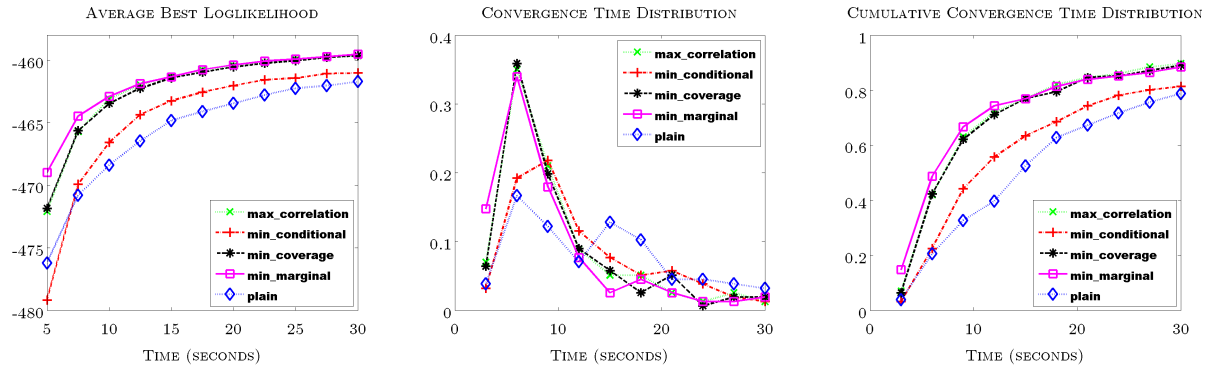
We ran block Gibbs sampling (Algorithm 4.3), using each of the heuristics to grow the blocks. The settings we used in the experiments are as follows. Sampling with each setting is run 100 times ($L = 100$).

- **Input graph** G: Students Bayesian network with 376 binary nodes
- **Gibbs sampling duration** T: 30 seconds.
- **Number of blocks** B: 10
- **Max treewidth** Tw: 12
- **Minimum block size** S_{min}: 10
- **Maximum block size** S_{max}: 200
- **Minimum samples before blocking** N_S: 100

- **Block growth method** $\mathcal{M}$: `max_correlation`, `min_conditional`, `min_marginal` and `min_coverage`

Figure 6.41 shows the ABL, CTD and CCTD plots. Here we showed the convergence situation of block Gibbs sampling using all four types of heuristics, as well as using plain sequential Gibbs sampling. For CTD and CCTD, we used $\log P_{conv} = -465$.

Clearly using block sampling greatly sped up the sampling process, compared to using plain Gibbs sampling. Among the four different block growth heuristics, `max_correlation`, `min_marginal` and `min_coverage` performed better than `min_conditional`. The CCTD curve (Figure 6.41c) shows that at $t = 10$, `max_correlation`, `min_marginal` and `min_coverage` all have $\sim 70\%$ chance of reaching convergence, while `min_conditional` has $\sim 50\%$ chance of converging. For plain Gibbs sampling, it has $\sim 35\%$ chance.



(a) Average Best Likelihood **(b)** Convergence Time Distribution **(c)** Cumulative Convergence Time Distribution

Figure 6.41: Comparing different block growth method in block Gibbs sampling and plain Gibbs sampling. `max_correlation`, `min_marginal` and `min_coverage` performed better than `min_conditional`, and all blocking methods performed better than plain Gibbs sampling.

Figure 6.42 shows the average round time for the sampling methods. Understandably, plain Gibbs sampling runs much faster in each round compared with block Gibbs sampling. For various block growth heuristics, `min_conditional` and `max_correlation` take the longest time, since both of them require counting the joint occurrences of current states of the nodes in the block and the node in consideration, in all past samples (Equation 4.15 and Equation 4.17). For `min_marginal`, it only needs to count the occurrences of the current state of the node in consideration in all past samples (Equation 4.3.8), thus is much faster. In the end, `min_coverage` only needs to iterate the

past $|X|$ samples (Equation 4.3.8), thus is the fastest.

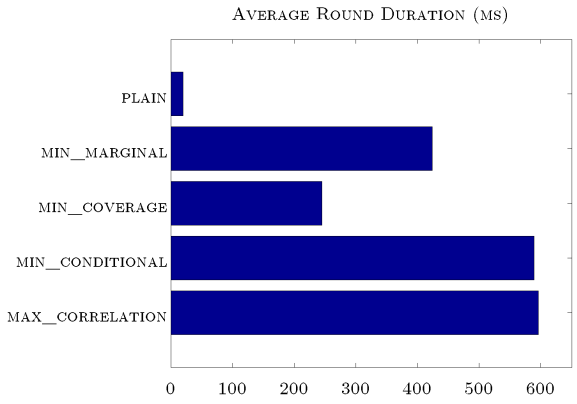


Figure 6.42: Average round time of sampling. `min_conditional` and `max_correlation` take the longest time due to higher algorithm complexity. All blocking methods take significantly longer time to finish than plain Gibbs sampling, due to block generation and joint sampling of nodes in the blocks.

Even though the blocking methods take considerably longer time than plain Gibbs sampling in each round, they are still able to make the sampling converge faster (Figure 6.41). It shows indeed jointly sampling blocks of correlated nodes makes state transition much faster in the MCMC chain. When doing blocking, using `max_correlation, min_marginal` and `min_coverage` heuristics seem to be the better way of growing blocks.

