THE UNIVERSITY *of* EDINBURGH

Modular probabilistic programming with algebraic effects

Oliver Goldstein

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2019

Abstract

Probabilistic programming languages, which exist in abundance, are languages that allow users to calculate probability distributions defined by probabilistic programs, by using inference algorithms. However, the underlying inference algorithms are not implemented in a modular fashion, though, the algorithms are presented as a composition of other inference components. This discordance between the theory and the practice of Bayesian machine learning, means that reasoning about the correctness of probabilistic programs is more difficult, and composing inference algorithms together in code may not necessarily produce correct compound inference algorithms. In this dissertation, I create a modular probabilistic programming library, already a nice property as its not a standalone language, called Koka Bayes, that is based off of both the modular design of Monad Bayes – a probabilistic programming library developed in Haskell – and its semantic validation. The library is embedded in a recently created programming language, Koka, that supports algebraic effect handlers and expressive effect types – novel programming abstractions that support modular programming. Effects are generalizations of computational side-effects, and it turns out that fundamental operations in probabilistic programming such as probabilistic choice and conditioning are instances of effects.

Table of Contents

1 Introduction **1**

2 Background **4**

2.1 Probabilistic programming 4

2.1.1 Generative models 5

2.1.2 Inference 6

2.2 Algebraic Effects 7

3 Modular inference: importance sampling **10**

3.1 Implementation 10

3.2 Models as factor graphs 10

3.3 Importance Sampling 11

3.4 Importance Sampling implementation 12

3.4.1 Population 12

3.5 Use case : First order Markov Chain 13

3.6 Caveats 13

4 Inference algorithms **15**

4.1 Resampling 15

4.2 Sequential Monte Carlo 16

4.2.1 Sequential Monte Carlo implementation 17

4.3 Trace Markov Chain Monte Carlo 20

4.3.1 MCMC 20

4.3.2 Program traces and the trace data structure 21

4.3.3 Trace Markov Chain Monte Carlo implementation 22

4.4 Resample Move Sequential Monte Carlo 24

4.4.1 Resample Move Monte Carlo implementation 24

4.5 Particle Marginal Metropolis Hastings 25

4.5.1 Particle Marginal Metropolis Hastings implementation 25
4.6 Evaluation 26
4.6.1 Quantitative Evaluation 26
4.6.2 Qualititative Evaluation 27

5 Use case : climate change modelling **30**
5.1 The Kalman Filter 30
5.1.1 Extended linear Gaussian state-space model 31
5.2 Experiments 32
5.2.1 Berkeley Earth Dataset 32
5.2.2 Inference hyper-parameters 34
5.3 Results 34
5.4 Evaluation and future work 35

6 Conclusions **39**
6.1 Summary 39
6.2 Future / further work 40

Bibliography **41**

A Additional figures for SMC / TMCMC based inference over the extended linear Gaussian climate model. **47**

Chapter 1

Introduction

Context

The Bayesian approach to probabilistic reasoning is known as Bayesian Inference, and is an increasingly popular method to perform machine learning that enjoys strong mathematical foundations[1]. Bayesian Inference has useful properties, compared to deep learning based machine learning, such as interpretability, explainability and expressiveness with respect to various forms of uncertainty [1, 2]. At the heart of the Bayesian paradigm is the notion of a generative model. Generative models can account for the expressiveness of uncertainty and interpretability. In order to perform probabilistic modelling, knowledge about a domain and the uncertainty associated with that knowledge is encoded into a generative model, using expert knowledge. Inference algorithms estimate parameters of the model, that generate the observed data, by conditioning models on associated observed data. For instance, a climate researcher may create a basic model of a city when studying urban heat islands. The climate researcher would condition the model of the city with empirical data, such as the observed temperatures of the city, in order to infer values of interest such as latent city temperatures or factors that influence the temperature of a city. Probabilistic programming languages provide a generic framework to perform inference.

To highlight the importance of probabilistic programming languages, large multinational technology companies such as Microsoft, Google and Uber have recently started to invest in their own probabilistic programming languages (Infer.NET [3], Ed-

[1]Section 1.1, 1.2, 2.1 (intro) and 2.1.1, are loosely based on – but not directly copied, from my IRR/IPP.

ward [4] and Pyro [5] respectively). Increasingly, machine learning use cases find their way into areas as diverse as critical defense national infrastructure [6][7] or medical diagnostics [8]. Society is therefore increasingly in a position that relies on programmers to produce correct machine learning related code. As probabilistic programming becomes more important, the code that underlies probabilistic programming languages will face increasing scrutiny. Fortunately, there are two important programming language research goals which aim to ensure the correctness of code; modularity and type-safety. In particular, they ensure that programmers can be certain early on – at compile time – that composing code will not cause programs to crash or introduce unintended (inference) algorithmic bias. These research goals have two effects; as programming libraries become safer and more modular, machine learning models will become cleaner, simpler and easier to maintain, which will additionally make machine learning more trusted, and perhaps more widely adopted.

Algebraic effects are a recently discovered abstraction which enable modular programming [9]. However, algebraic effects do not use monads [10]; an abstraction derived from a branch of mathematics known as category theory. Monads are known to be both unintuitive as well as not being closed under composition. Monad Bayes is a library for modular probabilistic programming published only last year, that makes essential use of types. However, whilst it is based off a denotational validation [11], it suffers from the issues associated with monads [12]. These issues motivate a previously unachieved construction, of a modular probabilistic programming language, that also uses this same denotational validation, to perform Bayesian inference using algebraic effects with expressive effect-types.

Problem Statement

Inference algorithms lie at the heart of probabilistic programming languages and are informally described as the combination of primitive blocks, for instance, Resample-Move Sequential Monte Carlo (RMSMC) [13] is described as Sequential Monte Carlo [14] (SMC) combined with one or more Markov Chain Monte Carlo steps (MCMC) [15]. Whilst probabilistic programming languages have existed since 1994 [16], most implementations do not program inference algorithms as compositions of other inference primitives [17]. This poses a question which this project aims to answer: can probabilistic programming languages be created that build correctness and compos-

ability into their core? Monad Bayes [17], whilst modular, uses monads which have the previously stated problems with composability and intuitiveness. Currently, there is no probabilistic programming language, that demonstrates the use of algebraic effect handlers to produce correct, modular and composable inference. The demonstration of a language is important for several reasons. Firstly probabilistic programming is a new use case for Daan Leijen's language, Koka. Secondly, it affirms the theoretical link of the equivalence of monads and effects. Thirdly it provides industry with a modular approach to perform probabilistic programming.

Proposed solution and contribution

I create a probabilistic programming library, Koka Bayes, with support for four inference algorithms, embedded in Koka, a contemporary functional language that supports algebraic effects. I construct the library iteratively, with the construction split into three vertical slices, proceeding in the order listed here:

1. Library with a basic inference strategy, demonstrated over a basic model.
2. Addition of four more inference algorithms to the library.
3. Demonstration of inference, over a more complex model, conditioned on real world data.

Koka Bayes is based off Monad Bayes's design and its semantic validation, and will be compared to it on scalability and execution time. I test and evaluate Koka Bayes with Monte Carlo inference, over a linear Gaussian model of climate.

Structure of dissertation

Chapter 2 introduces the concepts that underpin probabilistic programming and effect handlers. Chapter 3 introduces and evaluates the theory and implementation of importance sampling – a basic inference strategy. Chapter 4 presents four well known, and more advanced inference algorithms, which are motivated by the weaknesses of importance sampling. Chapter 5 demonstrates two of these inference algorithms over a model of climate change. Chapter 6 concludes with a summary of the contributions made and suggests future work.

Chapter 2

Background

2.1 Probabilistic programming

Probabilistic programming languages support random sampling of variables, as well as the conditioning of programs as a function of both observed data and the sampled values of the random variables [18]. The goal of writing probabilistic programs is to perform inference over the probabilistic program in order to obtain the *posterior* distribution, $p(\theta|D)$, specified by the program. The posterior distribution is a probability distribution over parameters θ, conditioned on observed data D.

$$p(\theta|D) = \frac{p(\theta,D)}{p(D)} = \frac{p(D|\theta)p(\theta)}{p(D)} = \frac{p(D|\theta)p(\theta)}{\int p(D,\theta)\,\mathrm{d}\theta} = \frac{p(D|\theta)p(\theta)}{\int p(D|\theta)p(\theta)\,\mathrm{d}\theta} \tag{2.1}$$

Using programs to represent probabilistic models was conceived as an idea in 1978 [19] and programming languages for Bayesian modelling and inference have existed since at least 1994 [16]. Probabilistic programming languages have been used for various tasks from reasoning about theory of mind, to matching players who have similar skill levels on Xbox [20, 21][1]. There are two distinct kinds of probabilistic programming languages, both of which span a wide spectra of use cases[2]:

- Languages such as ProbLog 2 [22] perform inference over logic programs, often called worlds, and concern themselves with computing the probability of a truth-related query about the world, given evidence or data.
- Languages such as Edward [4] or Tabular [23], infer, given data, the probability distribution of parameters that generated the given data.

[1] http://forestdb.org/

[2] http://probabilistic-programming.org/wiki/Home

Probabilistic programming aims to internalize general purpose inference mechanisms inside the compiler in order to allow the programmers to focus on writing models and have inference automatically done for them [18]. This specific goal has motivated the construction of many languages such as Gen [24]. However, Ackermann & Freer [25] show that, in general, computing continuous conditional probability distributions – such as the posterior – cannot be automated, as the halting problem can be reduced to the computation [25].

2.1.1 Generative models

Probabilistic programs represent Bayesian models. A Bayesian model is the combination of a statistical model with a prior distribution [26]. A statistical model is a set of random variables $\mathbf{x}_\theta$ parameterized by θ. For each value of θ, x_θ is a random variable with probability density function $p(x|\theta)$. Data is then used to pick out a particular member of the family $\{p(x|\theta)\}_{\theta\in\Theta}$. This process is called estimating or learning the parameters of a statistical model. Once a particular θ is chosen, a probabilistic model results. Probabilistic programs often do inference by conditioning Bayesian models on whether parameter samples drawn from the prior produce data equal to the observed data.

An example of a statistical model is a Bernoulli distribution $p(x|\theta) = \theta^x(1-\theta)^{1-x}$. Combining this statistical model with a probability distribution over θ results in a Bayesian model. An example of such a probability distribution is the beta distribution:

$$p(\theta) = p(\theta|\alpha_0,\beta_0) = \frac{1}{Z(\alpha_0,\beta_0)}\theta^{\alpha_0-1}(1-\theta)^{\beta_0-1}$$

Multiplying this prior with the statistical model results in the following Bayesian model:

$$p(x,\theta|\alpha,\beta) = \frac{1}{Z(\alpha,\beta)}\theta^{x+\alpha+1}(1-\theta)^{\beta-x}$$

.

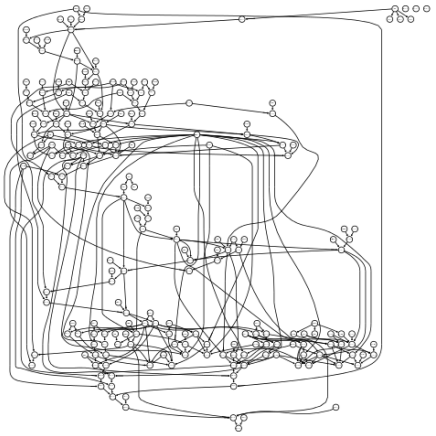


Figure 2.1: Bayesian network from [27]

Probabilistic programming provides those familiar with generative models, three important benefits:

- As the complexity of models increases, the mathematical language that represents them starts to matter. For example, other mathematical languages such

as graphical models can be challenging to understand visually (see Figure 2.1), especially when models use complicated control flows.

- Probabilistic programming separates inference from the modelling. This allows modellers to focus on modelling rather than the complex process of getting inference algorithms to work for their particular models.

- Many formalisms such as factor graphs or graphical models do not indicate the type of parameters to factor nodes or the distribution that underlie variable nodes. Probabilistic programs can express these types, which is useful information for modellers and inference algorithms.

2.1.2 Inference

There are three main forms of inference used by probabilistic programming languages; exact, Monte Carlo, and variational.

- Exact inference algorithms calculate the posterior exactly, which means computing every term of the denominator in equation 2.1 (which corresponds to a sum in the discrete case).

- Monte Carlo inference algorithms approximate conditional probability distributions with samples generated from proposal distributions. Monte Carlo algorithms are general and guarantee asymptotically exact convergence. A prototypical example of a Monte Carlo inference algorithm, that I present in more detail in Chapter 4, is Sequential Monte Carlo (SMC) [14].

- Variational inference algorithms approximate conditional probability distributions with functions chosen from a parametric class. These functions are then optimized to match the conditional probability distribution of interest by minimizing statistics that indicate the difference between two distributions – typically the KL-divergence. As the parametric class may not in general include the true posterior distribution, variational methods are not guaranteed to be asymptotically exact.

2.2 Algebraic Effects

Plotkin and Pretnar's [9] algebraic effects and handlers, are a language feature that generalize commonly occurring control flow constructs e.g. mutable state, non-determinism, exceptions, interactive input-output etc. Koka is a functional language that places algebraic effects at the core of its design. Koka also uses type signatures that not only describe the input and output value of a function, but also *effect annotations* that indicate the side-effects that come from behaviours that a function may perform.

Listings 2.1 to 2.4 give simple examples of programming with effects. In Listing 2.1, `add` is well defined for all values of `x` and `y`, and does not cause any effects, so the Koka type system allows its effect annotation to be `total`. By adding in a print statement, Listing 2.3 may no longer be marked as total, and must be annotated with the `console` effect. Koka supports polymorphic effect types. In Listing 2.4, the presence of _e in the type signature means there is *some* effect that can be unified with *_e*, in addition to the already existing `total` effect type. In this case, *_e* will be unified with `console`. Specific effect operations can also be declared with the `effect` keyword. Listing 2.2 details three effects used by Koka Bayes. The `sample` effect takes the empty type and returns a double, the `score` effect takes an `exp` data type and returns the unit type and the `yield` effect takes and returns the unit type. Adding the `sample` operation to a computation, requires its computation to include sample. In Koka, an effect type indicates that the computation may incur the particular effect, though it does not have to incur this effect. However, if the effect type is not indicated in the signature then the computation will not incur the effect. Precise inference of the exact effect type is in general undecidable since Koka is Turing complete (consider a loop that may not terminate or cause an exception).

User defined effect operations cause a runtime exception unless the program provides an appropriate effect handler. Handlers provide the semantics for computations built out of effect operations [28]. In a computation, the presence of an effect acts as a signal, which propagates outwards until it reaches an effect handler of the computation with a matching clause [29]. The typing judgments for deep and shallow handlers [10] In Figure 2.2 and Figure 2.3 provide further insight on the workings of effect handlers. Handlers are specified via sets of mappings, which consist of *return clauses* or *effect operations clauses*. For each effect operation, both types of handler specify a contin-

uation. A continuation represents the remaining computation which is currently being handled. The only difference between the types of deep and shallow handlers is the type of the continuation k. The continuations are similar in that they take the effect parameter p with type B_i. However, the effects under which the continuation proceeds along with the return types of the continuations differ:

- In the deep case, the effect operations E are handled by the handler, leaving a new effect E'. Handlers in Koka, default to deep handlers.
- In the shallow case, because the continuation returns an un-handled computation, the continuation indicates that the effect E is not discharged.

Listing 2.1: Total addition function

```
1 fun add(x : int, y : int)
2     : total int
3     { return x + y }
```

Listing 2.2: Effect constructors

```
1 effect control yield() : ()
2 effect control sample() : double
3 effect control score( s : exp ):()
```

Listing 2.3: Addition with print

```
1 fun add(x : int, y : int)
2 : <total,console> int {
3     println("Hello World!")
4     return x + y }
```

Listing 2.4: Polymorphic addition function

```
1 fun add(x : int, y : int)
2 : <total|_e> int {
3     println("Hello World!")
4     return x + y }
```

FIGURE 2.2

DEEP HANDLER TYPING RULES

$$E = \{\mathrm{op}_i : A_i \to B_i\}_i$$
$$H = \{\mathbf{return}\, x \mapsto M\} \uplus \{\mathrm{op}_i\, p\, k \mapsto N_i\}_i$$
$$[\Gamma,\ p : A_i,\ k : B_i \to_{E'} C \vdash_{E'} N_i : C]_i$$
$$\Gamma,\ x : A \vdash_{E'} M : C$$

$$\Gamma \vdash H : A\ ^{E} \Rightarrow^{E'} C$$

FIGURE 2.3

SHALLOW HANDLER TYPING RULES

$$E = \{\mathrm{op}_i : A_i \to B_i\}_i$$
$$H = \{\mathbf{return}\, x \mapsto M\} \uplus \{\mathrm{op}_i\, p\, k \mapsto N_i\}_i$$
$$[\Gamma,\ p : A_i,\ k : B_i \to_{E} A \vdash_{E'} N_i : C]_i$$
$$\Gamma,\ x : A \vdash_{E'} M : C$$

$$\Gamma \vdash H : A\ ^{E} \Rightarrow^{E'} C$$

Listing 2.5: Weighted effect handler

```
1  fun weighted(w : exp,
2  action : () -> <score|e> a)
3               : e (exp, a) {
4    var w_t := w
5      handle(action) {
6        return x -> (w_t, x)
7        score(s) -> {
8        w_t := mult_exp(w_t, s)
9        resume(())
10 }}}
```

Listing 2.6: Random Sampler effect handler

```
1 fun random_sampler(action){
2   handle(action) {
3     sample() -> {
4     resume(random())
5 }}}
```

The `weighted` handler in Listing 2.5 handles score effects. The handler (colored in **olive**), is initially called with `wp`. Line 3 creates a reference to this weight, so its value can be updated as successive score effects are encountered. Line 4 handles the 'action' using the `handle` keyword. The computation takes the unit type and returns an *a* with with effect annotation <**score**|e>, i.e. it contains score effects (effects are colored in **magenta**) and any other effects *e*. As score effects are encountered, this handler multiplies the parameter to each score effect with the previously stored score, which, at the point in the program when the first score statement is encountered, will be the initial weight `wp`. This iterative multiplication progressively calculates and then assigns, a real valued score to a computation. After each handled score effect, the result of each multiplication is stored as the new score. The program is then resumed with the unit type. Return values are returned in a pair along with the stored weight. The `random_sampler` handler in Listing 2.6 handles sample effects by resuming computations with random doubles, thus converting sample effects into Koka's built in ndet effects.

Algebraic effects have been used to perform probabilistic programming before in Pyro's Poutine library [5]. Moore and Gorinova et al. [12, 30] realize that interceptors in Edward [4] and other operations such as Tracing, Conditioning or accumulating the log joint density of a program, are accidental implementations of effect handlers. However, none of the works use effect types and their inference algorithms are not based off of a denotational semantics and thus not necessarily correct.

Chapter 3

Modular inference: importance sampling

3.1 Implementation

The source code of Koka Bayes is on Github[1] and was based off the design of Eff-bayes[2]. However, there are two important differences between Koka Bayes and Eff-Bayes. Firstly, Eff-Bayes is in a different language that doesn't have effect annotations. Secondly, Eff-Bayes only implements Importance Sampling & SMC. I created Unix/Linux based `make` commands to ensure straightforward installation and running. Importance Sampling is a simple inference algorithm which underlies three compound inference algorithms, and therefore is introduced in this Chapter.

