Chimple v0.01 User Documentation

Contents

1 Installing Chimple

Chimple is automatically installed with DMPL. Refer to the documentation of DMPL to see how to install DMPL.

2 What is Chimple?

Chimple is a tool for performing inference on generative models. It is designed to foster rapid development rapid and simple code.

Features

- Supported in Java and Matlab.
- Any Matlab or Java program can be written as a Chimple program.
- Performs automated Markov chain Monte Carlo (MCMC) sampling and rejection sampling.

Future Features

- Automatic Variational Bayes inference on stochastic programs.

3 A Short Introduction to Generative Models and Stochastic Programs

A *generative model* is a probabilistic model for describing randomly generated observable data given some latent parameters. It induces an underlying distribution on the observed data. From a programmer's perspective, a generative model can be described more simply: it is a computer program, possibly invoking random number generators, which produces some output (corresponding to the observed data). Because of this second interpretation, a generative model is sometimes called a *stochastic program*.

Since a stochastic program has a random component, running it twice will (generally speaking) produce two different outputs. By running the program multiple times, we can sample from the underlying distribution on the data specified by the generative model.

In the typical situation, we wish to condition on the output (i.e. we observe the data) and wish to perform inference on the hidden variables that generated it. In some sense, we wish to "run the program backwards" to observe the distribution of the latent variables. Chimple allows us to perform this type of inference.

In Chimple, each new variable has to be explicitly and unambiguously defined, using only variables which have previously been defined. (Properly defined recursive definitions are valid, assuming they terminate.) The crucial differences between a generative model for data and a regular program are as follows:

1. **Some instructions or variable definitions are allowed to be stochastic.**

In order to do this, the Chimple language has a finite library of atomic random functions which can be combined to construct more complex random functions. These functions are called *monkeys*[1], or ERMs (for *elementary random monkeys)*.

An example of a monkey is **chimpflip**. In the absence of an argument, chimpflip returns the outcome of a random coin flip (i.e. 0 or 1 with probability 1/2 each). With a weight argument w, chimpflip returns the outcome of a biased random coin (1 with probability w, 0 with probability 1-w).

A second example of an monkey is **chimprand**, which generates a random number uniformly distributed between 0 and 1.

Here are a few examples of building generative models of data.

In this first example, we simply create a randomly biased coin, and flip it 20 times.

```
% Generates a random weight for a coin
weight=chimprand('weight');
for i=1:20
  % given an input weight, creates a coin which flips heads with
  % probability equal to weight
  X(i)=chimpflip(weight,sprintf('X',i));
end
```

In the second example, we generate two binary random variables – lung_cancer and cold – with probabilities 0.01 and 0.2 respectively, and define cough as a binary variable equal to 1 if either lung_cancer or cold is 1.

2. **At termination, the user specifies which variables or data he wants to sample**

[1] A parallelized MCMC approach to producing the works of Shakespeare might involve a thousand monkeys on a thousand typewriters

```
%defines the probability of lung cancer to be 1%
lung_cancer=chimpflip('LG',0.01);
%defines the probability of cold to be 20%
cold=chimpflip('cold',0.2);
%cough is present if cold or lung cancer is present
cough=or(cold, lung_cancer);
```

A generative model is usually run several times. Each time the program completes, Chimple records which variables are to be kept and displayed to the users (though technically, at the end of each run, a sample of all variables which were created is in memory, typically only some are of interest to the users). These are the latent variables of the system.

3. The user can specify a condition for a sample to be accepted

If run forward, a generative model for data would simply do that - generate data. The computational complexity would be exactly that of running the program. There would also be no major hurdles in probabilistic programming - simply explicitly execute each instruction until termination, and repeat as many times as specified.

In practice, however, data is not generated by a program, but given to us - measured from sensors, extracted from documents. Thus, the user needs to condition the output of the program on the observed data. This, in turn, effects the conditional distribution of the monkeys in the probabilistic program.