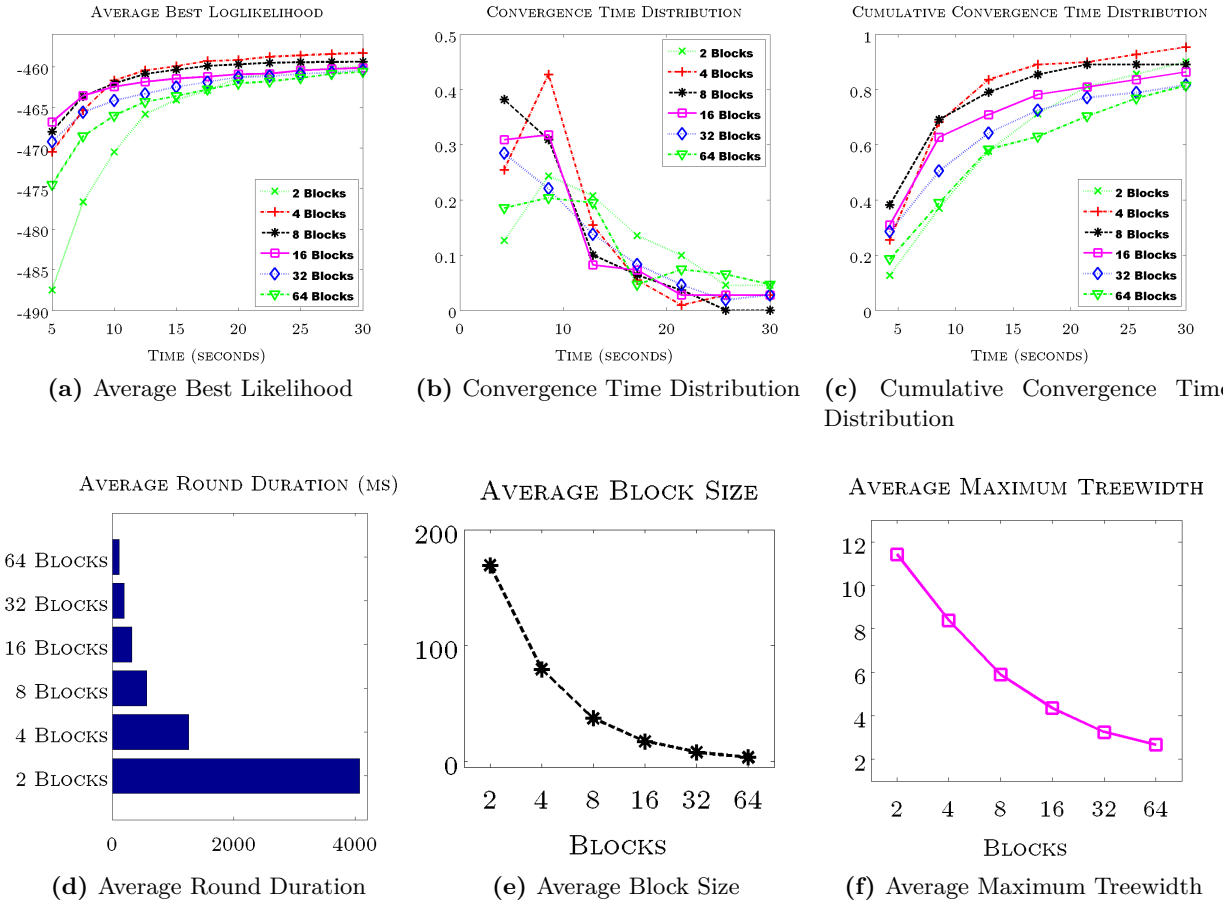
6.5.3 Impact of Block Size and Count

In this experiment, we test how the block counts and block sizes affect sampling process. The settings we used are as follows:

- **Input graph** G: Students Bayesian network with 376 binary nodes
- **Gibbs sampling duration** T: 30 seconds.
- **Number of blocks** B: 2, 4, 8, 16, 32, 64
- **Max treewidth** Tw: 14
- **Minimum block size** S_{min}: 1
- **Maximum block size** S_{max}: 200
- **Minimum samples before blocking** N_S: 100

- **Block growth method $\mathcal{M}$:** `max_correlation`

Figure 6.43 shows the ABL, CTD and CCTD plots. Here we showed the convergence situation of block Gibbs sampling using different block counts. For CTD and CCTD, we used $\log P_{conv} = -465$. Additionally, it shows the average round duration (Figure 6.43d), average block size (Figure 6.43e) and average maximum treewidth (Figure 6.43f).



(a) Average Best Likelihood **(b)** Convergence Time Distribution **(c)** Cumulative Convergence Time Distribution

(d) Average Round Duration **(e)** Average Block Size **(f)** Average Maximum Treewidth

Figure 6.43: Comparing different block counts and sizes in block Gibbs sampling. $B = 4, B = 8$ have the best performances, with $B = 16, B = 32$ trailing. Having too few blocks ($B = 2$) or too many blocks ($B = 64$) all have bad performances. As block count decreases, block size increases (Figure (e)), and that leads to increased treewidth (Figure (f)), thus the round duration increases (Figure (d)).