3.2 Models as factor graphs

In the context of Koka Bayes, the input to inference algorithms are models. In Koka Bayes, following the design of Monad Bayes, a model is defined, as a *thunk* or delayed computation that returns a value x of polymorphic type a accompanied by at least score and sample effects:

- The `sample` effect, when handled by the `random_sampler` handler, equips models with an operation that returns a uniformly distributed probability floating point value from the unit interval, expressing probabilistic non-determinism thus the `ndet` effect.

[1] https://github.com/theneuroticnothing/koka-bayes
[2] https://github.com/ohad/eff-bayes

- The score effect, when handled by the `weighted` handler, provides a mechanism to successively multiply the arguments to score effects together. The weighted handler calculates the likelihood of model samples, given observed data.

3.3 Importance Sampling

Importance sampling approximates integrals with sample averages. Statistics of the posterior such as its expectation, or the partition function of a Bayesian model (denominator of Equation 2.1). For example, the partition function can be computed as follows:

$$Z(\theta) = \int \tilde{p}(x;\theta)\,\mathrm{d}x = \int \frac{\tilde{p}(x;\theta)}{q(x)} q(x) \approx \frac{1}{n}\sum_{i=1}^{n} \frac{\tilde{p}(x;\theta)}{q(x)}\,\mathrm{d}x$$

$$\mathbb{E}_p[x] = \int x p(x)\,\mathrm{d}x \approx \frac{1}{n}\sum_{i=1}^{n} x_i$$

In general:

$$\mathbb{E}_p[g(x)] = \int g(x)p(x)\,\mathrm{d}x \approx \frac{1}{n}\sum_{i=1}^{n} g(x_i)$$

This generalization allows integrals to be approximated as follows:

$$I = \int g(x)p(x)\,\mathrm{d}x = \frac{\int g(x)\tilde{p}(x)\,\mathrm{d}x}{\int \tilde{p}(x)\,\mathrm{d}x} = \frac{\int g(x)\frac{\tilde{p}(x)}{q(x)}q(x)\,\mathrm{d}x}{\int \frac{\tilde{p}(x)}{q(x)}q(x)\,\mathrm{d}x}$$

$$= \frac{\mathbb{E}_q[g(x)\frac{\tilde{p}(x)}{q(x)}]}{\mathbb{E}_q[\frac{\tilde{p}(x)}{q(x)}]} = \frac{\mathbb{E}_q[g(x)\frac{\tilde{p}(x)}{\tilde{q}(x)}]}{\mathbb{E}_q[\frac{\tilde{p}(x)}{\tilde{q}(x)}]} \approx \frac{\frac{1}{n}\sum_{i=1}^{n} g(x_i)\frac{\tilde{p}(x_i)}{\tilde{q}(x_i)}}{\frac{1}{n}\sum_{i=1}^{n}\frac{\tilde{p}(x_i)}{\tilde{q}(x_i)}} = \frac{\sum_{i=1}^{n} g(x_i)w_i}{\sum_{i=1}^{n} w_i}$$

where $q(x) = 0 \implies g(x) = 0$ and the importance weights (w_i) and the normalized importance weights (*niw*) are:

$$w_i = \frac{\tilde{p}(x_i)}{\tilde{q}(x_i)} \qquad niw = \frac{w_i}{\sum_{i=1}^{n} w_i}$$

In this formulation of importance sampling, neither the prior $\tilde{q}(x)$ nor the likelihood function $\tilde{p}(x)$ needs to be normalized. w_i is referred to as an importance weight.

3.4 Importance Sampling implementation

Importance sampling uses two steps. The first step constructs a representation of the posterior. This step uses both the `score` and `sample` effects, handled by two distinct handlers; `weighted` and `random_sampler`, respectively. The second step normalizes the importance weights.

3.4.1 Population

Listing 3.3 implements Importance Sampling as a function parameterized by a model, that returns a histogram approximating the posterior. We implement histograms by a list of pairs of values corresponding with their weights. Values may appear multiple times in the histogram. The `populate` function in Listing 3.1 constructs the histogram, by repeatedly executing the model, following the design of Monad Bayes. Given the model terminates without causing an exception, each execution encounters a return statement, whereby the handled model returns a pair of, an *importance weight* (via the weighted handler) and a returned model sample, which are added to the histogram. These two-tuples are referred to as particles [14]. The `weighted` handler constructs the score progressively as the running of the model encounters score statements. The weights that parameterize the weighted handler (importance weights) are initialized uniformly to $1/k$, where k is the number of particles. Once the entire histogram has been filled, the histogram is normalized such that the weights sum to one.

Listing 3.1: Populate

```
1 fun populate(k : int,
2 model : () -> <score|e> b) :
    e histogram<b> {
3   list(1, k) fun(i) {
4     weighted(Exp(0.0)) {
5       score(div_exp(Exp(0.0),
6       Exp(log(k.double)))) //
            score 1/k
7       model()
8 }}}
```

Listing 3.2: First Order Markov Chain

```
1  val g = fun() {
2      var x       := 0.0
3      val vrs      = 1.0
4      val scor-vrs = 0.2
5      for(0, 5) fun(i) {
6        x := normal(x, vrs)
7        score(
8        normal_pdf(x, scor-
             vrs, 3.0)
9        )}
10     x}
11 g
```

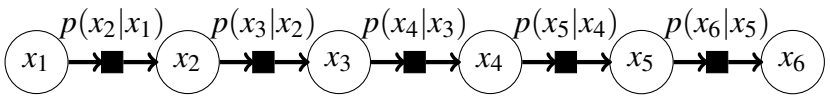


Figure 3.1: Factor graph of first order Markov Chain.

Listing 3.3: Importance Sampling with 2000 particles

```
1 fun importance_sampling(model : model<a,e>) : <ndet|e> histogram<a> {
2   random_sampler{normalise(populate(2000, model))}}
```

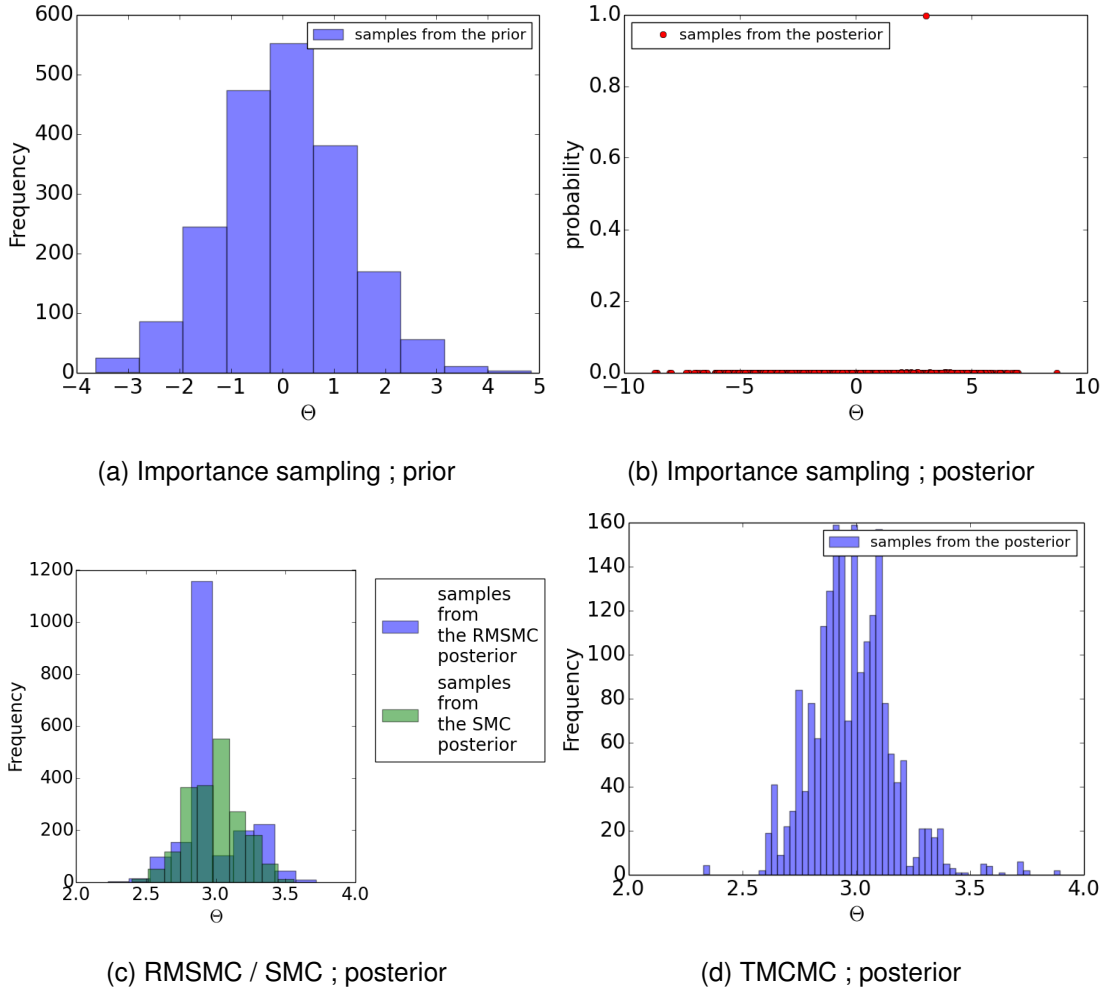
Given a model which only samples from the proposal density (prior distribution), then creating the histogram, handled with the `weighted` and `random_sampler` handlers, corresponds to *importance sampling*. However, models that sample variables of interest, in a sequential fashion, such that sample statements depend on previous sample statements via a transition probability distribution, parameterized by the value of the previously sampled value from the proposal density i.e. $p(x_{t-1}|x_t)$, then, handling such a model corresponds to Sequential Importance Sampling [31].

3.5 Use case : First order Markov Chain

Figure 3.1 demonstrates Importance Sampling on a Gaussian first order Markov Chain. The code corresponding to this model can be seen in Listing 3.2. In this chain, all transition distributions are Gaussian. At each variable node, the sampled random value is scored with respect to a Gaussian function with the data acting as the mean – modelling corruptions of the observed values by Gaussian noise. The parameters that account for the standard deviation of both the corruption and the transition distributions are chosen arbitrarily at constant values of 0.2 and 1.0 respectively.

3.6 Caveats

Importance Sampling suffers from sensitivity to the choice of the proposal distribution. If the proposal distribution (the prior) assigns low probability mass in regions of high probability associated with the posterior distribution then few samples will be drawn from the posterior. This is reflected in Figure 3.2c whereby the returned histogram returns samples from the Gaussian prior $\mathcal{N}(0,1)$. Examining the likelihood of each particle reveals the flaw of importance sampling. Figure 3.2d shows that most particles have zero probability and only a single particle from the prior, which was sampled



(a) Importance sampling ; prior

(b) Importance sampling ; posterior

(c) RMSMC / SMC ; posterior

(d) TMCMC ; posterior

Figure 3.2: Results for inference over the first order Markov Chain model.

from the posterior (a few points at $\Theta = 3$) have all of the probability mass.

Resample Move Sequential Monte Carlo (RMSMC) and Sequential Monte Carlo (SMC), are both able to overcome the limitations of Importance Sampling, through the use of a divide and conquer approach to posterior estimation [32]. Trace Markov Chain Monte Carlo (TMCMC), is, in the limit of samples, also able to overcome the limitations of a prior distribution that does not closely resemble the posterior, though the domain where it works efficiently is generally different to that of SMC. Other approaches to solving the issues of importance sampling exist in the literature, such as suggesting proposal distributions closer to the posterior [33]. RMSMC, SMC and TMCMC are implemented in the next Chapter.

Chapter 4

Inference algorithms

This section describes four inference algorithms with operational descriptions of their implementations. In particular:

1. I implement 2 building blocks from Monad Bayes: resampling and MH update.
2. I combined these into: SMC, TMCMC, RMSMC and PMMH.

4.1 Resampling

In order to solve the aforementioned limitations of importance sampling, Doucet et al. [14] implements a resampling mechanism that probabilistically removes particles with low importance weights, and probabilistically multiplies particles with high importance weights. Composing sequential importance sampling with this resampling procedure results in Sequential Monte Carlo [17].

Doucet et al. [34] suggest three resampling procedures; multinomial, systematic and residual sampling. The multinomial sampling approach is adopted in the Sequential Monte Carlo implementation that I provide. In multinomial sampling, in order to generate n samples from the target distribution, n uniformly random samples are generated from the unit interval. Each of these randomly generated samples $u_1, u_2 ... u_n$ are used as input to a generalized inverse cumulative probability distribution of the normalized weights w, i.e. $F^{-1}(u_i)$. This will return the particle x_i with i such that $u_k \in [\sum_{s=1}^{i-1} w_s, \sum_{s=1}^{i} w_s)$ [35]. This returns a particle with weight w_i with probability $\frac{w_i}{\sum_{i=1}^{N} w_i}$.

From an operational point of view, the resample function in the code first calculates the total weight of all of the particles in the histogram that represents the posterior. A random sample from the uniform distribution is multiplied with this total weight, which results in a random proportion R of the total weight. The histogram of all particles is then recursed over such that each particle subtracts its own weight, from R. Eventually the value is subtracted until it decreases to zero or below. At this point, the particle that is being considered at that point in the recursion is returned. In this way, the probability of a particle being returned is in direct proportion to its weight.

4.2 Sequential Monte Carlo

Sequential Monte Carlo (SMC) is an inference algorithm that aims to sample from high dimensional target probability densities $\pi(x_{1:n})$ (such as the posterior), which cannot be sampled from directly, by using samples from lower dimensional densities that increase in dimension i.e. $\pi(x_1)$ then $\pi(x_{1:2})$ etc. Typically SMC is used for *filtering* or *smoothing* over models with outputs y and latent variables x [14, 34]. In smoothing, the objective is; given data $\{y_1, ...y_n\}$, to infer the probability distribution $p(x_t|\{y_1, ...y_n\})$ s.t. $t < n$ and in the case of filtering to infer the probability distribution $p(x_t|\{y_1, ...y_n\})$ with $t = n$. More formally, the objective in filtering is to obtain the distribution $p(x_{1:n}|y_{1:n})$ where $p(x_{1:n})$ is the prior and $p(y_{1:n}|x_{1:n})$ is the likelihood [14].

$$p(x_{1:n}) = \mu(x_1) \prod_{k=2}^{n} f(x_k|x_{k-1})$$

$$p(y_{1:n}|x_{1:n}) = \prod_{k=1}^{n} g(y_k|x_k)$$

In these equations, the function f is the state-transition function and g is the observation function. Using Bayes rule:

$$p(x_{1:n}|y_{1:n}) = \frac{p(y_{1:n}|x_{1:n})p(x_{1:n})}{p(y_{1:n})} = \frac{p(y_{1:n}|x_{1:n})p(x_{1:n})}{\int p(y_{1:n}|x_{1:n})p(x_{1:n})\,\mathrm{d}x_{1:n}}$$

and a simple proof provided by Doucet et al. [31] demonstrates a recursive relationship exists between posteriors at different steps. SMC exploits this recursive relationship between the posteriors at different time-steps in order to perform efficient inference [31][32]:

$$p(x_{1:n}|y_{1:n}) = p(x_{1:n-1}|y_{1:n-1}) \frac{f(x_n|x_{n-1})g(y_n|x_n)}{p(y_n|y_{1:n-1})}$$

$$w_n = w_{n-1} \frac{f(x_n|x_{n-1})g(y_n|x_n)}{p(y_n|y_{1:n-1})}$$

The SMC algorithm is presented in three steps [34]. The first step samples particles, the second computes importance weights and the third resamples the particles.

4.2.1 Sequential Monte Carlo implementation

The Bootstrap Filter is an example of a well known, modular, SMC algorithm and is the combination of sequential importance sampling with a resampling procedure [14, 32]. The SMC algorithm implemented here is an instance of this algorithm. SMC first populates a histogram with particles that will represent the posterior distribution. The models that are used to populate the histogram are transformed into a series of functions, which when called, run the model by a specified number of score statements. This corresponds to advancing a computation by a number of suspension points. By partially evaluating a model, one is able to do a bit of inference followed by running the model and then resampling, before doing more inference. This corresponds to the layout of the generic SMC algorithm [34] which has successive rounds of weight evaluation steps followed by resampling steps. This is achieved in code by running the model, through the `populate` function parameterized by the `populate_func()` function on lines 1 – 3 in Listing 4.4. The `populate_func()` function composes the `advance` handler with the `yield_on_score` handler.

In Listing 4.2, the `yield_on_score` handler handles score effects by emitting a yield effect, before resuming the program. Yield effects allow the computation to be suspended. When handled by the `advance` handler, they can be positioned in such a way as to allow for the computation to proceed by a specific amount. The `advance` effect handler, upon encountering a yield statement, turns the handled computation into a function that takes a value a of type int. This value a is conditioned on by the handler, such that if it is greater than zero, the yield effect is discharged and the program is resumed with $a-1$ placed on the stack, otherwise the yield effect persists and 0 remains on the stack before program resumption. This corresponds to lines 6 - 11 in Listing 4.1. Calling these functions with an integer n, will mean at most n score statements are handled in-between resampling steps, thus bringing the computation closer to returning a model sample. When this value reaches 0, the computation will once again yield. When the advance handler resumes each computation, because of the deep nature of the advance handler, subsequent score statements will still be handled by this handler,

which will now by default, emit a yield. This is because the parameter with which it is called, will stay at 0 once it reaches 0. The `advance` handler is used to handle yields which other advance handlers emit (i.e. they are composed). When the function induced by the advance handler is consistently passed a small integer n and there are many score statements in the computation, then there will be a linear increase in the number of handlers used per score statement.

Listing 4.1: Advance effect handler

```
fun advance(action
    : () -> <yield|e> b)
    : e ((a : int)
    -> <yield|e> b) {
  handle(action) {
    return x -> fun(a){x}
    yield() -> fun(a) {
      if (a > 0) then {
        (mask<yield>{
        resume(())})(a - 1)
      } else {
        yield()
        (mask<yield>{
        resume(())})(0)
}}}}
```

Listing 4.2: Yield on score effect handler

```
fun yield_on_score(action) {
  handle(action) {
    return x -> x
    score(w) -> {
    score(w);
    yield();
    resume(())
}}}
```

Listing 4.3: Finalize effect handler

```
fun finalize(action) {
  handle(action) {
    return x -> x
    yield() -> resume(())
}}
```

Listing 4.4: Functions that underlie Sequential Monte Carlo

```
fun populate_func() {
    advance { yield_on_score { model() } }
} // This function is use to initially populate the histogram.

fun adv_func(wm) {
  match(wm) { (w, m) -> { weighted(w) { advance { m(step_size)}}}}
} // This function advances each particle by a certain step_size.

fun fin_func(wm) {
    match(wm) { (w,m) -> { weighted(w) { finalize{ m(0) }}}}
} // This function handles remaining yields given no more steps.
```

After only a few thousand handled score statements, each single score statement will

be handled by many thousands of nested advance handlers, with thousands of separate parameters stored on the stack. This behaviour is the cause of a space leak, that is likely responsible for a dysfunctional behaviour of RMSMC and the poor time complexity of SMC. A solution to this issue is to use shallow effect handlers. Shallow effect handlers don't rehandle the computation and instead return a computation that may still incur effects which match those that have been handled. Shallow handlers can be called recursively on the handled computation, so that there is at most one handler per yield statement. I didn't use shallow handlers because Koka's current implementation does not support them.