Conditioning on the output of the program retroactively makes most variables in the generative model implicitly defined, which in turns appears to make a sampling algorithm necessary.

Let us continue the medical diagnosis example:

```
%defines the probability of lung cancer to be 1%
lung_cancer=chimpflip('LG',0.01);
%defines the probability of cold cancer to be 20%
cold=chimpflip('cold',0.2);
%cough is present if cold or lung cancer is present
cough=or(cold, lung_cancer);
%we observe that the patient is coughing
addHardChimpConstraint(cough,1);
```

The above program represents the following model:

We assume that lung-cancer occurs in 1% of patients screened, and cold in 20% of them (independently). Presence of either disease will result in a cough in the patient.

Assume a patient arrives with a cough. We want to know if he has lung cancer. (The correct answer is that 4.8% of the samples should return true.).

We can therefore abstract the generative model process as follows:

- Generate hidden or *latent* random variables. Some of the latent variables are *structural* (i.e. they are not of direct interest to the user), while others are regular. The relationship between all the latent random variables can be arbitrarily complex.
- Specify a model for generating the *observed* random variables Y.
- Provide the program with the true values Y_0 of observed variables Y.
- Ask for samples of the regular latent variables X which best explain the observations Y. The metric of how well the data is explained is the score function P(X) P(Y|X).

In summary, stochastic programming operates as follows:

A) Write a program (with random instructions) which explains how the data was generated.

B) Given actual data, explain what was the most likely 'path' through the program which accounts for the observed data.

Mathematically, the desired probability distribution on the latent variables X is given by:

$$P(X \mid Y = Y_0) \propto P(X, Y_0) \propto P(X)\, P(Y = Y_0 \mid X)$$

This composite term is called `score'. The first part, P(X), will also be called the *trace prior* (i.e. it is the a priori probability of that particular program path). The second part, $P(Y = Y_0 \mid X)$, we will call the *data score*, or *data likelihood*. Note that while defining the data score as a conditional probability of the data given the latent variables is both convenient and intuitive, in general we will also use an arbitrary $L(Y_0, X)$ score function which the modeler will design with respect to the problem at hand.

The simplest technique for finding samples X corresponding to observations Y_0 is rejection sampling:

```
** Rejection sampling
Set Y=null
while Y~= Y_0
    generate (X,Y) according to the forward generative model
end
output X
```

Unfortunately, rejection sampling is an inefficient scheme which may take a large (and in many cases, infinite) amount of time before finding a sample.

Chimple's inference engine runs a modified version of the Metropolis-Hastings Markov chain Monte Carlo algorithm. MH-MCMC requires a proposal distribution Q(X’ | X), i.e. a sampling mechanism (with known probability transition), which, given a sample X, creates a new sample X' according to some arbitrary rule. The heart of the algorithm is illustrated by the following pseudocode:

```
** Markov Chain Monte Carlo
Start with a randomly generated X.
For t=1...T Do:
     generate X' according to Q(X'|X)
     compute
                    $\alpha = \min\left(1, \frac{P(X', Y_0)Q(X' \mid X)}{P(X, Y_0)Q(X \mid X')}\right)$

     With probability alpha, accept the change (i.e. set X=X'),
otherwise discard X'
end
output X
```

MH-MCMC asymptotically converges to a sample of the true distribution - however, there remains the question of the proposal distribution. Note that in our case, a sample X corresponds to the entire execution path of a program – i.e. the value taken by all the monkeys encountered in that path.

In general, X contains a large number of variables and one does not want to resample the entire vector X, since this would lead to a slow rate of convergence. To alleviate this problem, a good probabilistic programming language should only resample part of the vector X. However, partial resampling could lead to problems, as the resulting X' may not be a consistent sample at all (i.e. a sample that could have been generated by the model).