We can see that $B = 4, B = 8$ lead the convergence metrics, with $B = 16$ and $B = 32$ following. $B = 2$ and $B = 64$ have the slowest convergence. We analyze the reason as the following:

- When there are too few blocks, such as when $B = 2$, the block size is too large (Figure 6.43e), and large blocks tend to have large treewidth (Figure 6.43f). The time of sampling a block

is exponential to treewidth, thus, when block number is small, the exponential increase in sampling time cannot compensate the benefit of jointly sampling larger blocks of nodes.

- When there are too many blocks, such as when $B = 32$ or 64, the block size is too small (as small as 1 or 2 nodes), and sampling these small blocks of nodes jointly do not help much in transitioning the graph's state faster (since eventually when block size goes down to 1, it will become plain Gibbs sampling). Instead, the overhead associated with sampling these small blocks, such as generating blocks, building junction tree, doing caliberation, etc, outweighed the benefit of jointly sampling blocks of nodes.

Figure 6.44 summarizes the tradeoff. In our particular experiment, when $B = 4, B = 8$ led to optimum convergence, which makes the optimum block size to be around $40 \sim 100$. However, the optimum size in this experiment may not apply to other input graphs, since the convergence speed depends on many factors, including block count (affects overhead of generating blocks and distribution workload), correlation between nodes in blocks (affects state transition speed), state number for nodes and blocks (affects speed in sampling nodes jointly) and so on. Among these factors, correlation between nodes in blocks is hardest to estimate, and it differs even for blocks with same sizes. One of the possible future work direction is thus trying to estimate the optimum block size during block Gibbs sampling, and gradually adjust the block size to approach the optimum to speed up later sampling.

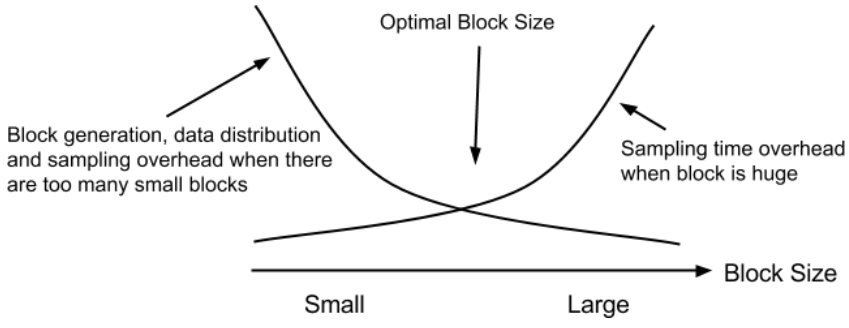
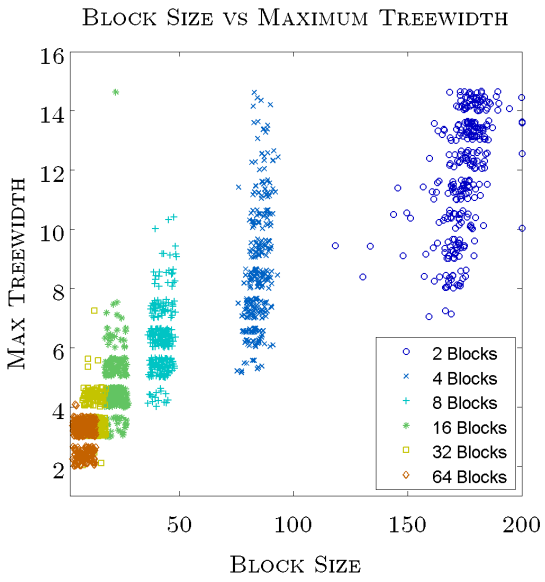
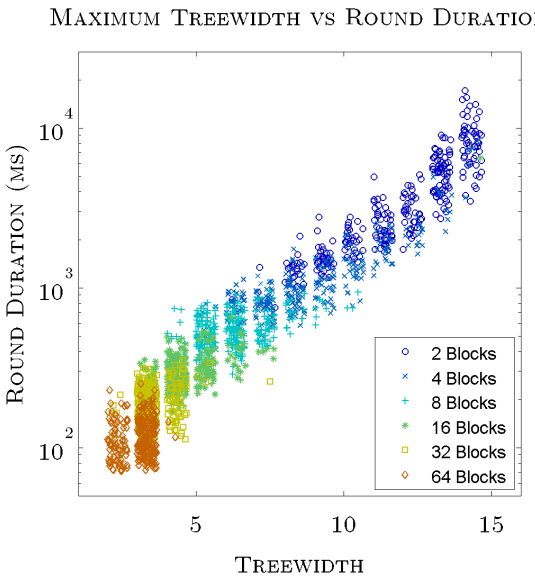


Figure 6.44: Overhead associated with block Gibbs sampling

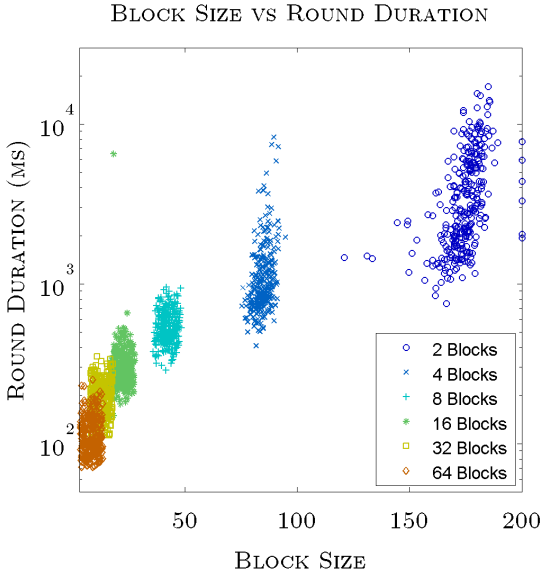
It is interesting to look at the relationship among block count, block size, maximum treewidth and round duration. In Figure 6.45 we show the scatter plot using sample points collected.