After `populate_func()` has populated the histogram, through repeatedly executing the model and handling it with the aforementioned handlers, the histogram is now a list of functions and is passed to the main loop that executes SMC. This main loop checks if the maximum number of steps has been reached. If there are still steps remaining – a resampling step occurs and each model is advanced by `step_size`. In order to advance the computations and handle yielding computations, the `adv_func` function on lines 5 - 7 in Listing 4.4, is mapped over the histogram that represents the posterior. The `adv_func` function has three components. The first is the weighted handler, which takes the existing particle weight and updates it by multiplying it with the parameters of successive score statements. In general, throughout the library, underflow issues are avoided when computing with large negative exponential powers, by encapsulating the powers within the `Exp` data-type and operating on the powers rather than the floating point result of evaluating the power. This means that a high degree of precision is likely to be preserved under multiplication, addition and division. The second is the `step_size` parameter that is passed to each function in the histogram. This is passed to one of the models and is used to advance the model computation. The third is the advance handler that is used to handle yields from previous advance handlers.

The number of steps given as a parameter is an upper bound on the number of steps between resampling operations. This is because the advance handler handles return statements by simply discarding the parameter and returning the model sample. Resampling steps alone will reduce the diversity of the particles because particles with marginally lower weights than other particles will steadily be eliminated.

The other branch of the main loop is executed when there are no more steps remaining in the SMC algorithm. This branch, applies `fin_func` to each particle in the histogram. There are also three key steps here. The first involves the weighted handler that keeps track of the model weight. The second is the `step_size` parameter that is now zero. This can be understood by examining Listing 4.3. Here, the finalize handler handles yield statements, by immediately resuming. The consequence of this is that if any other advance handler yields, the parameter a will not affect the control flow. By immediately resuming each yield, we run the model to capacity and then we encounter a return statement. SMC returns the histogram as the output.

4.3 Trace Markov Chain Monte Carlo

4.3.1 MCMC

In the context of Bayesian Inference, Markov Chain Monte Carlo methods perform random walks over the posterior distribution by using transition kernels to generate samples based off current samples. Trace Markov Chain Monte Carlo (TMCMC), uses the space of program traces as the distribution and therefore uses two key components. The first key component is the concept of a program trace which is a list of random choices a program has made [17]. The second key component is the Metropolis Hastings update procedure.

To describe MCMC, I follow the theoretical description of Metropolis Hastings provided by Mackay [36]. The Metropolis Hastings algorithm is a common instance of an MCMC method, and is comparable to importance sampling in that it is a technique to estimate target densities through the use of proposal densities. However, the denotational validation of inference, off of which Monad Bayes and therefore Koka Bayes is based, is not restricted to density functions [11]. The Metropolis Hastings algorithm assumes access to some multiplicative constant of the target density $P^*(x)$ s.t. $P(x) = {}^{P^*(x)}/_{Z}$ for any state x, as well as a proposal density $Q(x';x^i)$ that depends on the state at the current step i; x^i. This proposal density Q can be *any* fixed positive density ($Q(x';x^i) > 0$ for all x, x') from which samples can be drawn, in order for the method to converge to the target density [36].

MCMC uses a proposal distribution to construct a Markov Chain with the target

distribution (the posterior), as the limiting distribution. In Koka Bayes, the target distribution is approximated with a list of samples. At the first step, an initial list is created and initialized with an initial state x^0 – sampled arbitrarily. In general, at time-step i, new states x^{i+1} are proposed, based off of the current state x^i. Whether or not new states are added to the list or not (the Metropolis Hastings update procedure), is conditional on the result of evaluating a Bernoulli random variable. This random variable is parameterized by an un-normalized ratio a:

$$a = \frac{P^*(x')}{P^*(x^i)} \frac{Q(x^i;x')}{Q(x';x^i)} \tag{4.1}$$

such that if $a \geq 1$ then the new state is accepted, and if $a < 1$ then the new state is accepted with probability a. If the Bernoulli random variable evaluates to `False` then the new state is rejected, else the old state is added to the list once more.

The ratio a is greater at higher probability areas of the target density, weighted by the symmetry of the proposal distribution. For example, if the proposal distribution is not symmetrical then the ratio will tend to move in a certain direction more than others. If the proposal distribution is symmetrical then the right hand fraction reduces to one. A symmetric proposal distribution Q has the property: $\forall i.Q(x^i;x') = Q(x';x^i)$

4.3.2 Program traces and the trace data structure

The arbitrary initialization of the list that approximates the posterior, corresponds to the instantiation of a single list of all of the random choices made over a single run of the model. The posterior distribution in the space of traces is thus represented by a list of lists. This means that an initial list of *trace values* (double-precision floating point randomly sampled values that correspond to random choices in the program) represents the starting point in the Metropolis Hastings algorithm.

The `Trace` data type is central to the implementation of TMCMC and is used as a parameter to iterations of the Metropolis Hastings update procedure. This data type consists of a four-tuple of: the model; the approximated probability of the model result given observed data (obtained via the `weighted` handler); the trace values, and the final model result itself. For example, for the first order Markov Chain in Listing 3.1: the model is the function `g`, the approximated probability of the model is the weight that results from handling the score statements of the model, the trace values consist of

twelve doubles resulting from the six normal distribution samples and the model result would consist of the final returned value `x`.

4.3.2.1 Perturbation of the program trace

Perturbation of the program trace is the way in which new program traces are proposed. The perturbation occurs via the `perturb_trace` function which randomly selects one of the trace values and replaces it with a new uniformly distributed sample. The MH update in Koka Bayes is based off the update procedure that Monad Bayes uses. The modification of the trace may induce a model which has more or less sample statements than there were previously (if the control flow is modified based on the outcome of a sample operation). If the model now encounters more sample statements, then these further samples are assigned uniformly random values from the unit interval. The approximated probability and the length of the traces of both the model with the perturbed trace, and without the unmodified trace, parameterizes the Bernoulli random variable in the MH update procedure, that is conditioned on when accepting proposed traces. This is a result from Monad Bayes, which generalizes the Metropolis Green Hastings theorem to program traces. The ratio that parameterizes the Bernoulli distribution is therefore different to the ratio a in Eqn. 4.1. The proposal distribution favours exploring programs with longer traces, which corresponds to proofs relating the Metropolis Green Theorem to program traces in [11].

4.3.3 Trace Markov Chain Monte Carlo implementation

The `tmcmc` function uses the `replay` effect handler over sample statements to initialize the list and when doing MH updates. The replay effect handler in Listing 4.5 handles sample statements. It keeps track of the number of sample statements encountered during the action being handled and extends or truncates the trace where necessary. When a return statement is encountered, the modified trace is returned in a tuple along with the original return value.

Listing 4.5: Replay effect handler

```
1  fun replay(trace, action) : <sample|e> (list<double>, a) {
2    var new_trace = trace
3    var index     := -1
4    handle(action) {
5      return x -> (split(new_trace, index+1).fst, x)
6      sample() -> {
7        index := index + 1
8        match(new_trace[index]) {
9          Nothing -> {
10           val rnd = sample()
11           new_trace := new_trace + [rnd]
12           resume(rnd) }
13         Just(random_value) -> resume(random_value)
14 }}}}
```

Listing 4.6: MH Update

```
1  fun mh_step(trace) {
2    val n_trace = perturb_trace(trace)
3    match(trace) {Trace(model, p, old_tr, _) -> {
4        val p2b = with_randomness(model, n_trace)
5        match(p2b){ (new_tr, (q, b)) -> {
6          val ratio = min(1.0, (q * old_tr.length)
7          / p * new_tr.length)))
8          val accept = bernoulli(ratio)
9          if(accept) {  Trace(model, q, new_tr, b) } else {
               trace}
10 }}}}}
```

The `mh_step` function implements the Metropolis Hastings update procedure. This function is parameterized by a `Trace` data structure. The update procedure modifies the trace of the data structure on line 2 in Listing 4.6 using `perturb_trace`. After the trace has been modified, the model is run with the `with_randomness` function on line 6 in Listing 4.6 that uses two handlers; the `replay` effect handler parameterized by the perturbed trace, and the `weighted` effect handler. This function returns a tuple of three things; the new modified trace, the new probability of the model and the final sample that results from the model execution. The result of the `accept` Bernoulli random variable dictates which `Trace` data structure, `mh_step` returns.

The `tmcmc` function takes a model, the number of tmcmc steps, an initial probability of the model and a burnin parameter (which controls for the number of initial samples that are discarded) and returns a pair. The burnin is used to control for the highly dependent nature of the simulation on initial program traces. The first element of the returned pair is another list of pairs. Each pair in this list contains trace values paired with the corresponding return values. The reason that both the trace and the result are stored are due to efficiency reasons, in that once the posterior trace list is returned, the model does not need to be run again. The second element is the final `Trace` data structure which results when the algorithm terminates. The first element of the returned pair is enough to be able to approximate the posterior.

4.4 Resample Move Sequential Monte Carlo

Resample Move Sequential Monte Carlo (RMSMC) [13] uses a move step in addition to resampling steps in SMC. Move steps perform one or more iterations of MCMC after each resampling step. This solves an important issue of SMC which is that particles with high importance weights at early time-steps, may induce poor random choices later on, which degenerates particles, that are otherwise good approximations of the posterior.

4.4.1 Resample Move Monte Carlo implementation

In-between resampling and advance steps, each particle has Markov Chain Monte Carlo Metropolis Hastings updates step applied to it. This is achieved in Listing 4.8. Specifically this formula is placed into the `adv_func` function from Listing 4.4. Four parameters are passed to the tmcmc function: a model, `t_steps` is the number of Metropolis Hastings steps to perform, w – the weight of the model sample and the number of burnin samples which is set to 0.

Listing 4.7: Resample Move Sequential Monte Carlo Update.

```
1 tmcmc( f() { m(step_size) }, t_steps, w, 0 ).snd.trace_m
```

4.5 Particle Marginal Metropolis Hastings

Resample-Move uses MH steps inside SMC. Particle Marginal Metropolis Hastings (PMMH) reverses this pattern by using SMC as an MH proposal distribution [37]. PMMH is defined over models that decompose to a prior over parameters $p(\theta)$ coupled with a state space model $p_\theta(x_{1:t}|y_{1:t})$ that features latent variables. PMMH calculates the posterior distribution $p(\theta|y_{1:t}) \propto p_\theta(y_{1:t})p(\theta)$, by using SMC to obtain an estimate of $p_\theta(y_{1:t})$ which is then factored into the MH acceptance ratio. PMMH is primarily used for parameter estimation in time-series models [17].

4.5.1 Particle Marginal Metropolis Hastings implementation

The `parameter_model` represents the prior distribution over parameters, the result of which is used to parameterize the `main_model`. The main model has SMC (without weight normalization) applied to it and the sum of the importance weights is used as the score that approximates $p_\theta(y_{1:t})$. The TMCMC algorithm is then applied to this combined model. As all sample and score statements inside the main model are handled by SMC, TMCMC will only perform inference over un-handled samples inside the parameter model.

Listing 4.8: PMMH algorithm

```
1  fun new_model(parameter_model : model<b,_e>,
2  main_model : b -> model<a,_e1>,
3  particle_num : int, step_num : int)
4  : model<b, _e> {
5    val g = fun() {
6      val params = parameter_model()
7      val smc_hist = smc(particle_num, step_num, 1,
8          main_model(params), False)
9      score(sum_weights(smc_hist))
10     params
11   }
12   g
13 }
14 fun pmmh(parameter_model, main_model) {
15   val pmmh_model = new_model(parameter_model,
16     main_model, 10, 10)
17   tmcmc(pmmh_model, 10, Exp(0.0), 0)}
```

4.6 Evaluation

4.6.1 Quantitative Evaluation

Figure 4.1 details timing experiments, run over a Logistic Regression model ported from Monad Bayes to Koka Bayes, on a Macbook Pro with a 2.7 GHz Intel Core i7 processor (16 GB 2133 MHz LPDDR3 RAM). The Logistic Regression model was chosen to ensure comparability between the libraries. However, there are considerable issues in comparing the performance of Koka Bayes to that of Monad Bayes. In particular, I did not run Monad Bayes over the same processor architecture that I tested Koka Bayes on, and Monad Bayes uses suspension points after every score statement in the Logistic Regression model. This means that after every score statement, resampling steps occur. Performing resampling steps after every score statement becomes intractable for models with many score statements for Koka Bayes's implementation of RMSMC and (to some extent) SMC. I expect that this intractability derives from the space leak of the deep advance handler. RMSMC has poor overall performance in that whilst it copes with arbitrary rejuvenation steps – it tolerates only one resampling step and therefore is not displayed in this section. Improvement in the efficiency of RMSMC is left as future work, because Koka needs to fix an identified bug with shallow handlers. This would then allow me to implement SMC^2, which is a compound inference algorithm based off of RMSMC.

Figure 4.1a shows the exponential ($R^2 > 0.985$ – this coefficient of determination means that 98.5% of the variation in time is described by the variation of an exponential function fitted via maximum likelihood estimation over parameters A and B from the parametric class : Ae^{Bx}) scaling of the SMC algorithm as the `step_size` parameter is decreased – which controls how many score statements are handled before resampling steps occur – is changed from N (ADVANCE N) to 4 (ADVANCE 4). Further profiling would be warranted to investigate the reasons underlying the increase in time complexity, i.e. whether it is due to the deep advance handler or due to the resampling mechanism interacting with handlers or some other reason. However, the remarkably quick computation corresponding to advancing the computation all the way (ADVANCE N) before performing resampling steps, suggests that running the model and the resampling procedure without many nested handlers, consists of a negligible fraction of the total computation time. Monad Bayes has a linear scaling, whilst doing one resampling step after every score statement (ADVANCE 1) and takes under

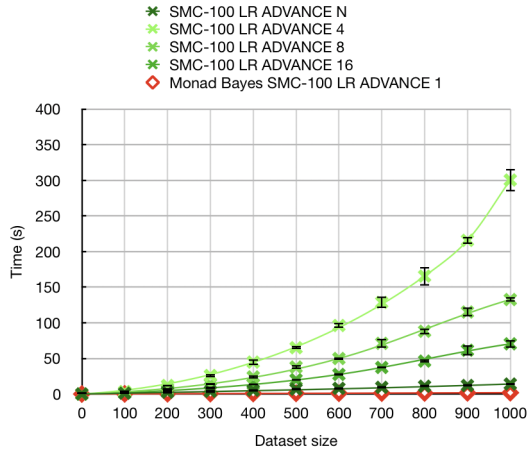
two seconds rather than over three hundred. Figure 4.1c shows that with an increasing number of particles, Koka Bayes achieves an approximately linear scaling ($R^2 > 0.98$) for the SMC algorithm. However, the gradient of the scaling is approximately 68 times steeper than Monad Bayes.

I did not control for the processor architecture, in comparing Koka Bayes to Monad Bayes. However the relative performance between TMCMC and SMC, indicates that TMCMC tends to be faster than SMC. Figure 4.1b shows a polynomial of order two increase, in the time complexity of the Koka Bayes TMCMC algorithm, as the dataset increases in size. Monad Bayes only has a linear scaling over increasing dataset sizes. Unfortunately, without Koka based profiling tools, it is hard to identify where this change in performance stems from. Whilst Koka Bayes performs worse over an increasing dataset size, Figure 4.1d shows that it is almost twice as fast as Monad Bayes as the number of MH steps in the algorithm increases, even though Koka Bayes has a higher startup time.

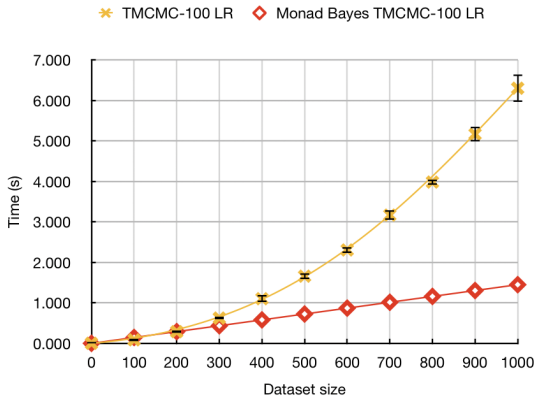
4.6.2 Qualititative Evaluation

Table 4.1 details the number of lines of code (LoC) in the inference algorithms implemented in Koka Bayes. The LoC reflect the implementation complexity and implementation effort of algorithm implementations and provide some indication of the ease, with which reasoning about the correctness of the algorithms can be done. Koka Bayes performs competitively with respect to both WebPPL and Anglican. The modular design of Koka Bayes reflects that of Monad Bayes and therefore I expect that the number of lines of code to implement further compound inference algorithms such as SMC^2 [17] will be smaller than the number of lines required to build the algorithms in monolithic designs such as WebPPL and Anglican. In addition, the type-safety of Koka Bayes prevents many different forms of runtime errors that occur in languages such as Pyro, WebPPL and Anglican.

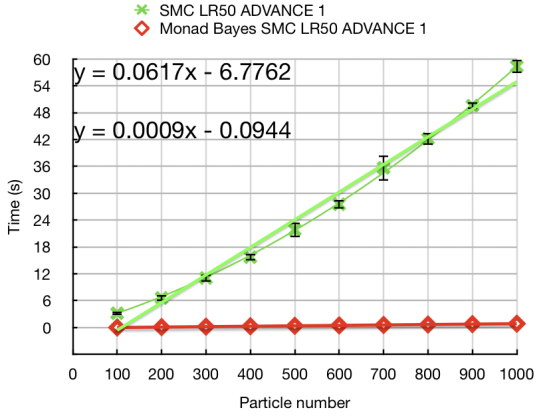
Many of the same design principles that Monad Bayes uses are also found in Koka Bayes. The similarity of the libraries is supported by the equivalence expressivity of algebraic effects and monads [38]. For instance, as previously mentioned, Monad Bayes uses the MonadSample, MonadCond and MonadInfer monads. In Koka Bayes, these correspond to the sample, score effects and the model type, respectively. Monad



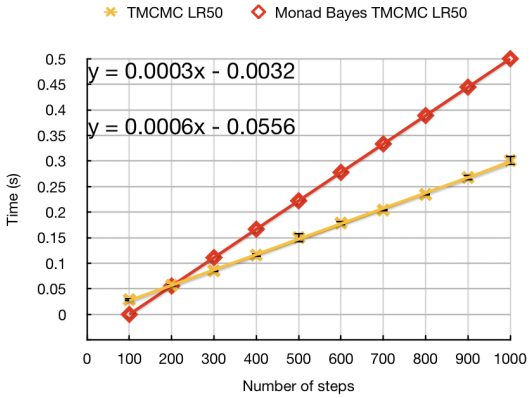
(a) SMC performance : increasing dataset.



(b) TMCMC performance : increasing dataset.



(c) SMC performance : increasing particles.



(d) TMCMC performance : increasing steps.

Figure 4.1: The top row uses the dataset size as an independent variable. The bottom row uses the number of steps / particle number as an independent variable. LR50 denotes a logistic regression model with 50 data points. ADVANCE N in general corresponds to the number of score statements advanced through before resampling steps occur. The algorithm name followed by a number denotes the number of particles or steps of the algorithm. The error bars indicate the standard-deviation of the time over four runs of the algorithms. The top equation in 4.1c refers to Koka Bayes, whilst in 4.1d it refers to that of Monad Bayes.