Consider the following program:

```
A=chimpflip(‘A’);
B1=chimpflip(‘B1’);
if A
     B2=chimpflip(‘B2’);
     B3=chimpflip(‘B3’);
     C=and(and(B1,B2),B3);
else
     B4=chimpflip(‘B4’);
     C=not(B4);
end
D= chimpflip(‘D’);
```

Suppose that we observed that C is false, and let us start an MCMC sampler with sample path X=(A=true, B1=true, b2=true, B3=false, C=false,D=true). Following the idea of a partial resample, let us suppose we create a new sample path X’ by only resampling A, making it now false, leading to a proposed new state X’=(A=false, B1=true, B2=true, B3=false, C=false,D=true). Let us note this is in fact not an admissible sample – even though C is still false. Indeed, if A is false, B2 and B3 should not exist, but B4 should.

To tackle this problem, Chimple keeps track of a computational state called the *trace* (or *sample path*). Roughly speaking, the trace can be thought of as a log of all code executed to obtain the current sample, including the values taken by each elementary random procedure (along with the likelihood of that value). To resample, we can simply consider a monkey in the trace at random. All computations leading to this point are kept as is (along with their likelihood). That particular monkey is resampled according to a kernel of choice, and all subsequent code is re-executed, but only if it needs to be, i.e. only if the line of code considered depends directly or indirectly on the resampled monkey.

Let us consider the above example again, and start with trace X=(A=true, B1=true, b2=true, B3=false, C=false). We choose a monkey at random and resample it.

- Suppose A is resampled to *false*. B1 is not resampled because it does not depend on A (even though B1 is created after A). B2 and B3 are discarded since they corresponded to an unused branch of computation. B4 is sampled (say to value *true*), and C is recomputed to *false* (incidentally equal to its previous value). D is kept to the same value since it is not affected by A.

- Suppose B3 is resampled to *true*. A, B1 and B2 keep their previous values, and C is recomputed to true. D is kept to the same value since it is not affected by B3.
- Similarly, if B2 is resampled to false, all other variables keep their previous values and the computation terminates. D is kept to the same value since it is not affected by B2.
- If D is chosen, it is resampled to false and all other variables keep their current values.
- Note that C is always *indirectly* recomputed - it is not a monkey (it is however random, as a function of other random variables), so it is never chosen directly by the algorithm to be resampled.

The details of how Chimple keeps track of which variables need to be recomputed, and how to compute the acceptance-rejection ratio for this particular method of proposal, are beyond the scope of this report. Let us finish by mentioning that a variety of other search methods can also be used to find samples, for instance A*, or sequential sampling.

4 Chimple User Tutorial

This tutorial describes how to code using MATLAB™ Chimple. We provide four tutorials.

The first tutorial revisits the code for the medical problem and for flipping random coins.

4.1 *Defining Elementary Random Monkeys and writing the Stochastic Program*

A Chimple program consists of two parts: a stochastic program, which generates the observed data, and a meta-program, which governs the MCMC sampling of the stochastic program.

The conditioning expression, or data cost, can be included in both the stochastic program, and the meta-program.

4.1.1 Medical example: chimpflip, naming monkeys, conditioning statements

Recall the following stochastic program:

```
function [lung_cancer]=medical_BN()

%defines the probability of lung cancer to be 1%
lung_cancer=chimpflip('LG',0.01);
%defines the probability of cold cancer to be 20%
cold=chimpflip('cold',0.2);
%cough is present if cold or lung cancer is present
cough=or(cold, lung_cancer);
%we observe that the patient is coughing
addHardChimpConstraint(cough,1);
```

The program starts by defining the ‘lung_cancer’ monkey as a binomial (or ‘flip’) variable, equal to 1 with probability 1%. Note that the monkey takes two parameters: the first parameter of any monkey is its *name*. The name of an monkey is simply a unique `internal’ identifier for each monkey. Since stochastic program (even complex, recursive ones) correspond to directed probabilistic models, the requirement that each monkey can be given a unique name is always possible. In that sense, each monkey may have two names. The first is the internal name of the monkey, which is always the first argument of the monkey call (in this case, ‘LG’). The second, which is optional, is the name of the value the monkey actually takes, in this case, ‘lung_cancer’. A future version of Chimple will remove the need from the user to specify a unique internal name – as it will be derived automatically from the program at execution time. While using the same name for the internal and the Matlab name of the variable is good Chimple programming practice, Chimple does not require both names to be identical (as evidenced by the lung_cancer variable).