(a) Block Size vs Maximum Treewidth. Treewidth is cut out at $Tw = 14$. Blocks with bigger treewidth are removed.



(b) Maximum Treewidth vs Round Duration. Round duration can be seen to be exponential to maximum treewidth.



(c) Block Size vs Round Duration

Figure 6.45: Relationship among block count, block size, maximum treewidth and round duration. Each dot represents a sample. Larger block size leads to bigger treewidth, which results in exponential growth of sampling time in a round.

Figure 6.45a shows relation between block size and maximum treewidth, when block size varies. Clearly, when block size increases, maximum treewidth increases almost linearly, with random fluctuations. Note here when $B = 2$, the maximum treewidth is cut out at $Tw = 14$, because

we are limiting the maximum treewidth as 14 in our experiment setting. All blocks with a larger treewidth are discarded.

Figure 6.45b shows relation between maximum treewidth and round duration. Here the trend is very clear: Round duration is exponential to maximum treewidth. It also proves that the complexity of junction tree algorithm message passing is indeed exponential to the maximum treewidth.

Figure 6.45c shows relation between block size and round duration. The round duration is almost exponential to the block size, and that is easy to understand: Treewidth is almost linear to block size (Figure 6.45a), round duration is exponential to treewidth (Figure 6.45b, thus round duration is almost exponential to block size. This explains the overhead in Figure 6.44: When block size increases, the sampling time overhead outweights the benefit of jointly sampling larger blocks of nodes.

6.6 Parallelized Gibbs Sampling

In this section we experiment with Gibbs sampling parallelization methods, as discussed in Chapter 5. Two methods will be evaluated: **Chromatic Gibbs sampling** and **parallel block Gibbs sampling**.

6.6.1 Chromatic Gibbs Sampling

The idea of chromatic Gibbs sampling is discussed in detail in Section 5.4. In short, chromatic Gibbs sampling does a minimum coloring of the PGM so that nodes with the same color are not adjacent to each other, then go through each color and sample all nodes in that color in parallel. As shown in Equation 5.7, the time complexity for one round of chromatic Gibbs sampling is $O(N/P + C)$, where N is the number of nodes, P is the number of parallel computing units, and C is the number of colors. In contrast, the time complexity of one round of sequential Gibbs sampling is $O(N)$. The theoretical speedup from using chromatic Gibbs sampling is thus

$$\text{Speedup} = \frac{N}{N/P + C}. \tag{6.11}$$

When C is small (say, $C = kN$, where $k \ll 1$), the speedup can approach P:

$$\text{Speedup} = \frac{N}{N/P + C} = \frac{N}{N/P + kN} = \frac{1}{1/P + k}$$
$$\lim_{k \to 0} \text{Speedup} = P$$

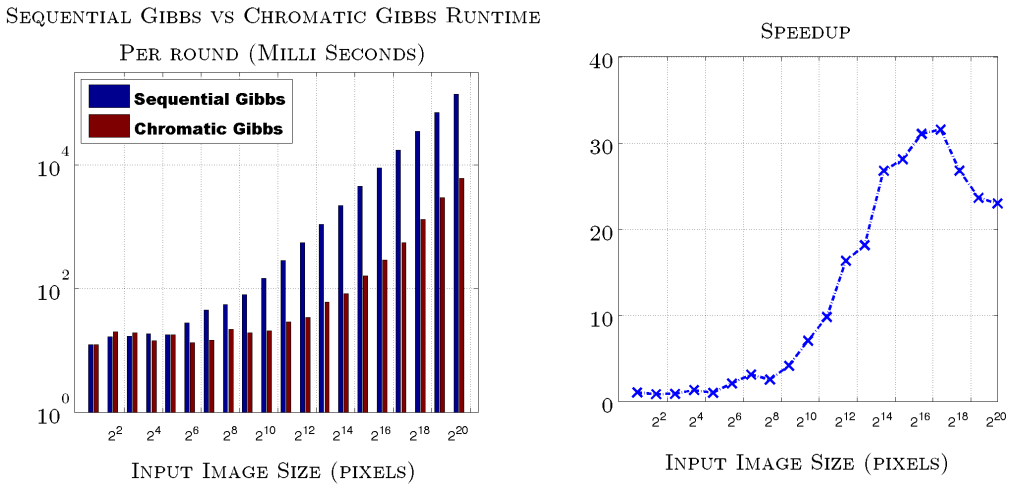
We will use POF's MRF model to experiment the speedup of chromatic Gibbs sampling. We will run POF with chromatic Gibbs sampling and sequential Gibbs sampling, and compare the runtimes of one round of sampling. Since the runtime of one round of Gibbs sampling (both chromatic and sequential) only depends on the size of input image, we vary the input image size N (pixels) from 2^1 to 2^{20}. All input images are randomly generated. For each input image size, sampling are conducted 30 rounds for both chromatic and sequential Gibbs, and average runtime is used.