Library	MH	SMC	RMSMC	PMMH
Koka Bayes	124	104	30	10
Monad Bayes	67	70	11	4
WebPPL	314	334	0	N/A
Anglican	100	87	N/A	N/A

Table 4.1: Lines of code excluding comments and import statements. To deal with excessive newlines caused by additional curly brackets, I compacted curly brackets into single lines. This makes for a fairer comparison with Monad Bayes's Haskell code base which doesn't use curly brackets. N/A indicates that the algorithm is not available in a given language.

Bayes also uses the principle of suspension points in a program that can be advanced or finished. Koka Bayes's similarity to Monad Bayes, crucially means that it is close to Scibiors semantic validation of inference [11]. This means that reasoning about the code with respect to the presentation of the inference algorithms is easier.

Chapter 5

Use case : climate change modelling

The 2015 Paris Agreement aims to keep temperature increases to "well below 2 °C above pre-industrial levels and pursuing efforts to limit the temperature increase to 1.5 °C above pre-industrial levels" [39]. Remarkably, no temporal or spatial demarcation as corresponding to "pre-industrial levels" [40] has been officiated. The consequence of this, is that the notion of climate change since "pre-industrial levels" is ambiguous. Hawkins et al. [41] suggest a period of 1720-1800 as the period corresponding to pre-industrial levels. They arrive at this estimate by controlling for macroscopic climate events such as volcanic events, solar activity and changes in the Earth's orbit. Fortunately, there is a free, publicly usable Kaggle dataset[1], released by Berkeley Earth, of average global land temperatures dating back to 1750. With this dataset, I demonstrate the use of Sequential Monte Carlo and Trace Markov Chain Monte Carlo based inference over a simple extension of a linear Gaussian state-space model. The goal of this Chapter is not to draw new conclusions about climate change, but to demonstrate the potential of Koka Bayes.

5.1 The Kalman Filter

Linear Gaussian state-space models are known as Kalman Filters [42]. The linear Gaussian model builds on a vector of latent unobserved states $\mathbf{x}$ which produce an observable vector $\mathbf{y}$. The evolution of unobserved states follows a first order Markov Chain. In particular, later latent states are related to previous latent states via multiplication with a fixed *state-transition* matrix $\mathbf{A}$ and addition with a time-dependent vector

[1] https://www.kaggle.com/berkeleyearth/climate-change-earth-surface-temperature-data

$\mathbf{w}_t \sim \mathcal{N}(0, Q)$ where Q is a co-variance matrix.

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{w}_t$$

At discrete time intervals t, the corresponding unobserved state $\mathbf{x}_t$ produces an observable output $\mathbf{y}_t$. The observable vectors $\mathbf{y}$ are evaluated by transforming the unobserved state $\mathbf{x}$ by a *observation* matrix C and then adding a vector $\mathbf{v} \sim \mathcal{N}(0, R)$ where R is a co-variance matrix.

$$\mathbf{y}_t = \mathbf{C}\mathbf{x}_t + \mathbf{v}_t$$

5.1.1 Extended linear Gaussian state-space model

The extended linear Gaussian model I create is both a restriction and a generalization of the Kalman Filter. I restrict the model by assuming that the unobserved states $\mathbf{x}$ are one-dimensional floating point values. A corollary of this assumption is that the state-transition matrices $\mathbf{A}$ are one-dimensional floating point values. I generalize the model by having a time dependent state transition matrix $\mathbf{A}_t$ as well as a time dependent observation matrix $\mathbf{C}_t$. In addition, the covariance matrix R is also time dependent. With this formulation, the observable vectors $\mathbf{y}$ are still conditionally independent of each other, which can easily be verified by d-separation [43]. The joint probability distribution of the model therefore factorizes into the same form as a Hidden Markov Model:

$$p(\{\mathbf{x}_1 ... \mathbf{x}_n\}, \{\mathbf{y}_1 ... \mathbf{y}_n\}, \theta) = p(\theta) p(\mathbf{x}_1|\theta) \prod_{t=1}^{n-1} p(\mathbf{x}_{t+1}|\mathbf{x}_t, \theta) \prod_{t=1}^{n} p(\mathbf{y}_t|\mathbf{x}_t, \theta)$$

$$\mathbf{y}_t = \mathbf{C}_t\mathbf{x}_t + \mathbf{v}_t$$

$$\mathbf{x}_{t+1} = \mathbf{A}_t\mathbf{x}_t + \mathbf{w}_t \quad \mathbf{w}_t \sim \mathcal{N}(0, 1) \tag{5.1}$$

There are two important machine learning tasks that are typically performed when considering Kalman Filter models [42]. The first is parameter *learning* or *system identification*, which involves being given a series of outputs $\{\mathbf{y}_1, ... \mathbf{y}_n\}$ and to subsequently infer the values of the parameters $\theta = \{\mathbf{A}, \mathbf{C}, Q, R\}$. The other task is *filtering* mentioned in Section 4.2.1, which involves obtaining the probability distribution $p(\mathbf{x}_t|\mathbf{y}_1, ..., \mathbf{y}_n)$ with $t = n$. The inference methods perform both tasks over the climate model presented.

5.2 Experiments

5.2.1 Berkeley Earth Dataset

The dataset consists of single data points that represent average land temperatures for the entire Earth, for each month from the period of 1756 until 2016 (260 years). There are no data points missing from the dataset. Each datapoint is paired with a corresponding 95% confidence interval. These uncertainty intervals take into account both spatial incompleteness of the temperature readings and statistical uncertainty, which arises because the temperature readings themselves may not be true reflections of the Earth's temperature [44]. Whilst no data is missing from the dataset provided to me, in early years only two weather stations are used to calculate the entire Earth's temperature and so the corresponding uncertainty values for those years with fewer weather stations are much higher. In order to convert the confidence intervals to standard deviations such that they can be parameterized by a Gaussian random variable, I sought the advice of both Dr. Schurer and Dr. Zeke Hausfather – the curator of the Berkeley dataset. They advised that in order to convert the confidence intervals to a standard deviation, divide each value by 3.92 (the number of standard errors that fit into a 95% confidence interval) to obtain the standard error and using that as the standard deviation – corresponding to a sample size of one.

5.2.1.1 Setting model hyper-parameters

I split the Berkeley Earth data chronologically, into thirteen vertical slices of twenty years, giving a balance between resolution and sample size. There need to be enough slices so that important climatic trends are represented, yet not too many slices – because there would then be few data points per block, resulting in both less interesting inference and higher variance in the temperature values which may not indicate the true latent temperature of the Earth. This partition results in twenty year slices of climate readings which, for each slice, I slice horizontally into months. This leaves thirteen twenty year slices and $\mathbf{x}_t$ represents the latent temperature of a particular twenty year month block. I assume that every twenty years, the Earths monthly climate changes in accordance with Eqn. 5.1. For each twenty year month block the model detailed in Listing 5.1 is used.

Listing 5.1: Linear Gaussian Model applied to climate data for only one month

```
var x := [] // Reference to list that stores latent variables.
for(0, 12) fun(i) {
  if(i == 0) {
    x := x + [(sample() * diff) + subtract] // Initial state dist.
  } else {
    val a_val = normal(a_mean,a_std_dev) // state-transition
    val w_val = normal(w_mean,w_std_dev) // transition noise
    x := x + [exn-get(x, i - 1) * a_val + w_val] // Transition.
  }
  val month_ys = exn-get(ys,i) // Read in temperatures.
  val month_vs = exn-get(vs,i) // Read in uncertainty.
  val c_row = multivariate_gaussian(month_ys.length, 1.0 +
      mult_bias_of_thermometer, mult_bias_of_thermometer_std_dev)
  val v_row = convert_uncertainty_to_rand(month_vs)
  val predictions = plus(mult(exn-get(x, i)
                         , c_row), v_row) // Predictions
  score_predictions(month_ys, predictions, score_var)
  ()
}
```

I chose parameters based on a consultation and subsequent email follow-ups with Dr. Andrew Schurer – an expert climate researcher based in Edinburgh – who has personal experience with the Berkeley dataset. I used this consultation to try and elicit priors for global temperature values in 1750. I used a prepared script, an interactive email dialogue and discussions about likely percentiles in order to elicit priors, as per recommendations in the literature [45]. I found prior elicitation to be challenging, because the expert was unable to provide an absolute estimate in degrees Celsius of the temperature, and he largely depended on estimates of relative climate values from other empirical climate simulations to obtain such 'priors'. In order to avoid being biased from such estimates, I decided to use an uninformative uniform prior over $\mathbf{x}_0$ parameterized by the minimum and maximum temperatures in the first twenty months, for each month. The uniform distribution was parameterized in this manner, to allow the Monte Carlo based inference strategies to remain tractable, yet realistic. This choice is realistic because it excludes inferred initial temperatures below and above the minimum and maximum recorded temperature during the first block. This choice creates tractability because the search space is reduced relative to an alternative choice; using the lowest and highest temperatures ever recorded on Earth. In the end, I did not re-

ceive a useful prior from the expert.

The observation matrices $\mathbf{C}_t$ reflect the systematic multiplicative bias of the thermometers that are used to measure the temperatures [46]. In order to reflect the uncertainty of the expert on this particular bias, and to reflect the high accuracy of mercury as well as electronic thermometers, this parameter was set to $\mathcal{N}(1, 0.05)$ throughout the slices. However, experimenting with different suggestions made by domain experts in thermometer calibration would be warranted to improve the model. The additive noise parameter $\mathbf{v}_t$ is parameterized by uncertainty values given in the original dataset. The state-transition values $\mathbf{A}_t$, are parameterized as $\mathcal{N}(1, 0.4)$, in line with empirical observations of the given Berkeley data. The score statements were parameterized by Gaussian probability density functions with μ chosen to be the observed data and σ chosen at a value of 4.7. This corresponds to assuming the observable data are corrupted by Gaussian noise. The standard deviation was small enough to ensure relatively accurate inference, yet large enough to accommodate both numerical stability issues in the inference process and prevent particle degeneracy in SMC.

5.2.2 Inference hyper-parameters

The parameters I chose to control for the scale of the inference, as well as the time taken for the inference, are detailed in Table 5.1. I chose the largest number of particles for SMC and the highest step number for TMCMC, that Koka managed without throwing a runtime exception. In particular, Koka is unable to write or read files with more than a few thousand lines and Koka is unable to perform key functions such as map and split over lists that have many thousands of elements. I ran both inference algorithms twice to get an idea of the variance of the inference procedures when inferring temperatures. As Koka can only handle 2,000 particles – a value small enough that it fails to provide high resolution of the posterior distribution at earlier latent states, I combined four runs of SMC for each month that I analyzed. I used a burnin of 80,000 samples for TMCMC.

5.3 Results

Figure 5.1a and 5.1g demonstrate that overall, SMC and TMCMC differ over the inferred global temperature change. SMC infers an *increase* in global temperatures of

Algorithm	Time (s)	Steps/particles	Posterior representation	Runs
TMCMC	1900 ± 320	100,000	20,000 samples	2
SMC	1100 ± 40	2,000	8,000 samples	2

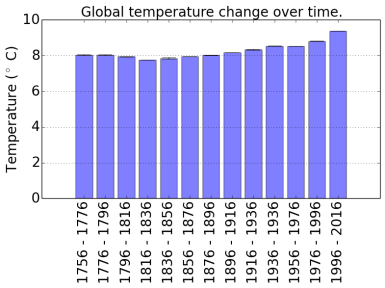
Table 5.1: Inference hyper-parameters and also time taken. Time was recorded on a Macbook Pro (2016).

approximately 1.6 degrees Celsius over the entire time period. TMCMC infers a *decrease* in global temperatures of approximately three degrees Celsius. However, there are similarities; for example, during the month of December, the temperatures of both algorithms show a similar increase between 1756 and 2016, of 1.5 degrees in the case of TMCMC and of about 1.8 degrees in the case of SMC. TMCMC and SMC both infer Gaussian distributed temperatures for the month of March. Often, TMCMC ends up in local optima which results in a single sample representing the entire posterior for all time periods. Examples of this phenomena are depicted in Figure 5.2, for two twenty year blocks at the beginning and end of the 260 year period studied.

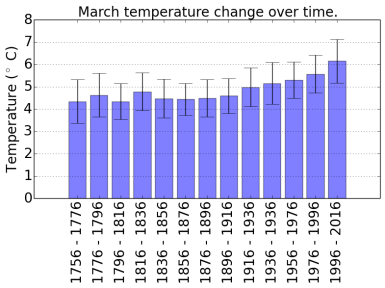
5.4 Evaluation and future work

The global temperature change inferred by SMC corresponds to the findings of Berkeley Earth researchers [48]. SMC performs well here because it is an algorithm designed with linear state-space models in mind [14]. However, even with the use of SMC, the compatibility intervals for monthly temperatures in 1756 - 177 often overlap with those of 1996 - 2016 (see Figure 5.1b for an example). This overlap means that whether climate change is occurring, can, to some extent be doubted. Future work would investigate inference algorithms that achieve more accurate compatibility intervals. The reason the compatibility intervals for SMC based inference are large is related to the parameter that controls for corruption of the noise, which has to be large enough to prevent particle degeneracy in SMC. RMSMC reduces particle degeneracy and would likely allow for a smaller noise corruption parameter thus providing more accurate inference.

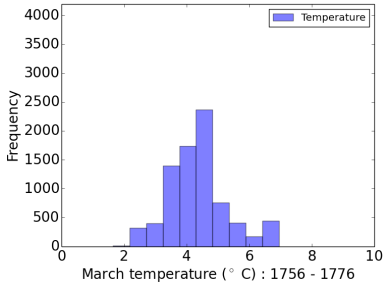
TMCMC infers the wrong temperature change. I suggest a few reasons for this. Firstly, TMCMC works by optimizing over a real valued vector space with many di-



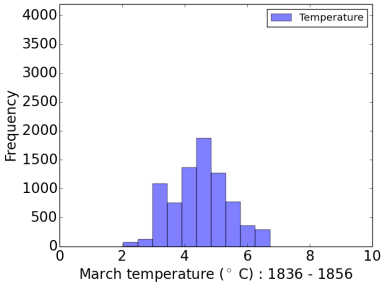
(a) Global temperature change over time



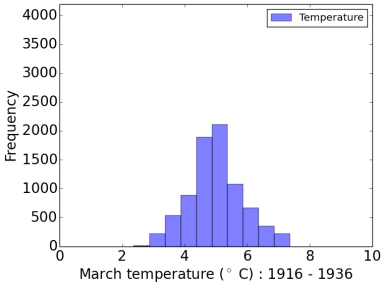
(b) March temperature change SMC



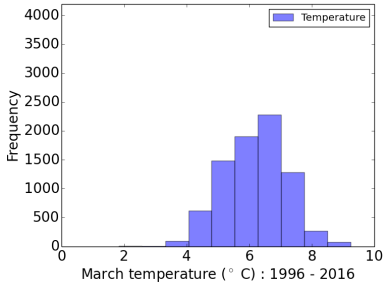
(c) March temperature 1756-1776



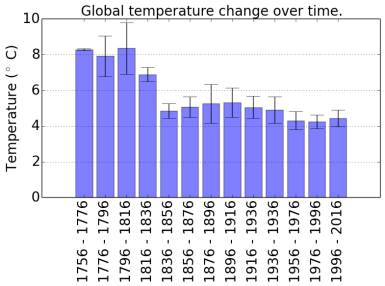
(d) March temperature 1836-1856



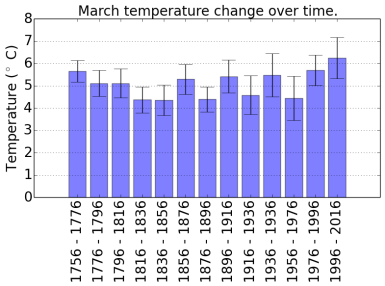
(e) March temperature 1916-1936



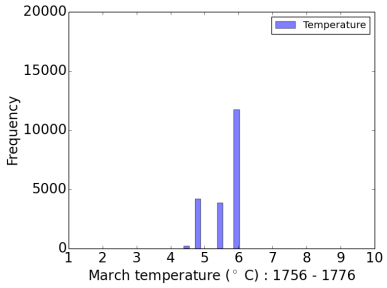
(f) March temperature 1996-2016



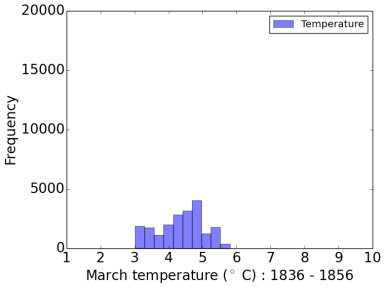
(g) Global temperature change over time



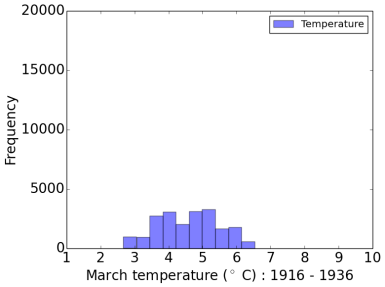
(h) March temperature change TMCMC



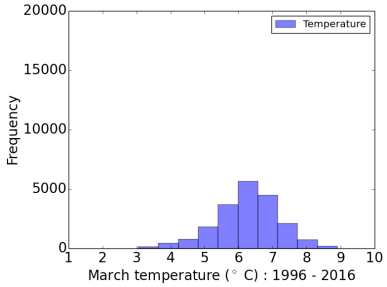
(i) March temperature 1756-1776



(j) March temperature 1836-1856

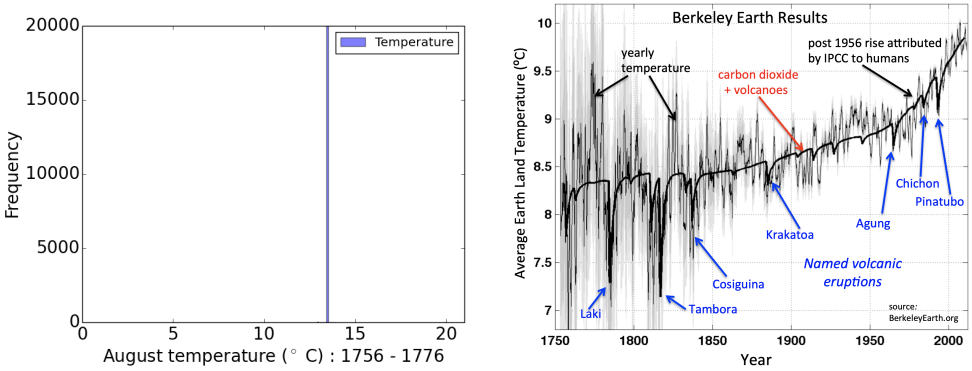


(k) March temperature 1916-1936



(l) March temperature 1996-2016

Figure 5.1: The top two rows show the result of inference using SMC and the bottom two rows show the result of inference using TMCMC. The confidence intervals of the temperature change over March for both SMC and TMCMC represent the standard deviation of all model samples that represent the posterior. The confidence intervals of the global temperature change refer to the standard deviation of running the inference twice.