In short, an monkey call will *always* take the following form

```
Matlab_name=erm_name('internal_name',parameters);
```

where Matlab_name is optional (in case the monkey is used directly in a computation), and parameters is optional as well (depending on the monkey).

In the case of chimpflip, there is only parameter, the *weight* of the `coin’. If not specified, it defaults to 0.5.

The output of a stochastic programs consists of the variables which will be kept (in other words, the sampled variables). While all variables could always be included as part of a global “trace” output for a stochastic program, if the user only cares about the values taken by certain variables, he can safely exclude other “structural” variables. In this case, the user only cares about the occurrence of lung cancer or not.

In order to run our stochastic program, we always need to write a very simple “meta-program”, which specifies the sampling parameters (and potential conditioning statements).

```
burnin = 0;
samples = 300;
spacing = 10;
results = chimplify(@medical_BN,burnin,samples,spacing);
```

Burnin specifies how many samples are used for the MCMC ‘burn-in’ (i.e. how many MCMC iterations are run without keeping results). Samples specifies the number of required samples. Spacing specifies how many iterations are run between each conserved sample.

The last command, ‘chimplify’, runs the stochastic program. It has the following syntax:

```
results=chimplify(@program_name,burnin,samples,spacing,arguments,condition,carg)
```

where condition may be unspecified; results will always be a cell array of the outputs of the stochastic program (in this case, it will be a cell array of length 300, each cell containing one value). Arguments is a cell array of arguments which will be passed to the program ‘program_name'. condition is a cost function which takes the output of the program as input, and returns the negative log of a data likelihood function. Carg is an optional cell array of supplemental arguments to the condition function.

Let us detail how to use arguments and conditioning statements (or likelihood functions), beginning with conditioning statements.

Conditioning statements – or score functions, as they are generally defined in Chimple – can be declared by the user in two different ways:

- The user is allowed to either specify a cost function directly inside the stochastic program. This is done with the command “`addChimpCost`”. `addChimpCost` is called with a single real argument. At the end of the execution of a sample stochastic program, all values which were given to arguments to addChimpCost are added to the overall trace likelihood. We also provide two special cases of `addChimpCost:`
    - The first, `addHardChimpConstraint`, takes two arguments and enforces hard equality of both arguments.
    - The second, `addSoftChimpConstraint`, takes two arguments and an optional third temperature argument, and enforces soft equality of both arguments

- In many cases, declaring a constraint inside the stochastic program can be seen as undesirable from a UI point of view. Chimple allows the user to declare a cost function as part of the arguments of chimplify. In Chimple, a cost function should always be declared to be a function of the outputs of the stochastic program (even if those outputs are not used). A cost function should not declare any additional monkeys and should consist only of : computation of the desired score function (regular Matlab code), and sending those negative-log score functions to Chimple. This is done using the regular addChimpCost function. However, recall that in the scope of the cost function, only the values of the variables output as a result of the stochastic program will be accessible. The names of those output variables won’t be accessible, and neither the name nor the value of non-output variables will be accessible. We therefore provide a method for accessing a variable (monkey or not) outside of the scope of the stochastic program. To access any value outside of the stochastic program, we simply use the `chimpConst’ method. chimpConst requires syntax similar to an monkey:

```
chimpConst('access_name', value);
```

Just like for an monkey, the access_name must be unique and used once - it cannot be used for another monkey – however, ALL monkeyS are accessible outside of the stochastic program by their unique name - they do not need to be stored under a chimpConst.