The experiment is conducted on a MacBook Pro with 2.6GHz Intel Core i7 (quad core) and 16GB RAM. The programs for both chromatic Gibbs sampling and sequential Gibbs sampling are written in Matlab. For chromatic Gibbs sampling, the parallel sampling for nodes with the same color are executed by Matlab's vectorized operation, *i.e.*, all nodes that can be sampled together are put into a vector structure in Matlab, and sampled together.

Matlab doesn't have an official document about how many cores do vectorized operations use, nor how much speedup it has over sequential operation, thus we don't know the exact P number in Equation 6.11. However, the purpose of this experiment is not in exact evaluation of speedup, but rather a verification of speedup capability of chromatic Gibbs sampling.

The speedup result is shown in Figure 6.46.

When input image size is small ($< 2^5$), chromatic Gibbs and sequential Gibbs have similar



(a) Runtime **(b)** Speedup

Figure 6.46: Comparing chromatic Gibbs sampling and sequential Gibbs sampling. As input image size increases, chromatic sampling starts to demonstrate significant speedup compared to sequential sampling.

runtime, because the overhead associated with running parallelized sampling overshadowed any speed gain. When the size keeps increasing, chromatic Gibbs starts to have significant speedup, until the speedup tops at $N = 2^{16}$ to about 32×. It shows the relative overhead in parallel sampling is deminishing. Then, the speedup drops a bit back to around 24×. The reason for this may be associated with how Matlab handles vectorized computation. It is able to handle vectorized computation very efficiently up to $N = 2^{16}$, but when the vector size becomes larger, the cost of running vectorized computation in Matlab starts increasing, making the speedup drop from its optimum value.

Overall, chromatic sampling demonstrated significant speedup compared to sequential Gibbs sampling. For an image with at least moderate size ($N = 2^{12}$ corresponding to a 64x64 image), the speedup was bigger than 16×. For some image size (*e.g.*, $N = 2^{16}$, corresponding to 256x256 image), the speedup can be bigger than 30×.

6.6.2 Parallel Block Gibbs Sampling

Block Gibbs sampling is introduced in Section 4.3, and we discussed a simple way to parallelize it in Section 5.5, with the pseudocode shown in Algorithm 5.3. In this section we experiment with the parallelized block Gibbs sampling.

From Algorithm 5.3, we can see the parallelization is applied when sampling the blocks. Instead of sampling each block sequentially, we can simaltaneously sample all blocks in parallel. We denote the blocks as $\boldsymbol{B} \equiv \{B_i\}$, and the time to sample each block as T_i, then the time to run sequential block Gibbs sampling will be

$$\text{Runtime of sampling blocks sequentially} = \sum_i T_i \tag{6.12}$$

When we parallelize it, each block will be sent to a separate computation unit and sampled. Assume there are P computation units, denote the blocks that are sent to j-th computation unit as $\boldsymbol{B}_j \equiv \{B_{i,j}\}$. In addition to the time to sample these blocks $T_{i,j}^{Comp.}$, there's also the time to distribute the data of each block to and from the computation units, denoted as $T_{i,j}^{Comm.}$. The time spent on j-th computation unit will be $\sum_i \left(T_{i,j}^{Comp.} + T_{i,j}^{Comm.}\right)$. The total time spent on sampling blocks in one round will equal to the time spent on the longest running computation unit:

$$\text{Runtime of sampling blocks in parallel} = \max_j \left\{ \sum_i \left(T_{i,j}^{Comp.} + T_{i,j}^{Comm.}\right) \right\} \tag{6.13}$$

In the ideal case, where the communication between computation units do not cost time, and all blocks take equal time to sample, the time of sampling all blocks in parallel will be P-times faster than the sequential version. However, in Algorithm 4.1 (block Gibbs sampling), when generating the blocks, their sizes are not always equal, and the treewidth of each block is non-deterministic, thus the time spent on sampling each block will be different. Also, when block size is big, the time to transfer data to and from other computation unit is not negligible. Thus the overall speedup will be smaller than P.

In our experiment, we continue using the "Students" network (Figure 6.40) used in the experiment for block Gibbs sampling (Section 6.5). We will increase the block count from 2 to 64, and test how much speedup parallel block Gibbs sampling gives over sequential block Gibbs sampling. The parallel block Gibbs sampling algorithm is implemented in Scala[17], and the parallelization is done in a multi-core shared memory architecture. Scala language provides naive support for running

[17] http://www.scala-lang.org/

multi-core applications, using their **parallel collections**[18]. The parallel collections will detect the number of processors in the current computer, and allocate one thread for each processor for the parallelization. Our experiments are conducted on a MacBook Pro with 2.6GHz Intel Core i7 (quad core) and 16GB RAM, thus, the P number is 4. Since we are using a shared memory architecture, the time cost of transferring data to and from other computation units $T_{i,j}^{Comm.}$ (Equation 6.13) is negligible, and the main limiting factor of speedup will be the inequality of block sampling time.