(a) August temperature 1756 - 1776 (b) Source : Berkeley Earth [47]

Figure 5.2: The leftmost image depicts the tendency of the TMCMC algorithm to become trapped in local optima, in this case for August, with only single samples representing the posterior. The rightmost image depicts the results of a press release in 2012 by Berkeley Earth depicting just over 250 years of global warming.

mensions – equal to the number of random choices that a model performs. In the case of the climate model there are a little under 1100 sample statements, though one could imagine a large model with many millions of sample statements – with some samples having less impact on the inference process than others. Even performing one hundred thousand TMCMC steps, in expectation, each sample statement will only be perturbed 110 times. This is a vanishingly small fraction of the total number of sample statements needed to explore every possible trace and so it may only explore a local optima. Secondly, because the model has a sequential structure, perturbing sample statements that correspond to early latent states of the model, later on in the inference process, carries a risk. This is because the change of sample affects later model parameters, and so if all other later samples had already been perfectly optimized for – then the perturbation will cause these samples to be corrupted. Whilst the probability of perturbing early states is low, if early states are rarely perturbed then the diversity of samples of earlier latent states will also be low, corresponding to a posterior distribution for early and sometimes later samples that consists of only one value, as per Figure 5.2a.

Some parameters themselves may be more important than others, so attaching a notion of importance (which could affect how many perturbations occur) to certain sample statements may be a useful direction of research. Further work could investigate the use of TMCMC that stages inference of different parameters, based on depen-

dencies in the model. Providing users of the library with a method to visualize Trace plots – diagrams that provide a visual cue as to whether Markov Chains are well mixed, would also make the library more transparent. In addition to such visualization tools, the library could provide useful statistical tests such as the Kolmogorov-Smirnov test, to indicate whether two distributions are equivalent – thus allowing combinations of parallel computed independent Markov Chains.

Future work could improve both the model and its dataset. For example, ameliorating the naïvety of the Extended Linear Gaussian model using a model that describes the locations of the temperature recordings and potentially the physical dynamics of the thermometers. The simplicity and paucity of the dataset also creates issues. In earlier years, few weather stations were used to estimate the temperature of the whole planet, which makes those estimates unreliable. Other sources of data, such as tree ring or ice core data coupled with the underlying data generating processes could be factored in to the model to improve estimates of climate change. In this way, a better generative model of global warming can be created which incorporates aspects such as the temperature capacity of mercury or urban heat islands. One could investigate statistical model comparison measures such as the Bayesian Information Criterion or Bayes factors to compare different models. For the scope of this project, creating complex models with corresponding inference algorithms that can cope with complex models, would likely be intractable both in terms of development time and inference time. Therefore, I left it as future work. To test further inference algorithms, a simplified simulator of the Earth could be created, that yields noisy measurements of ground truth latent states which could be compared to inferred states.

Chapter 6

Conclusions

6.1 Summary

I have implemented a basic modular probabilistic programming library in Koka based off the design of Monad Bayes. The library supports Importance Sampling and four compound inference algorithms (SMC, TMCMC, RMSMC & PMMH). I have compared the temporal scaling of TMCMC and SMC on larger datasets and inference, against that of Monad Bayes, finding that TMCMC performs competitively, but SMC does not. I have demonstrated the library on an extended linear Gaussian model with a dataset consisting of Earth's average land temperatures with uncertainties over the 260 year period starting in 1756 and ending in 2016.

During this experimental programming project, I encountered a number of challenges. I learned that developing with Koka is difficult, because a number of unforeseen internal compiler bugs and runtime exceptions occurred, which required time expensive workarounds. This is because Koka is a pre-alpha research language. Access to Daan Leijen – the author of the Koka language – was therefore crucial. At a few points in the development process (notably when finalizing RMSMC and creating a Gamma distribution sampling function from a a handbook on Monte Carlo methods [49]) I collaborated with Daan Leijen in order to overcome a number of internal compiler errors. The lack of Koka profiling tools and documentation, also hindered development efforts, especially when attempting to develop a shallow handler to replace the inefficient deep handler. The Bayesian inference library is not yet ready for deployment because of the readiness level of Koka and the library's current performance. Whilst the library will be of use to programming language researchers, the model and

library is unlikely to be of use to statistical and climate researchers due to the limitations of Koka as well as the model. The model is not a serious contender for climate change modelling due to the nave simplicity of the underlying generative model and the limited datasets used in the inference process. Koka has issues in dealing with even moderately sized data sets, far smaller than those used in climate related institutions, such as the MET office. Koka is only able to read and write (only overwrite) files with at most a few thousand, limited length lines. In order to overcome this issue, I staggered the reading and writing into separate stages. In light of such challenges, it is clear that modular probabilistic programming based off of the design of Monad Bayes can be implemented with algebraic effects.

6.2 Future / further work

Future work would focus on making Koka and the library more stable, reliable and performant before adding more sophisticated inference algorithms and models. Of particular interest would be the implementation of variational inference through the use of guide programs [24], profiling tools to assess inference convergence, as well as exact inference through disintegration [50]. Implementing back-propagation would help bring inference to a level competitive with deep learning [51]. Improving the computational efficiency of RMSMC & SMC, through programming language constructs such as shallow handlers, and the statistical efficiency, through the use of d-separation to calculate conditional independence of variables that assist in inference, would hopefully allow the speed of the library to scale similarly to other probabilistic programming languages. It would be interesting to see if probabilistic programming with effects could be implemented in a dependently typed programming language such as Agda [52], where the type system could offer even more guarantees than the type system of Koka. Whilst the development process was challenging, I enjoyed both constructing the library and learning from obstacles overcome. I believe that this work provides a solid foundation for further exploration into the exciting interface of algebraic effects and probabilistic programming.

Bibliography

[1] Scott Cheng-Hsin Yang and Patrick Shafto. Explainable artificial intelligence via bayesian teaching. In *NIPS 2017 workshop on Teaching Machines, Robots, and Humans*, 2017.

[2] David Barber. *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.

[3] John Guiver David Knowles Tom Minka, John Winn et al. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[4] Dustin Tran, Matthew W. Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, and Alexey Radul. Simple, distributed, and accelerated probabilistic programming. In *Advances in Neural Information Processing Systems*, pages 7598–7609, 2018.

[5] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *arXiv preprint arXiv:1810.09538*, 2018.

[6] Chuanhuang Li, Yan Wu, Xiaoyong Yuan, Zhengjun Sun, Weiming Wang, Xiaolin Li, and Liang Gong. Detection and defense of ddos attack–based on deep learning in openflow-based sdn. *International Journal of Communication Systems*, 31(5):e3497, 2018.

[7] Home Office Scientific Development Branch. Imagery library for intelligent detection systems (i-lids). In *2006 IET Conference on Crime and Security*, pages 445–448. IET, 2006.

[8] Babak Ehteshami Bejnordi et al. Diagnostic assessment of deep learning algorithms for detection of lymph node metastases in women with breast cancer. *Jama*, 318(22):2199–2210, 2017.

[9] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied categorical structures*, 11(1):69–94, 2003.

[10] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ACM SIGPLAN Notices*, volume 48, pages 145–158. ACM, 2013.

[11] Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K Moss, Chris Heunen, and Zoubin Ghahramani. Denotational validation of higher-order bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(POPL):60, 2017.

[12] Dave Moore and Maria I. Gorinova. Effect handling for composable program transformations in edward2. *arXiv preprint arXiv:1811.06150*, 2018.

[13] Walter R Gilks and Carlo Berzuini. Following a moving targetmonte carlo inference for dynamic bayesian models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(1):127–146, 2001.

[14] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[15] Radford M. Neal. Probabilistic inference using markov chain monte carlo methods. 1993.

[16] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex bayesian modelling. *The Statistician*, pages 169–177, 1994.

[17] Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. Functional programming for modular bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(ICFP):83, 2018.

[18] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

[19] Nasser Saheb-Djahromi. Probabilistic lcf. In *International Symposium on Mathematical Foundations of Computer Science*, pages 442–451. Springer, 1978.

[20] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. Papers using infer.net, 2018. [Online; accessed 22-December-2018].

[21] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill: a bayesian skill rating system. In *Advances in neural information processing systems*, pages 569–576, 2007.

[22] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. Problog2: Probabilistic logic programming. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 312–315. Springer, 2015.

[23] Andrew D Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. Tabular: a schema-driven probabilistic programming language. In *ACM SIGPLAN Notices*, volume 49, pages 321–334. ACM, 2014.

[24] Marco F. Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236. ACM, 2019.

[25] Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. Noncomputable conditional distributions. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 107–116. IEEE, 2011.

[26] Michael Gutmann. Introduction to probabilistic modelling. 2018.

[27] Cristina Conati, Abigail S Gertner, Kurt VanLehn, and Marek J Druzdzel. Online student modeling for coached problem solving using bayesian networks. In *User Modeling*, pages 231–242. Springer, 1997.

[28] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 809–818. ACM, 2018.

[29] Ohad Kammar and Matija Pretnar. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming*, 27, 2017.

[30] Maria I Gorinova, Dave Moore, and Matthew D Hoffman. Automatic reparameterisation in probabilistic programming.

[31] Arnaud Doucet, Simon Godsill, and Christophe Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and computing*, 10(3):197–208, 2000.

[32] Stephen T Buckland, Ken B Newman, Carmen Fernández, Len Thomas, and John Harwood. Embedding population dynamics models in inference. *Statistical Science*, pages 44–58, 2007.

[33] Jun S. Liu and Rong Chen. Sequential monte carlo methods for dynamic systems. *Journal of the American statistical association*, 93(443):1032–1044, 1998.

[34] Arnaud Doucet and Adam M Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of nonlinear filtering*, 12(656-704):3, 2009.

[35] Jeroen D. Hol, Thomas B Schon, and Fredrik Gustafsson. On resampling algorithms for particle filters. In *2006 IEEE nonlinear statistical signal processing workshop*, pages 79–82. IEEE, 2006.

[36] David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[37] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

[38] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proceedings of the ACM on Programming Languages*, 1(ICFP):13, 2017.

[39] Joeri Rogelj, Michel Den Elzen, Niklas Höhne, Taryn Fransen, Hanna Fekete, Harald Winkler, Roberto Schaeffer, Fu Sha, Keywan Riahi, and Malte Meinshausen. Paris agreement climate proposals need a boost to keep warming well below 2 c. *Nature*, 534(7609):631, 2016.

[40] Andrew P. Schurer, Michael E Mann, Ed Hawkins, Simon FB Tett, and Gabriele C Hegerl. Importance of the pre-industrial baseline for likelihood of exceeding paris goals. *Nature climate change*, 7(8):563, 2017.

[41] Ed Hawkins, Pablo Ortega, Emma Suckling, Andrew Schurer, Gabi Hegerl, Phil Jones, Manoj Joshi, Timothy J Osborn, Valérie Masson-Delmotte, Juliette Mignot, et al. Estimating changes in global temperature since the preindustrial period. *Bulletin of the American Meteorological Society*, 98(9):1841–1856, 2017.

[42] Sam Roweis and Zoubin Ghahramani. A unifying review of linear gaussian models. *Neural computation*, 11(2):305–345, 1999.

[43] Dan Geiger, Thomas Verma, and Judea Pearl. d-separation: From theorems to algorithms. In *Machine Intelligence and Pattern Recognition*, volume 10, pages 139–148. Elsevier, 1990.

[44] Robert Rohde, Richard Muller, Robert Jacobsen, Saul Perlmutter, Arthur Rosenfeld, Jonathan Wurtele, Judith Curry, Charlotte Wickham, and Steven Mosher. Berkeley earth temperature averaging process. *Geoinformatics & Geostatistics: An Overview*, 1(2):1–13, 2013.

[45] Kathryn Chaloner. Elicitation of prior distributions. *Bayesian biostatistics*, pages 141–156, 1996.

[46] Yudong Tian, George J. Huffman, Robert F Adler, Ling Tang, Mathew Sapiano, Viviana Maggioni, and Huan Wu. Modeling errors in daily precipitation measurements: Additive or multiplicative? *Geophysical Research Letters*, 40(10):2060–2065, 2013.

[47] Berkeley Earth. 250 years of global warming, 2012. [Online; accessed 27-July-2019].

[48] Robert Jacobsen Elizabeth Muller Saul Perlmutter Arthur Rosenfeld Jonathan Wurtele Donald Groom Charlotte Wickham Robert Rohde, Richard Muller. A new estimate of the average earth surface land temperature spanning 1753 to 2011, geoinfor geostat: An overview 1: 1. *of*, 7:2, 2013.

[49] Dirk P. Kroese, Thomas Taimre, and Zdravko I Botev. *Handbook of monte carlo methods*, volume 706. John Wiley & Sons, 2013.

[50] Chung-chieh Shan and Norman Ramsey. Exact bayesian inference by symbolic disintegration. In *ACM SIGPLAN Notices*, volume 52, pages 130–144. ACM, 2017.

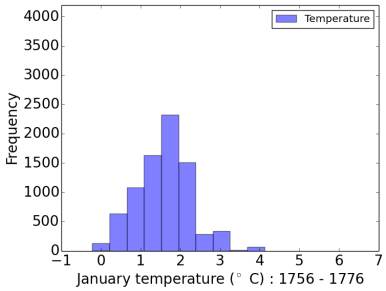
[51] Avi Pfeffer. Learning probabilistic programs using backpropagation. *arXiv preprint arXiv:1705.05396*, 2017.

[52] Ulf Norell. Dependently typed programming in agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

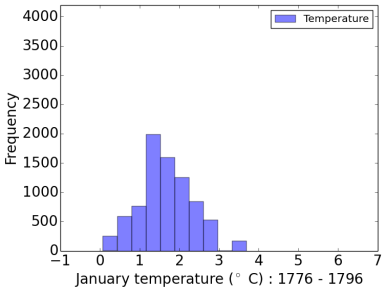
Appendix A

Additional figures for SMC / TMCMC based inference over the extended linear Gaussian climate model.

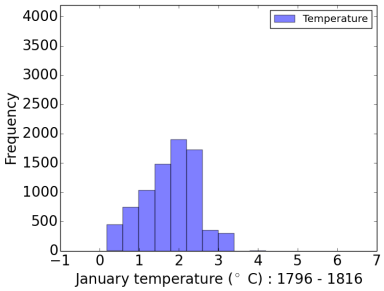
I am including these Figures to demonstrate the inference results for SMC and TMCMC over all months of the year. For each inference type, for each month, and for each twenty year block, a histogram is displayed. Each histogram uses the parameters found in Table 5.1. The histogram represents the values of the particles for SMC and the model values corresponding to each trace for TMCMC. In addition, the overall change for the months are plotted with uncertainty intervals that result from the standard deviation of the inferred temperature when running the algorithm twice. SMC performs well for all of the months (it corresponds to temperature increases that other researchers have found), in part because SMC was designed with linear Gaussian state-space models in mind. TMCMC sometimes performs reasonably well, but often becomes stuck in local optima for some months (e.g. August) – suffering from low diversity in these cases.