Then, the value stored under `access_name’ can be retrieved in the cost function using the command

```
Value=getChimpValue('erm_access_name');
```

Armed with the information above, we show how to rewrite the example with the cost function external to the stochastic program.

```
function [lung_cancer]=medical_BN()

%defines the probability of lung cancer to be 1%
lung_cancer=chimpflip('LG',0.01);
%defines the probability of cold cancer to be 20%
cold=chimpflip('cold',0.2);
%cough is present if cold or lung cancer is present
cough=or(cold, lung_cancer);


chimpConst('cough',cough);
```

```
function [out]=costfunction(lg_cancer,coughvalue)

% first argument is required because it is the output of medical_BN.m
% second argument lets the user choose the actual value cough as a
% parameter

addHardChimpConstraint(getChimpValue('cold'),coughvalue);
```

```
burnin = 10;
samples = 10000;
spacing = 10;

% did we observe cough?
cough_value=1;

results = chimplify(@medical_BN,burnin,samples,spacing,
{},@costfunction,cough_value);
```

4.1.2 Coin-flip example

```
function [out]=randomcoin()
```

```
% Generates a random weight for a coin
weight=chimprand('weight');
for i=1:20
  % Given an input weight, creates a coin which flips heads with
  % probability equal to weight
  X(i)=chimpflip(sprintf('X%d',i),weight);
end
out=weight;
sumout=sum(X);
chimpconst('sumvar',sumout);
```

```
function [out]=costfunction(weight,sumvalue)
% first argument is required because it is the output of medical_BN.m
% second arugment lets the user choose the actual value cough as a
% parameter
addSoftChimpConstraint(getChimpValue('sumvar'),sumvalue,0.01);
```

```
burnin = 100;
samples = 200;
spacing = 10;
sum_val=15;
results = chimplify(@randomcoin,burnin,samples,spacing,{},@costfunction,
{sum_val});
res = cell2mat(results);
hist(res);
```

4.2 Monkey dictionary and creating new monkeys

4.2.1 Current dictionary of monkeys

1. chimpbeta(alpha,beta,sigma) : creates a beta random variable with parameters (alpha,beta). The optional parameter sigma is used as a parameter for the kernel proposal of the monkey
2. chimpdirichlet(alphas): create a dirichlet random variable with priors specified by the vector alphas.

3. chimpdiscrete(probabilities,sampled_set) samples an element of the set sampled_set with probabilities specified by the first vector. The sampled_set is optional and defaulted to the vector counting from 1 to the length of the probability vector.
4. chimpflip(weight) flips a coin with returns 1 with probability specified by the parameter weight (optional, defaulted to 0.5) .
5. chimpperm(n) returns a random permutation over n elements.
6. chimprand returns a random number between 0 and 1.
7. chimprandn(mu,sigma,sigma_f) returns a gaussian distributed random variable (mean mu, std dev sigma). sigma_f is optional and is used for the proposal kernel. Mean and standard deviation are also both optional, defaulted to 0 and 1 respectively.

4.2.2 Creating a new monkey

While Chimple's main monkeys are coded in Java (and the dictionary will keep expanding), it is possible to code one's own monkey by following a simple procedure:

1. In the elementary_monkey directory, create a new directory with an arbitrary name.
2. The directory requires 4 files (see template for a template and example for an example)
    a. An monkey file, which will call ERM.m
    b. A base generator, erm_gen.m
    c. A likelihood computing function, erm_likelihood.m
    d. A kernel proposal function, erm_kernel.m
3. The main purpose of ERM.m is to pack all the arguments into a cell array and make the following call:
   ```
   runMonkey(@erm_gen,@erm_regen,@chimptemplate_kernel,@chimptemplate_like
   lihood,name,params);
   ```
4. erm_gen must take as input the list of all parameters (including irrelevant ones, like those used for resampling) of the monkey, and out a randomly sampled variable
5. erm_kernel must take as input a valid value of the monkey, followed by the list of all parameters, and returns both a new value, and the negative log of the hastings term (see the template for the definition of the hasting terms)
6. erm_likelihood must take a value value of the monkey, followed by the list of all parameters, and return the negative log of the likelihood of that value.