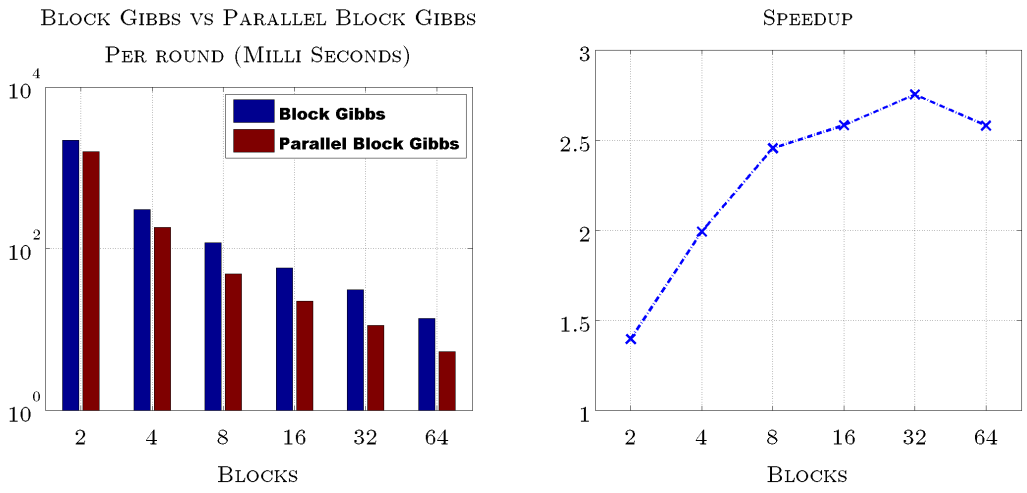
We summarize the experiment settings below:

- **Input graph** G: Students Bayesian network with 376 binary nodes (Figure 6.40)
- **Gibbs sampling duration** T: 240 seconds.
- **Number of blocks** B: 2, 4, 8, 16, 32, 64
- **Max treewidth** Tw: 14
- **Minimum block size** S_{min}: 1
- **Maximum block size** S_{max}: 200
- **Minimum samples before blocking** N_S: 100
- **Block growth method** $\mathcal{M}$: `max_correlation`
- **Computation units**: 1 (sequential block Gibbs), 4 (parallel block Gibbs)

The result is shown in Figure 6.47.

When there are 2 blocks, only 2 computation units can be used, thus the upper bound of speedup is 2×. However, due to inequality of block size and treewidth, the actual speedup is around 1.4×. When there are 4 blocks, all 4 computation units are used, so the overall speedup is better than 2 blocks, at around 2×. However, compared to $B = 8 \sim 64$ where the speedup is around 2.5×, the speedup from $B = 4$ is not significant. The reason is, when there are fewer number of blocks, the probability of generating a big block with high treewidth is higher (as evidenced in Section 6.5, in Figure 6.45a), thus the runtime of sampling one block has a bigger chance of dominating other blocks.

[18] http://docs.scala-lang.org/overviews/parallel-collections/overview.html



(a) Median Runtime **(b)** Speedup

Figure 6.47: Comparing parallel block Gibbs sampling and sequential block Gibbs sampling

Chapter 7

Conclusion

The thesis first discussed the probabilistic formulation of optical flow algorithm (POF), and proposed to use Gibbs sampling to solve the problem. Different methods to improve the POF algorithm are discussed with corresponding experimental evaluations. In this chapter we summarize our contributions, and point out future research directions.

7.1 Contributions

Optical flow methods try to find pixel movements between images in temporal sequences. We proposed a new probabilistic model of formulating optical flow (POF) using Markov random field, with three different energies to restrict flow movements. A series of experiments are conducted, comparing POF algorithm with various state-of-the-art OF algorithms, on different input shapes, movement patterns, and real world images. We showed that the POF approach is advantageous when dealing with smaller images, and is more robust when dealing with different input shapes. It also has better adaptibility when objects in the images move in different patterns. The IE result of POF method is very competitive even for larger images.

We also proposed a hybrid optical flow method POF-XOF, *i.e.*, using an optical flow algorithm's result as input to POF, to speed up POF algorihm. We showed that significant speedup can be achieved for convergence. We compared several input OF methods, and result shows that improvement varies depending on the input OF method used. Algorithms that generate similar

results to POF enable bigger speedup, while it also depends on input images.

When sampling nodes that are highly correlated, it is better to sample them jointly in a block. We proposed several heuristic measures to find highly correlated nodes, and compared with existing heuristics. We showed using our proposed heuristics, faster convergence can be achieved for sampling. We also investigated on the relationships between block size, block count, treewidth and runtime, and showed the tradeoff between block size and block count.

In a stochastic random walk, simulated annealing can be used to speed up the random walk process. In POF, we adapted simulated annealing into staged annealing. We showed that faster convergence in POF algorithm can be achieved with staged annealing.

In the end, we investigated two different methods to parallelize Gibbs sampling. We discussed theories on designing correct parallel Gibbs sampling algorithms. To parallelize sequential Gibbs sampling, chromatic Gibbs sampling method is investigated. Experiment result shows that significant speedup can be achieved. For block Gibbs sampling, we showed that as long as the blocks are not adjacent to each other, all of them can be sampled in parallel. Experiments on parallelizing block Gibbs sampling show expected speedup.