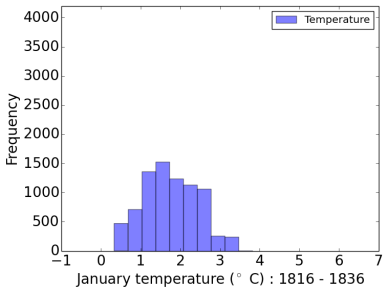
(a) January – SMC: 1756 - 1776



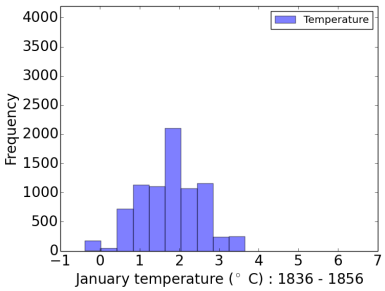
(b) January – SMC: 1776 - 1796



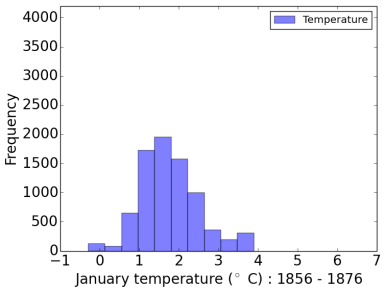
(c) January – SMC: 1796 - 1816



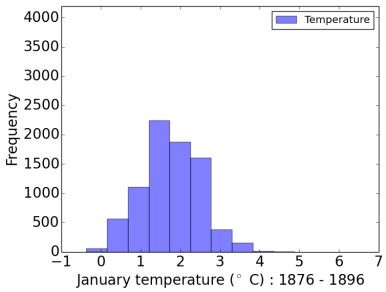
(d) January – SMC: 1816 - 1836



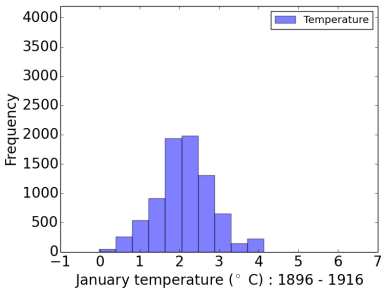
(e) January – SMC: 1836 - 1856



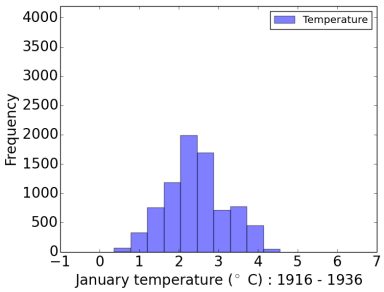
(f) January – SMC: 1856 - 1876



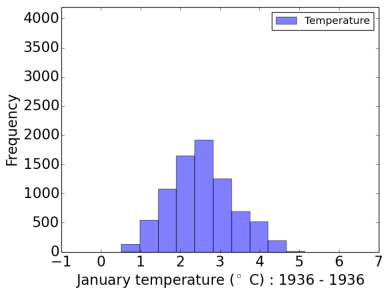
(g) January – SMC: 1876 - 1896



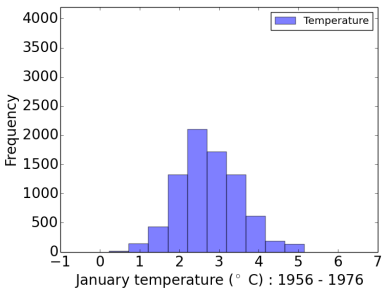
(h) January – SMC: 1896 - 1916



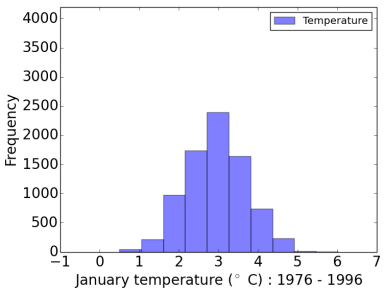
(i) January – SMC: 1916 - 1936



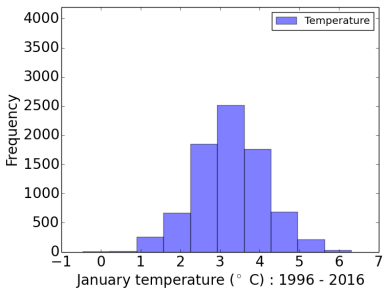
(j) January – SMC: 1936 - 1936



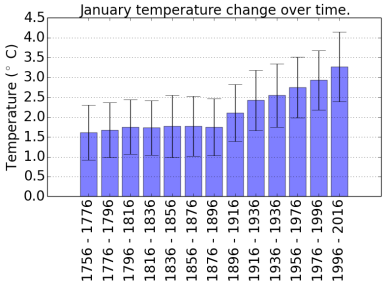
(k) January – SMC: 1956 - 1976



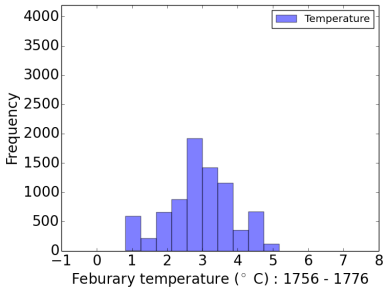
(l) January – SMC: 1976 - 1996



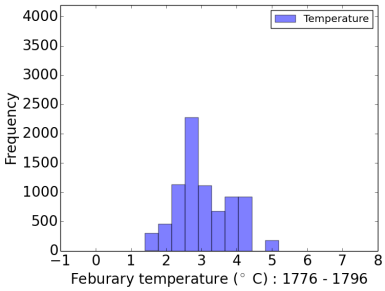
(m) January – SMC: 1996 - 2016



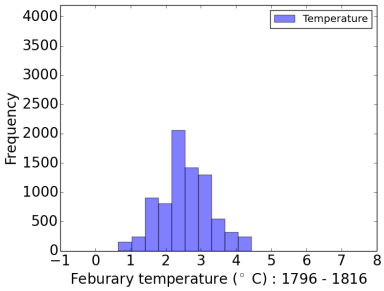
(n) January – SMC: Total



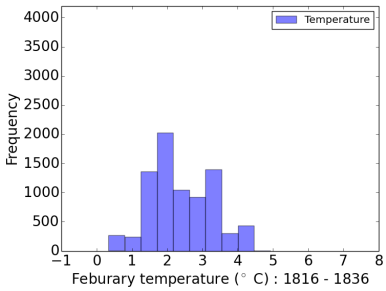
(a) February – SMC: 1756 - 1776



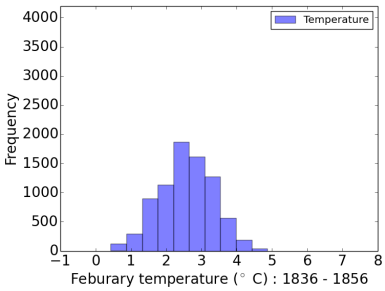
(b) February – SMC: 1776 - 1796



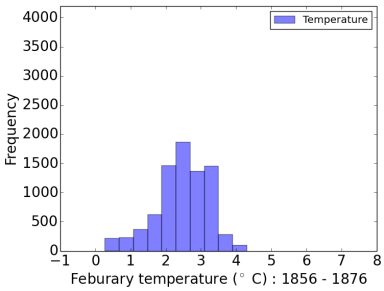
(c) February – SMC: 1796 - 1816



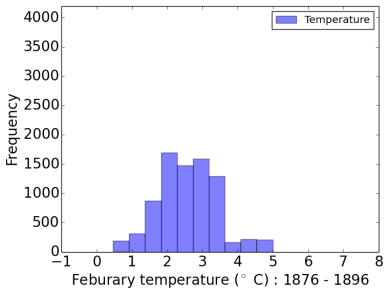
(d) February – SMC: 1816 - 1836



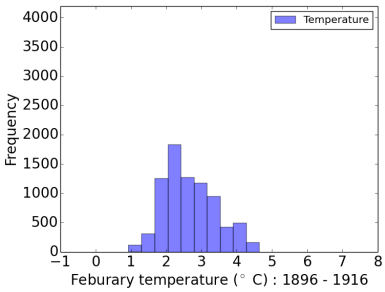
(e) February – SMC: 1836 - 1856



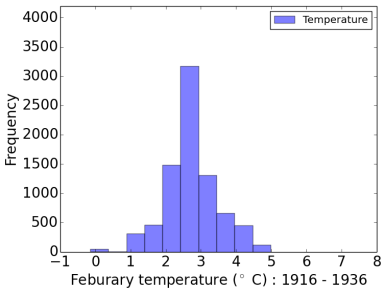
(f) February – SMC: 1856 - 1876



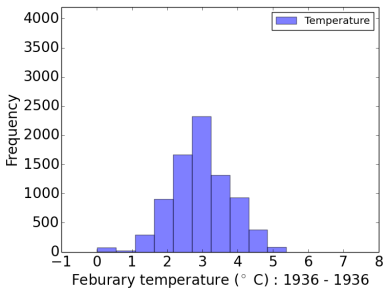
(g) February – SMC: 1876 - 1896



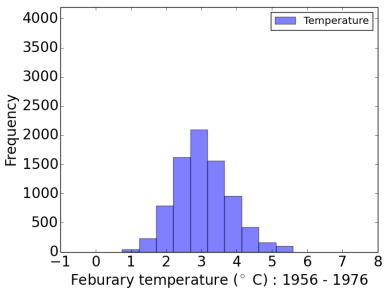
(h) February – SMC: 1896 - 1916



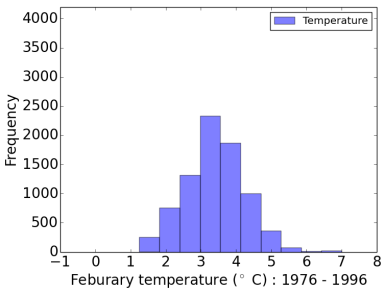
(i) February – SMC: 1916 - 1936



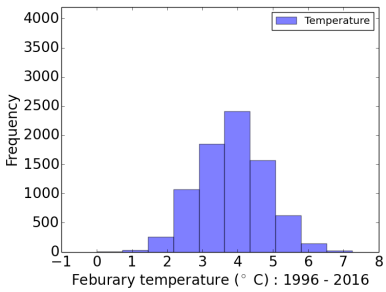
(j) February – SMC: 1936 - 1936



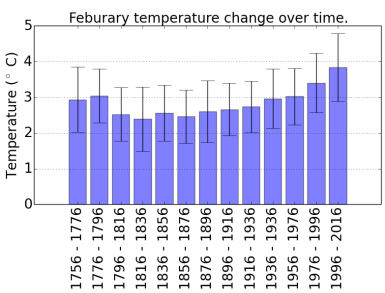
(k) February – SMC: 1956 - 1976



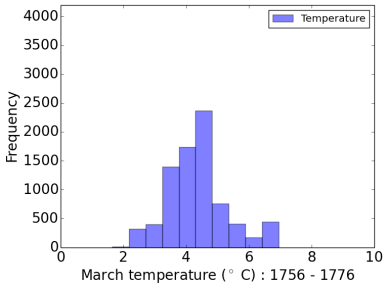
(l) February – SMC: 1976 - 1996



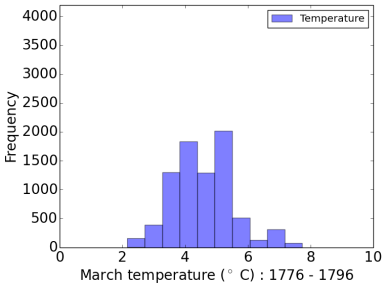
(m) February – SMC: 1996 - 2016



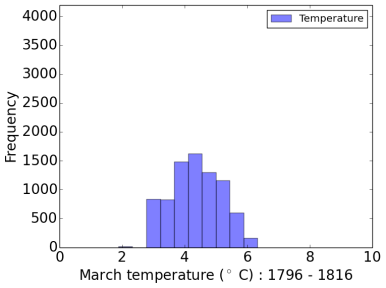
(n) February – SMC: Total



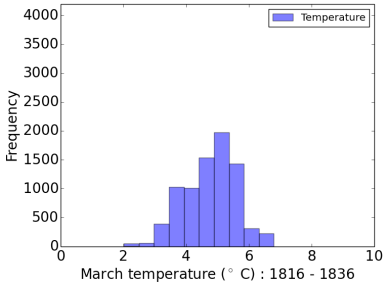
(a) March – SMC: 1756 - 1776



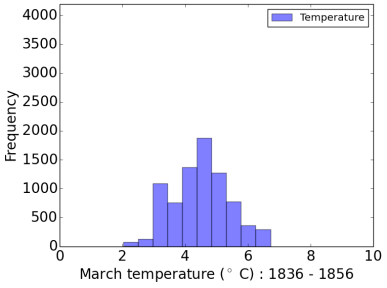
(b) March – SMC: 1776 - 1796



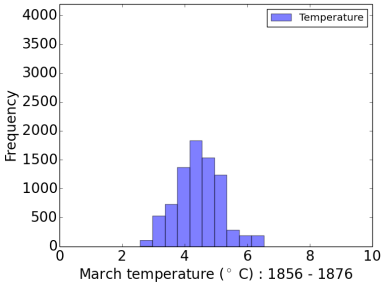
(c) March – SMC: 1796 - 1816



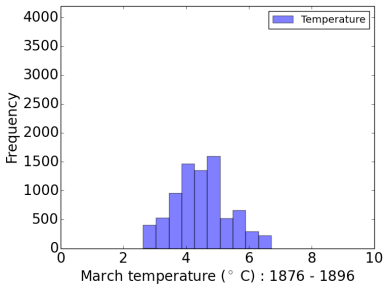
(d) March – SMC: 1816 - 1836



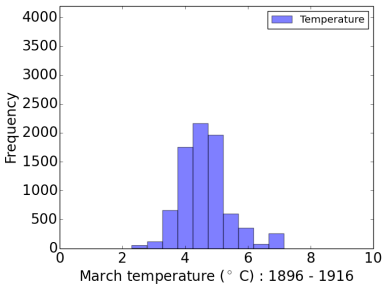
(e) March – SMC: 1836 - 1856



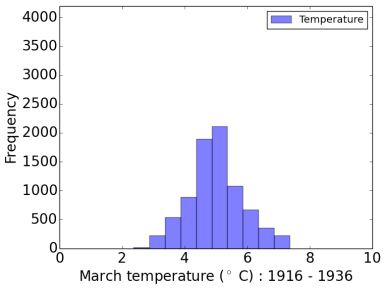
(f) March – SMC: 1856 - 1876



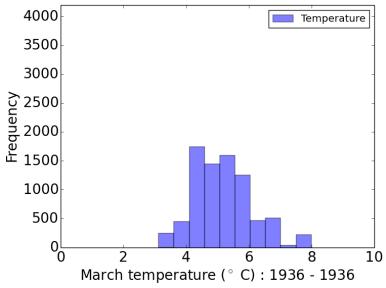
(g) March – SMC: 1876 - 1896



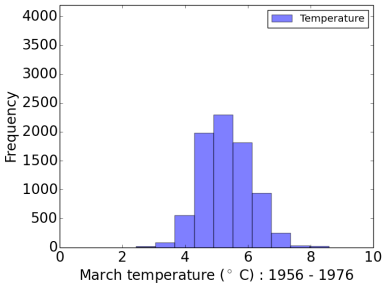
(h) March – SMC: 1896 - 1916



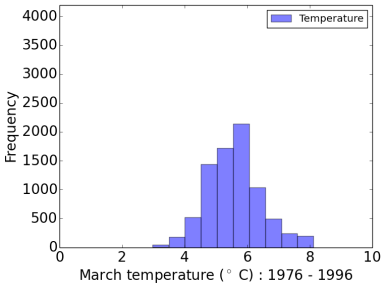
(i) March – SMC: 1916 - 1936



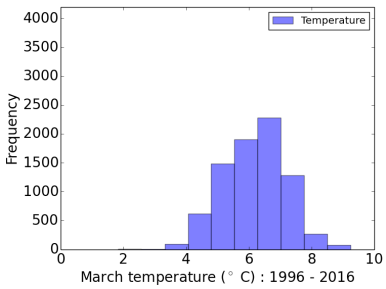
(j) March – SMC: 1936 - 1936



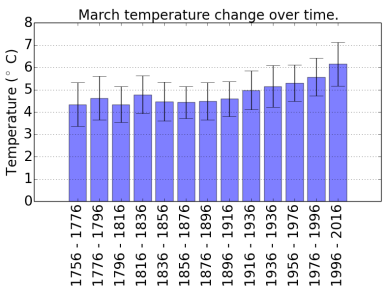
(k) March – SMC: 1956 - 1976



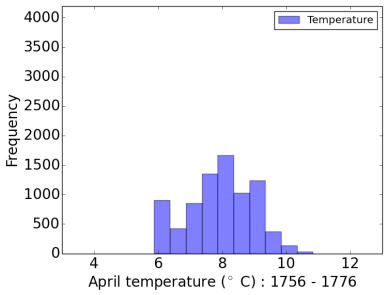
(l) March – SMC: 1976 - 1996



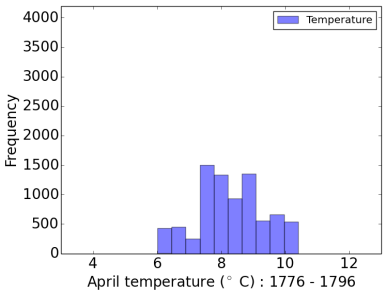
(m) March – SMC: 1996 - 2016



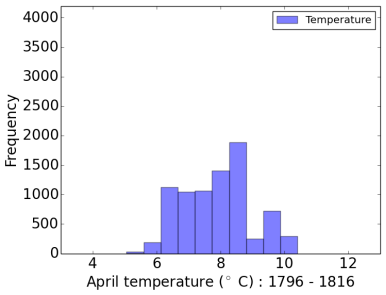
(n) March – SMC: Total



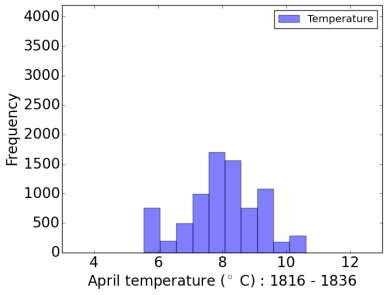
(a) April – SMC: 1756 - 1776



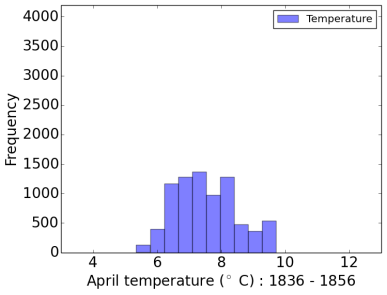
(b) April – SMC: 1776 - 1796



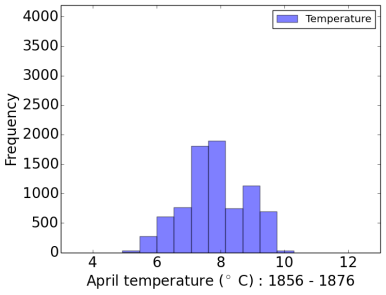
(c) April – SMC: 1796 - 1816



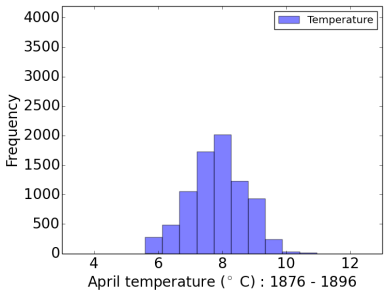
(d) April – SMC: 1816 - 1836



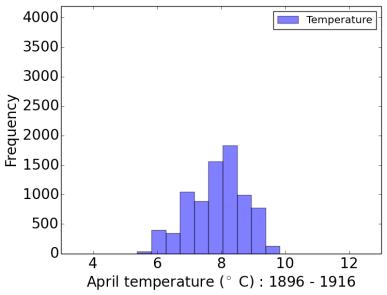
(e) April – SMC: 1836 - 1856



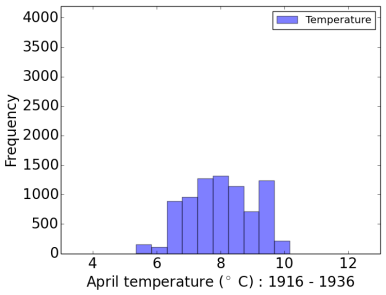
(f) April – SMC: 1856 - 1876



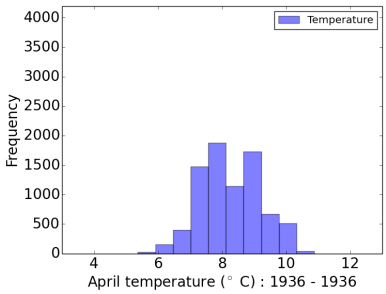