7.2 Future Work

One of the biggest concern of using the POF algorithm is the slow speed. Depending on problem domain, the better error performance of POF algorithm might be overshadowed by its slow runtime. Except for the speedup techniques we have discussed in the thesis, there are several other possible directions. For example, pyramid method is commonly used in other OF algorithms for speedup, and it can be applied to POF algorithm as well.

Also, we can extend POF to use continuous flow vectors. Currently flow vectors are discrete, thus the state space of a flow vector is discrete, and the size of the state space is quadratic to the maximum distance a pixel can travel in 2D plain. If we could sample the flow vector in a continuous space, sampling can be faster, and resulting continuous flow vectors can have better error performance.

A third direction to improve POF, is to enable better comformity among local flow vector patches. POF flow on larger images tend to be fragmented, due to limited constraining power from neighbor energies. Thus, higher level of energy function that involves bigger local area can be used for more constraint on local area flow vector conformity.

When doing block Gibbs sampling, we discussed the tradeoff between block size and block count. Currently we are fixing the block count between sampling rounds, but it can be changed to dynamically adjust its value to enable faster convergence.

When parallelizing block Gibbs sampling, the sampling of each block is sent to other computation units for parallelization. Currently we used multi-core shared memory approach, but as the size of blocks grow, it may be faster to sample the blocks in multiple GPUs. Prior work on parallelizing junction tree algorithm in GPU can be found in [50].

Bibliography

[1] Edward H Adelson and James R Bergen. Spatiotemporal energy models for the perception of motion. *JOSA A*, 2(2):284–299, 1985. 2.4

[2] Padmanabhan Anandan. A computational framework and an algorithm for the measurement of visual motion. *International Journal of Computer Vision*, 2(3):283–310, 1989. 2.4

[3] Alexander Andreopoulos and John K Tsotsos. Efficient and generalizable statistical models of shape and appearance for analysis of cardiac mri. *Medical Image Analysis*, 12(3):335–357, 2008. 3.4, 6.1.3, 6.1.7

[4] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987. 2.2

[5] Simon Baker, Daniel Scharstein, J. P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. A Database and Evaluation Methodology for Optical Flow. *International Journal of Computer Vision*, 92(1):1–31, November 2010. ISSN 0920-5691. URL http://link.springer.com/10.1007/s11263-010-0390-2. 2.4, 6.1.1, 6.1.1, 6.1.1, 6.1.3, 6.3.2

[6] J. L. Barron, D. J. Fleet, and S. S. Beauchemin. Performance of optical flow techniques. *INTERNATIONAL JOURNAL OF COMPUTER VISION*, 12:43–77, 1994. 6.1.1

[7] Umberto Bertele and Francesco Brioschi. *Nonserial dynamic programming*. Elsevier, 1972. 2.1.4

[8] Michael J Black and Paul Anandan. The robust estimation of multiple motions: Parametric and piecewise-smooth flow fields. *Computer vision and image understanding*, 63(1):75–104, 1996. 6.1.8

[9] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal*

of machine Learning research, 3:993–1022, 2003. 5.1

[10] Andrés Bruhn, Joachim Weickert, and Christoph Schnörr. Lucas/kanade meets horn/schunck: Combining local and global optic flow methods. *International Journal of Computer Vision*, 61 (3):211–231, 2005. 6.1.8

[11] Wray L Buntine. Operations for learning with graphical models. *arXiv preprint cs/9412102*, 1994. 5.1

[12] Marius Drulea and Sergiu Nedevschi. Total variation regularization of local-global optical flow. *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 318–323, October 2011. doi: 10.1109/ITSC.2011.6082986. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6082986. 3.5, 6.1.1, 6.1.8, 6.3.1

[13] Marius Drulea and Sergiu Nedevschi. Motion estimation using the correlation transform. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 22(8):3260–70, August 2013. ISSN 1941-0042. doi: 10.1109/TIP.2013.2263149. URL http://www.ncbi.nlm.nih.gov/pubmed/23686953. 3.5, 6.1.1, 6.1.8, 6.3.1

[14] Maáyan Fishelson and Dan Geiger. Optimizing exact genetic linkage computations. *Journal of Computational Biology*, 11(2-3):263–275, 2004. 2.2

[15] David Fleet and Yair Weiss. Optical flow estimation. In *Handbook of Mathematical Models in Computer Vision*, pages 237–257. Springer, 2006. 3.5

[16] David J Fleet and Allan D Jepson. Computation of component image velocity from local phase information. *International Journal of Computer Vision*, 5(1):77–104, 1990. 2.4

[17] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pages 721–741, 1984. 2.1.4, 2.3, 4.4

[18] Walter R Gilks. *Markov chain monte carlo*. Wiley Online Library, 2005. 2.3

[19] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *In Artificial Intelligence and Statistics (AISTATS)*, Ft. Lauderdale, FL, May 2011. 4.3.3, 4.3.6, 4.3.8, 4.3.8, 5.4, 5.4, 5.5, 6.5.2

[20] Berthold K Horn and Brian G Schunck. Determining optical flow. In *1981 Technical Symposium East*, pages 319–331. International Society for Optics and Photonics, 1981. 1, 2.4, 3.5, 6.1.1

[21] Claus S Jensen, Uffe Kjærulff, and Augustine Kong. Blocking gibbs sampling in very large probabilistic expert systems. *International Journal of Human-Computer Studies*, 42(6):647–666, 1995. 4.3.3

[22] C.S. Jensen and A. Kong. Blocking gibbs sampling for linkage analysis in large pedigrees with many loops. *The American Journal of Human Genetics*, 65(3):885–901, 1999. 4.3.6

[23] Finn V Jensen. *An introduction to Bayesian networks*, volume 210. UCL press London, 1996. 2.1.2

[24] Finn Verner Jensen, Kristian G Olesen, and Stig Kjaer Andersen. An algebra of bayesian belief universes for knowledge-based systems. *Networks*, 20(5):637–659, 1990. 2.1.4, 2.2

[25] Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999. 2.1.4

[26] Ross Kindermann, James Laurie Snell, et al. *Markov random fields and their applications*, volume 1. American Mathematical Society Providence, RI, 1980. 2.1.2

[27] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983. 4.4

[28] Uffe Kjærulff. Triangulation of graphs – algorithms giving small total state space. 1990. 2.2

[29] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009. 2.1.2, 2.2, 2.2, 4.3.2

[30] Steffen L Lauritzen. *Graphical models*. Oxford University Press, 1996. 2.1.2

[31] Steffen L Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 157–224, 1988. 2.1.4, 2.2

[32] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. In *IJCAI*, volume 81, pages 674–679, 1981. 1, 2.4

[33] David Mackay. *Information Theory, Inference and Learning Algorithms*, chapter 29, pages 357–371. Cambridge University Press, first edition, 2003. 2.3

[34] Ole J Mengshoel, Dan Roth, and David C Wilkins. Portfolios in stochastic local search: Efficiently computing most probable explanations in bayesian networks. *Journal of Automated Reasoning*, 46(2):103–160, 2011. 6.2

[35] Ole J Mengshoel, David C Wilkins, and Dan Roth. Initialization and restart in stochastic local search: Computing a most probable explanation in bayesian networks. *Knowledge and Data Engineering, IEEE Transactions on*, 23(2):235–247, 2011. 6.2

[36] Nicholas Metropolis and Stanislaw Ulam. The monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949. 5.3.1

[37] M. Otte and H.-H. Nagel. Optical flow estimation: Advances and comparisons. In *Proceedings of the Third European Conference on Computer Vision (Vol. 1)*, ECCV '94, pages 51–60, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc. ISBN 3-540-57956-7. URL http://dl.acm.org/citation.cfm?id=189359.189368. 6.1.1

[38] Judea Pearl. Probabilistic reasoning in intelligent systems. *Morgan Kaufmann, San Mateo, California*, 12:241–288, 1988. 2.1.2

[39] Dongzhen Piao, Prahlad G Menon, and Ole J Mengshoel. Computing probabilistic optical flow using markov random fields. In *Computational Modeling of Objects Presented in Images. Fundamentals, Methods, and Applications*, pages 241–247. Springer, 2014. 3.5

[40] Brian Ricks and Ole J Mengshoel. Methods for probabilistic fault diagnosis: An electrical power system case study. In *Annual Conference of the Prognostics and Health Management Society*, 2009. 5.1

[41] Sudeep Sarkar and Kim L Boyer. Perceptual organization in computer vision: A review and a proposal for a classificatory structure. *Systems, Man and Cybernetics, IEEE Transactions on*, 23(2):382–399, 1993. 2.4

[42] Prakash P Shenoy and Glenn Shafer. Axioms for probability and belief-function propagation. 1990. 2.1.4, 2.2

[43] Hedvig Sidenbladh, Michael J Black, and David J Fleet. Stochastic tracking of 3d human

figures using 2d image motion. In *Computer VisionâĂŤECCV 2000*, pages 702–718. Springer, 2000. 2.4

[44] Deqing Sun, Stefan Roth, and Michael J. Black. Secrets of optical flow estimation and their principles. *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2432–2439, June 2010. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5539939. 6.1.1, 6.3.1

[45] Deepak Venugopal and Vibhav Gogate. Dynamic blocking and collapsing for gibbs sampling. *arXiv preprint arXiv:1309.6870*, 2013. 4.3.3, 6.5.1

[46] Alessandro Verri, Sergio Uras, and Enrico De Micheli. Motion segmentation from optical flow. In *Alvey Vision Conference*, pages 1–6, 1989. 2.4

[47] Rene Vidal and Avinash Ravichandran. Optical flow estimation & segmentation of multiple moving dynamic textures. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 516–521. IEEE, 2005. 2.4

[48] Mutsumi Watanabe, Nobuyuki Takeda, and Kazunori Onoguchi. A moving object recognition method by optical flow analysis. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 1, pages 528–533. IEEE, 1996. 2.4

[49] Nevin Lianwen Zhang and David Poole. Exploiting causal independence in bayesian network inference. *arXiv preprint cs/9612101*, 1996. 2.1.4

[50] Lu Zheng, Ole Mengshoel, and Jike Chong. Belief propagation by message passing in junction trees: Computing each message faster using gpu parallelization. *arXiv preprint arXiv:1202.3777*, 2012. 7.2