(g) April – SMC: 1876 - 1896



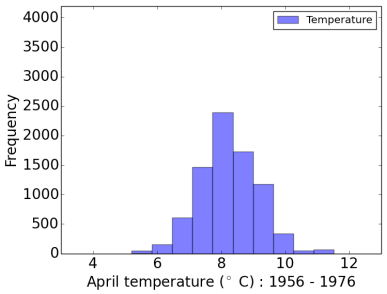
(h) April – SMC: 1896 - 1916



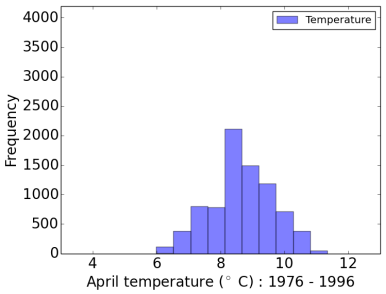
(i) April – SMC: 1916 - 1936



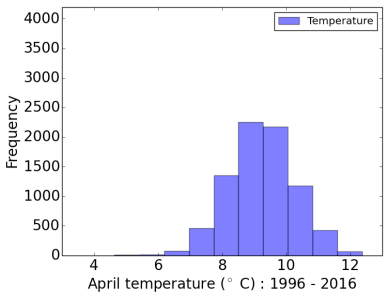
(j) April – SMC: 1936 - 1936



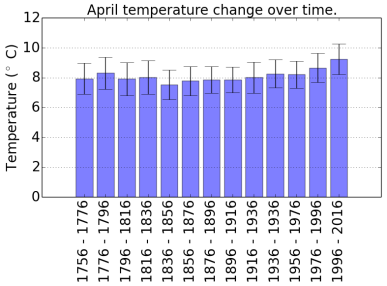
(k) April – SMC: 1956 - 1976



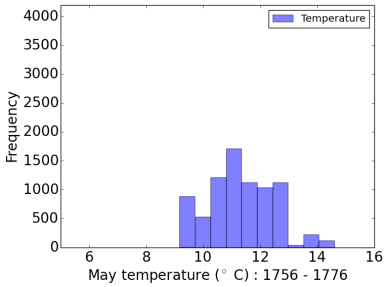
(l) April – SMC: 1976 - 1996



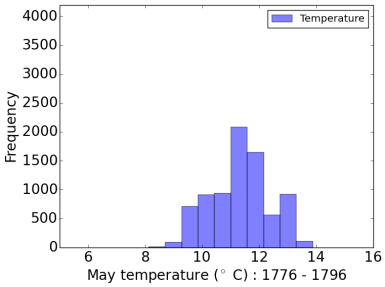
(m) April – SMC: 1996 - 2016



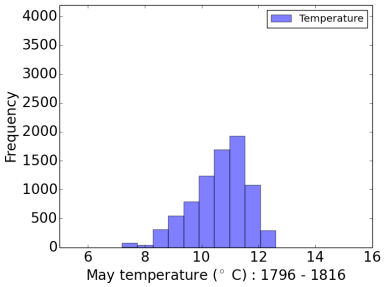
(n) April – SMC: Total



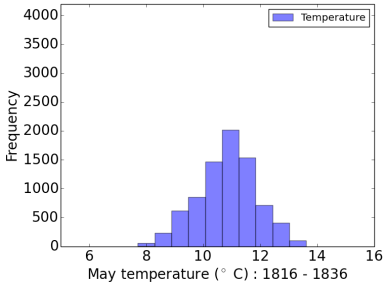
(a) May – SMC: 1756 - 1776



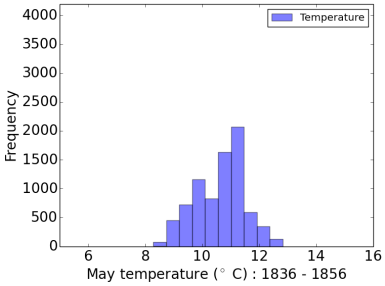
(b) May – SMC: 1776 - 1796



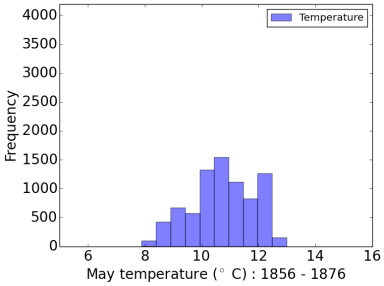
(c) May – SMC: 1796 - 1816



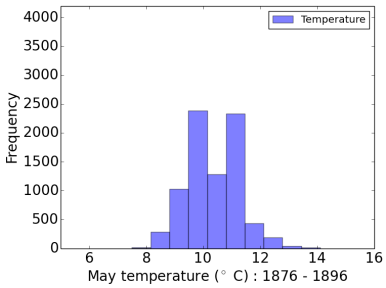
(d) May – SMC: 1816 - 1836



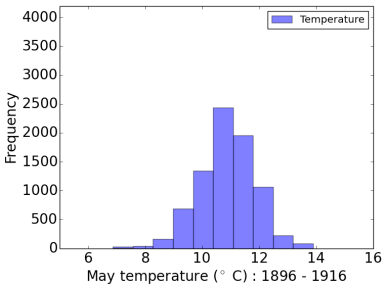
(e) May – SMC: 1836 - 1856



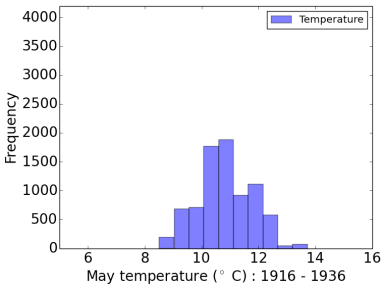
(f) May – SMC: 1856 - 1876



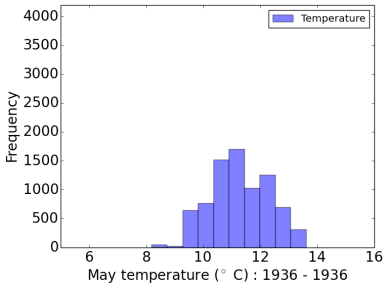
(g) May – SMC: 1876 - 1896



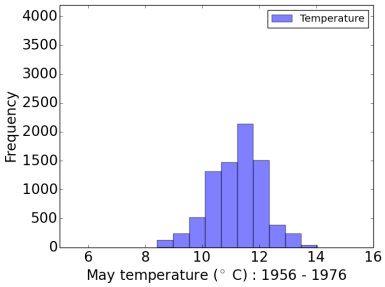
(h) May – SMC: 1896 - 1916



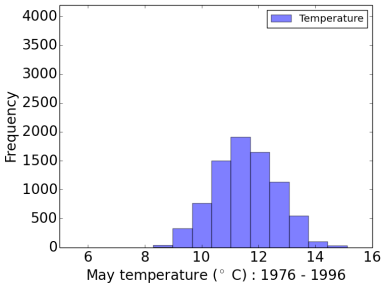
(i) May – SMC: 1916 - 1936



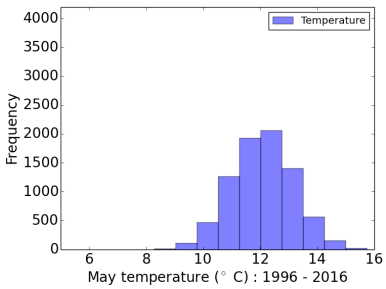
(j) May – SMC: 1936 - 1936



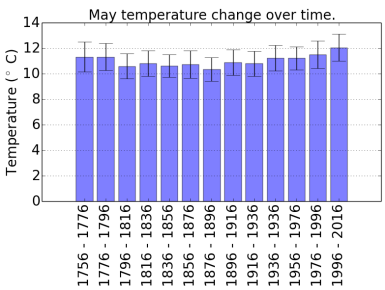
(k) May – SMC: 1956 - 1976



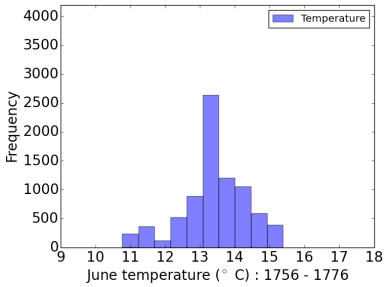
(l) May – SMC: 1976 - 1996



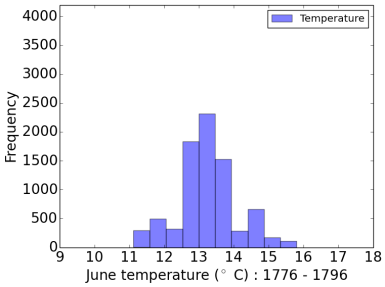
(m) May – SMC: 1996 - 2016



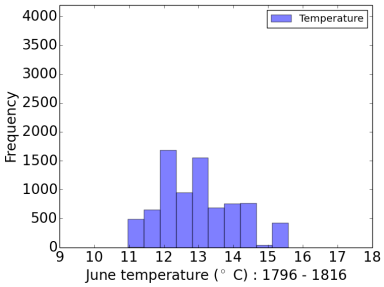
(n) May – SMC: Total



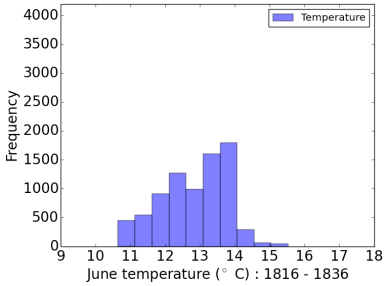
(a) June – SMC: 1756 - 1776



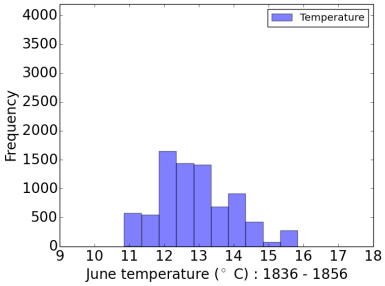
(b) June – SMC: 1776 - 1796



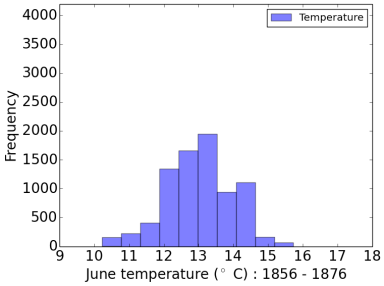
(c) June – SMC: 1796 - 1816



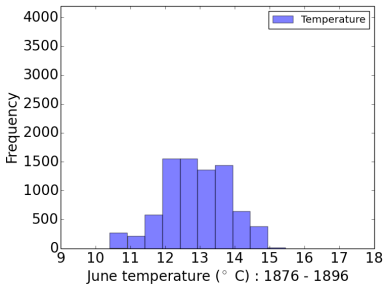
(d) June – SMC: 1816 - 1836



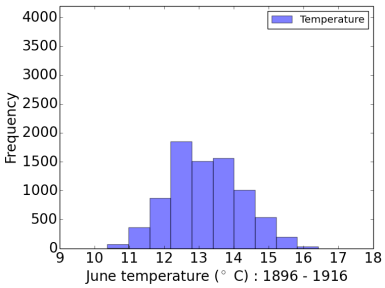
(e) June – SMC: 1836 - 1856



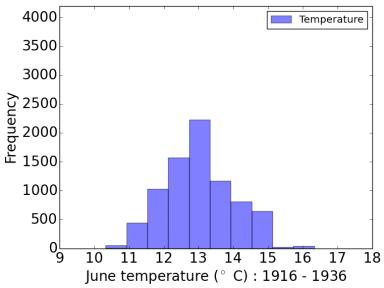
(f) June – SMC: 1856 - 1876



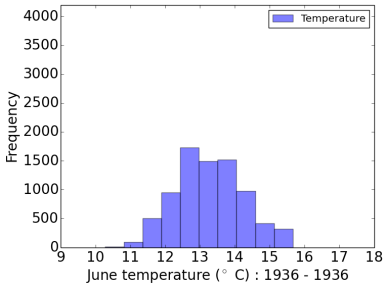
(g) June – SMC: 1876 - 1896



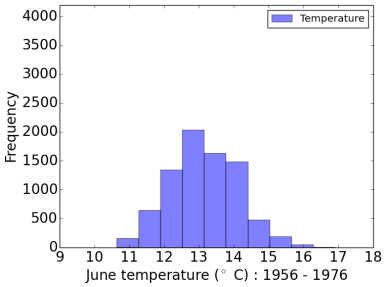
(h) June – SMC: 1896 - 1916



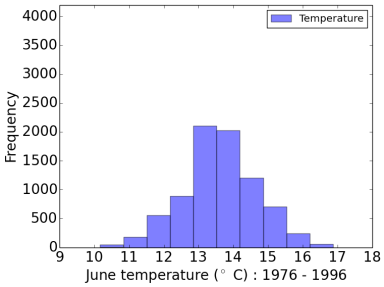
(i) June – SMC: 1916 - 1936



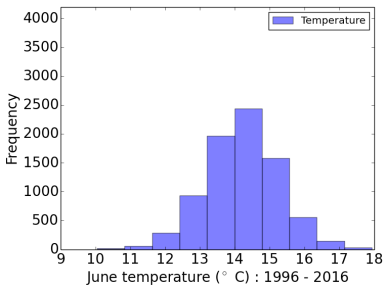
(j) June – SMC: 1936 - 1936



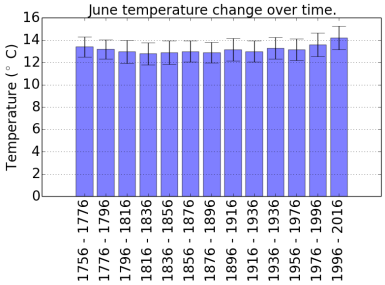
(k) June – SMC: 1956 - 1976



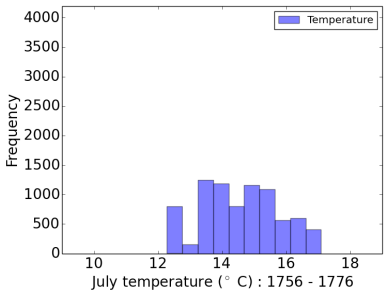
(l) June – SMC: 1976 - 1996



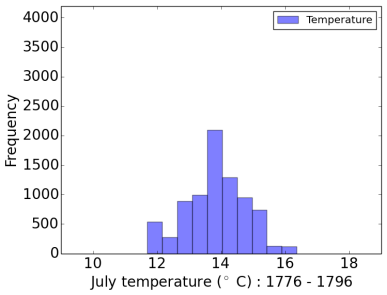
(m) June – SMC: 1996 - 2016



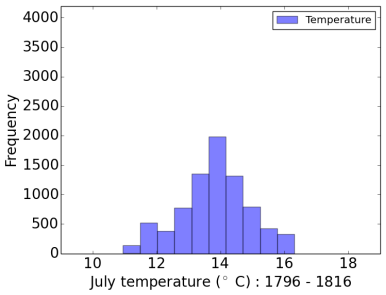
(n) June – SMC: Total



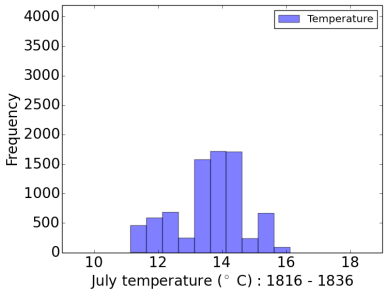
(a) July – SMC: 1756 - 1776



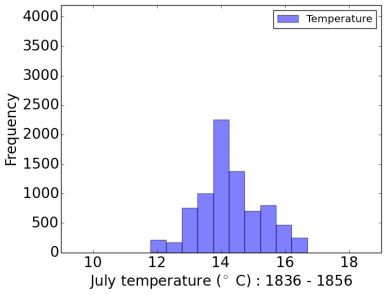
(b) July – SMC: 1776 - 1796



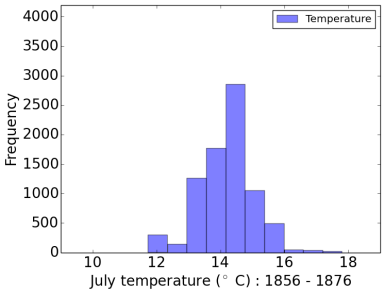
(c) July – SMC: 1796 - 1816



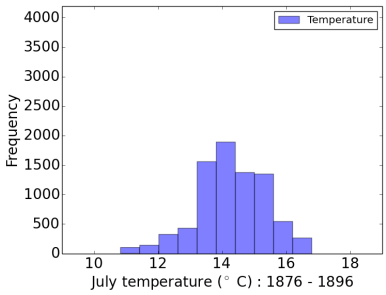
(d) July – SMC: 1816 - 1836



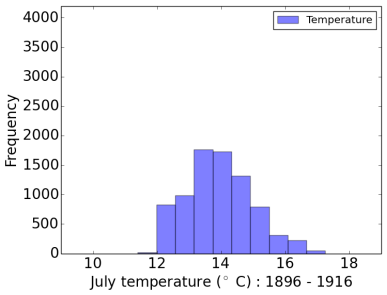
(e) July – SMC: 1836 - 1856



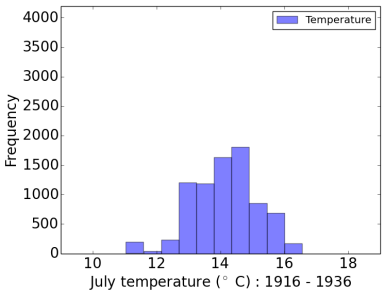
(f) July – SMC: 1856 - 1876



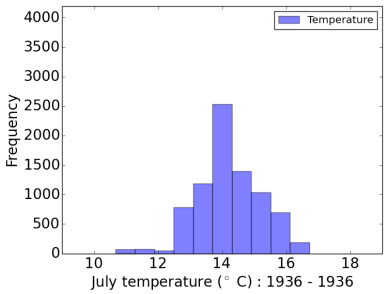
(g) July – SMC: 1876 - 1896



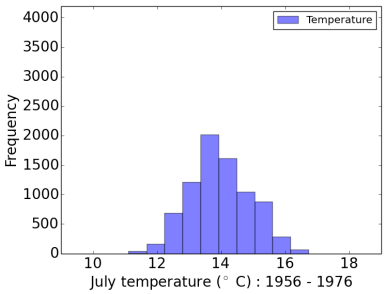
(h) July – SMC: 1896 - 1916



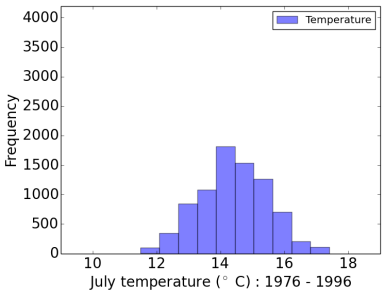
(i) July – SMC: 1916 - 1936



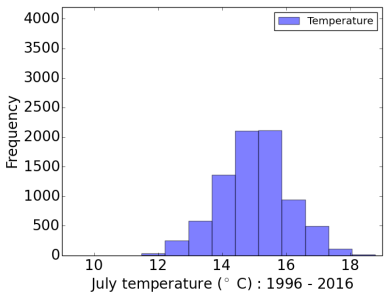
(j) July – SMC: 1936 - 1936



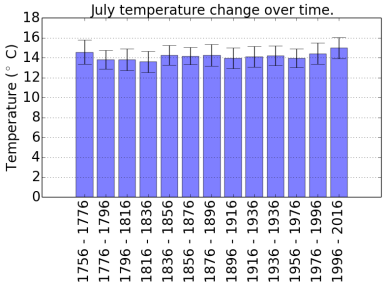
(k) July – SMC: 1956 - 1976



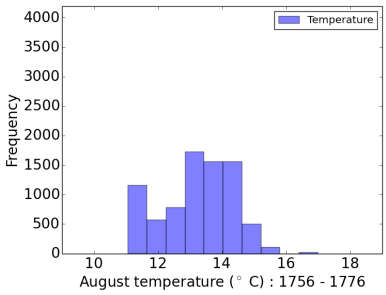
(l) July – SMC: 1976 - 1996



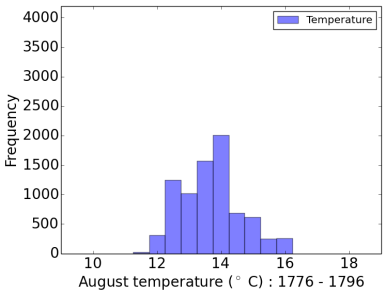
(m) July – SMC: 1996 - 2016



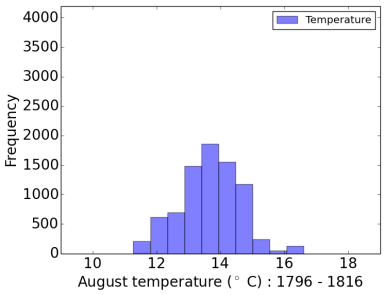
(n) July – SMC: Total



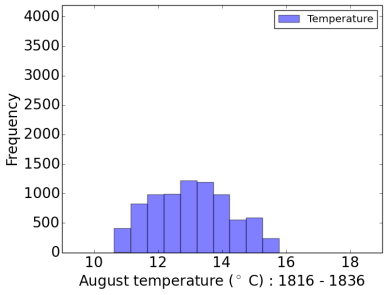
(a) August – SMC: 1756 - 1776



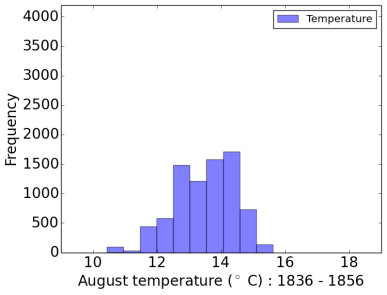
(b) August – SMC: 1776 - 1796



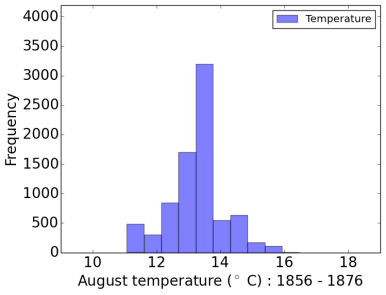
(c) August – SMC: 1796 - 1816



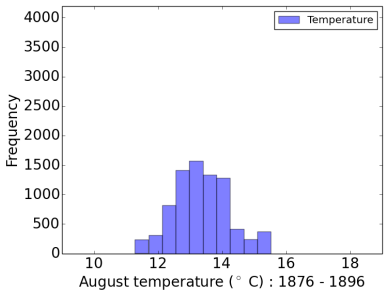
(d) August – SMC: 1816 - 1836



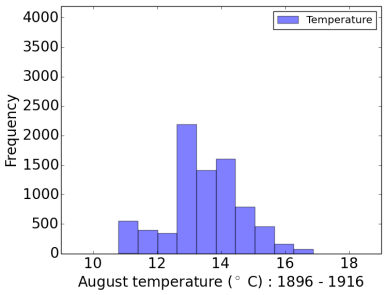
(e) August – SMC: 1836 - 1856



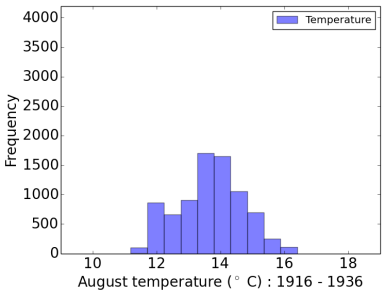
(f) August – SMC: 1856 - 1876



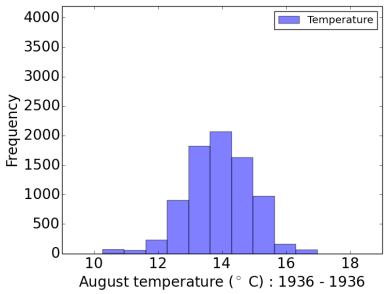
(g) August – SMC: 1876 - 1896



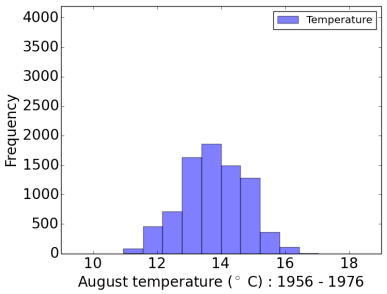
(h) August – SMC: 1896 - 1916



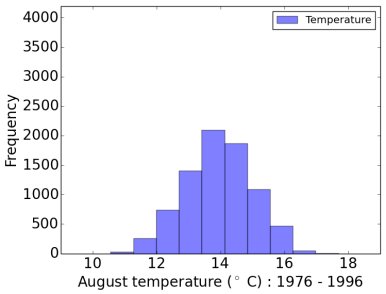
(i) August – SMC: 1916 - 1936



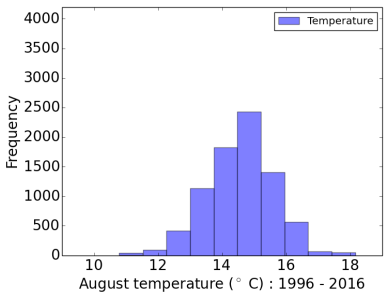
(j) August – SMC: 1936 - 1936



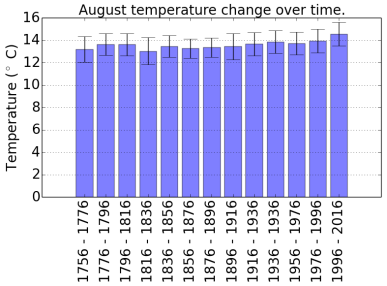
(k) August – SMC: 1956 - 1976



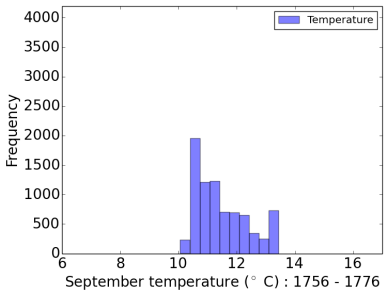
(l) August – SMC: 1976 - 1996



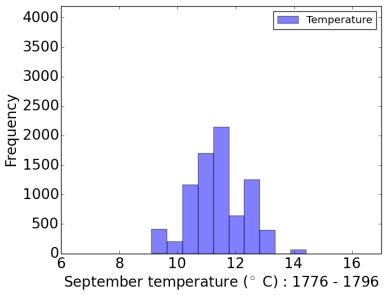
(m) August – SMC: 1996 - 2016



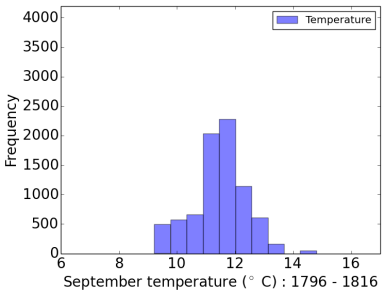
(n) August – SMC: Total



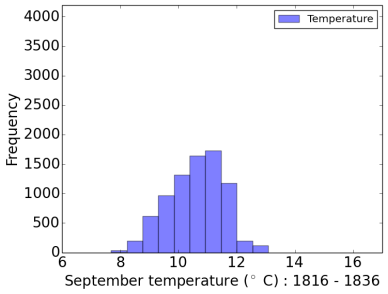
(a) September – SMC: 1756 - 1776



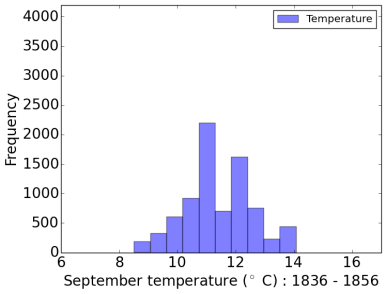
(b) September – SMC: 1776 - 1796



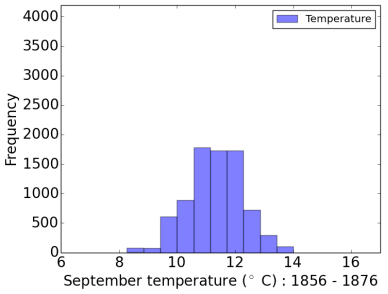
(c) September – SMC: 1796 - 1816



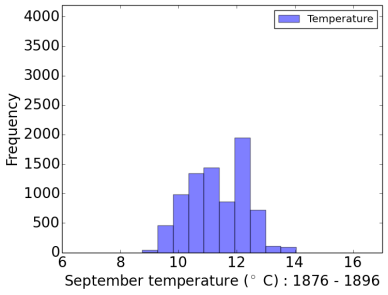
(d) September – SMC: 1816 - 1836



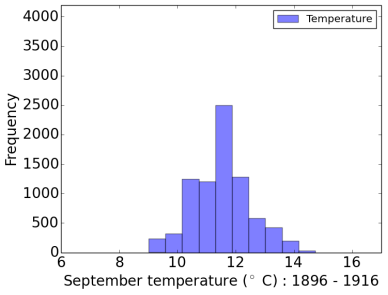
(e) September – SMC: 1836 - 1856



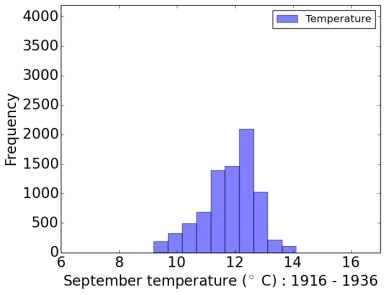
(f) September – SMC: 1856 - 1876



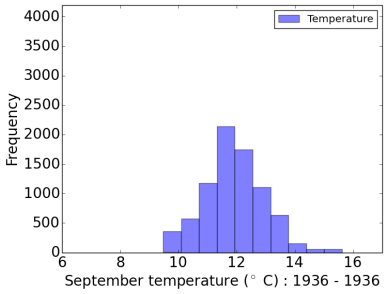
(g) September – SMC: 1876 - 1896



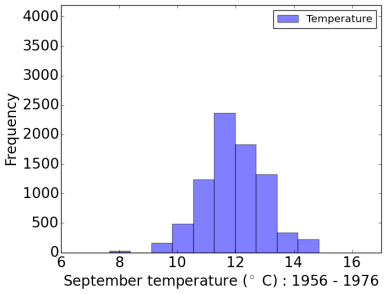
(h) September – SMC: 1896 - 1916



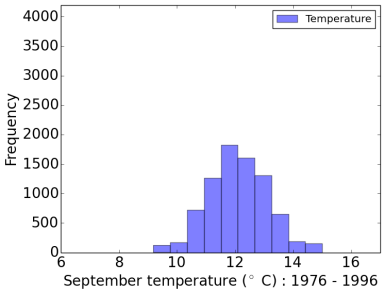
(i) September – SMC: 1916 - 1936



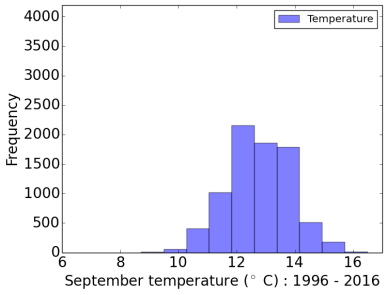
(j) September – SMC: 1936 - 1936



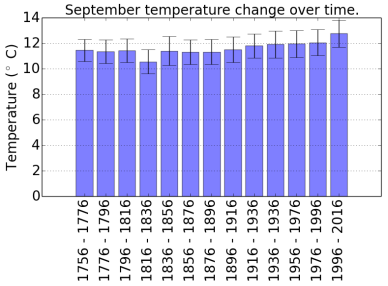
(k) September – SMC: 1956 - 1976



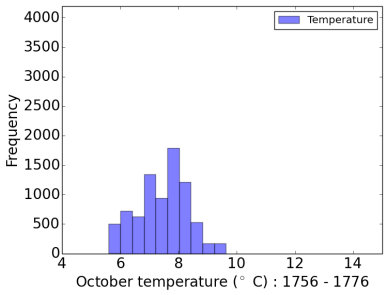
(l) September – SMC: 1976 - 1996



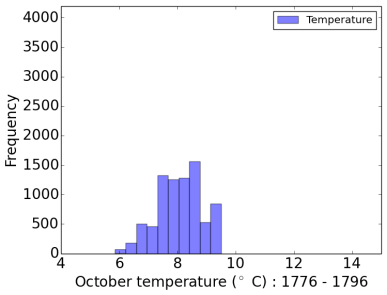
(m) September – SMC: 1996 - 2016



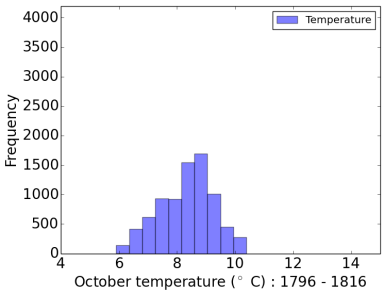
(n) September – SMC: Total



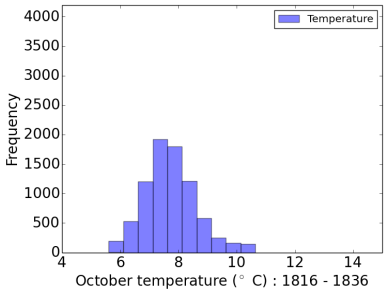
(a) October – SMC: 1756 - 1776



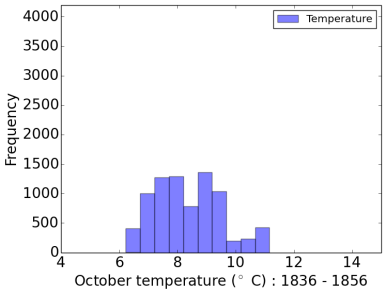
(b) October – SMC: 1776 - 1796



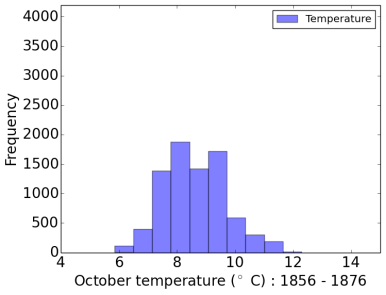
(c) October – SMC: 1796 - 1816



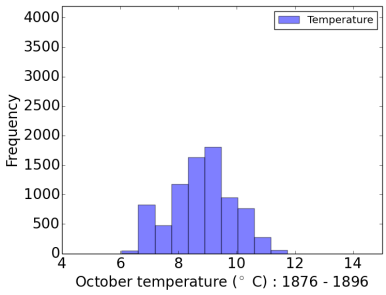
(d) October – SMC: 1816 - 1836



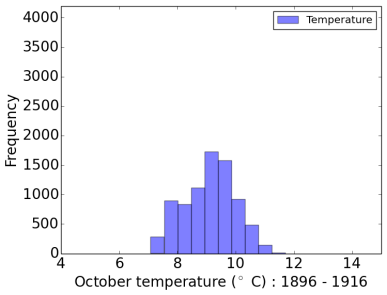
(e) October – SMC: 1836 - 1856



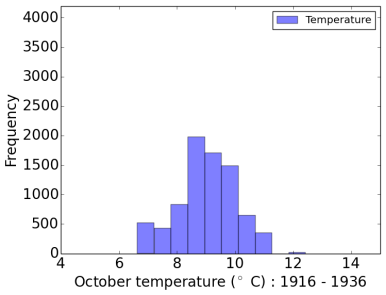
(f) October – SMC: 1856 - 1876



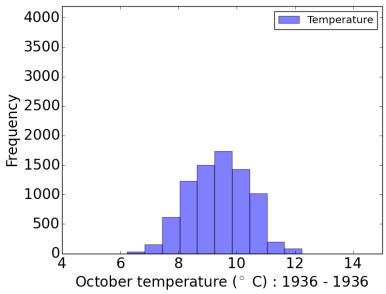
(g) October – SMC: 1876 - 1896



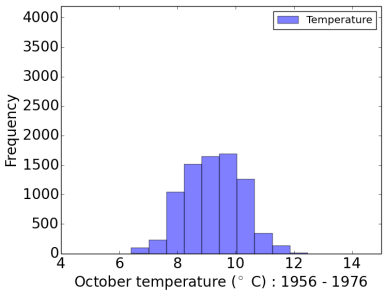
(h) October – SMC: 1896 - 1916



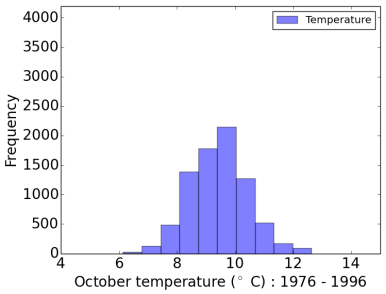
(i) October – SMC: 1916 - 1936



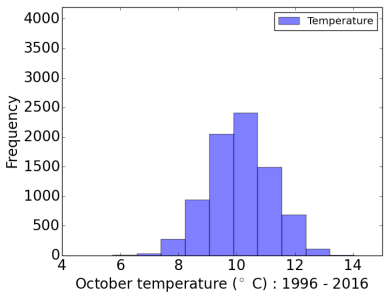
(j) October – SMC: 1936 - 1936



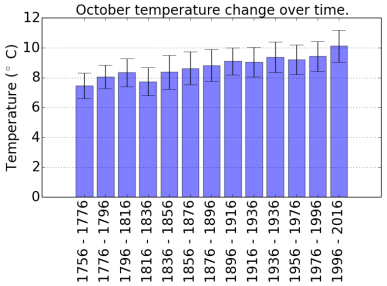
(k) October – SMC: 1956 - 1976



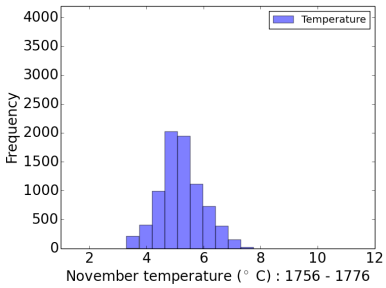
(l) October – SMC: 1976 - 1996



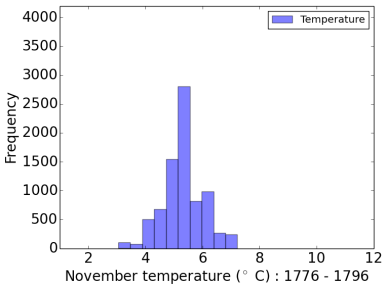
(m) October – SMC: 1996 - 2016



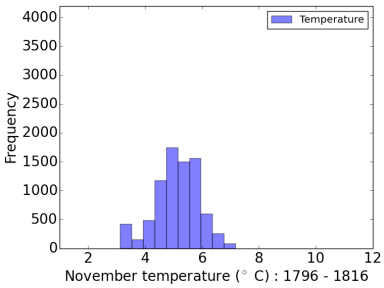
(n) October – SMC: Total



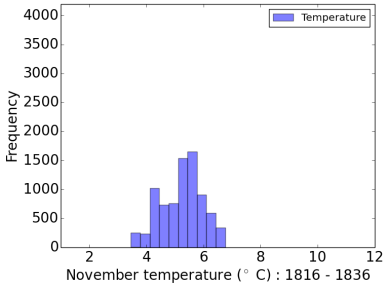
(a) November – SMC: 1756 - 1776



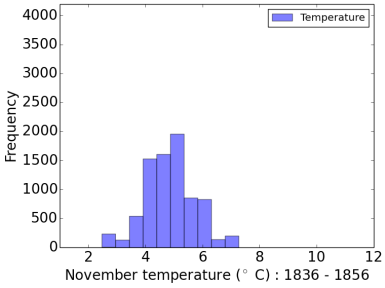
(b) November – SMC: 1776 - 1796



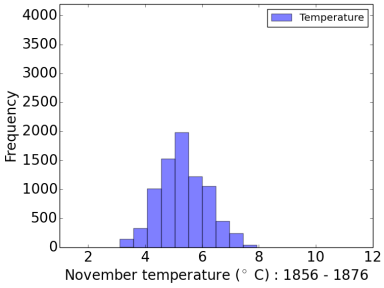
(c) November – SMC: 1796 - 1816



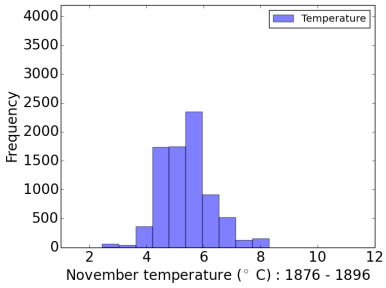
(d) November – SMC: 1816 - 1836



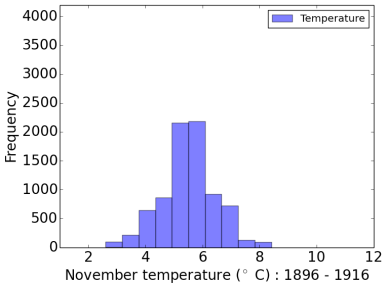
(e) November – SMC: 1836 - 1856



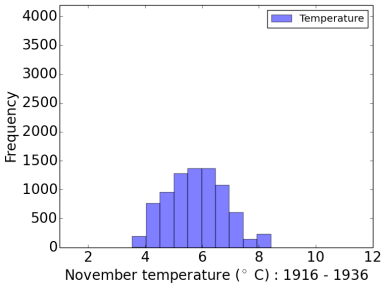
(f) November – SMC: 1856 - 1876



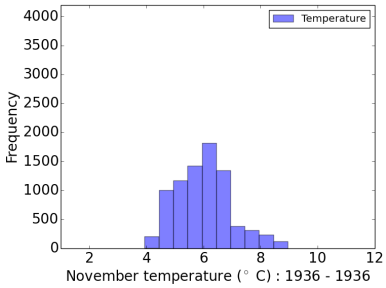
(g) November – SMC: 1876 - 1896



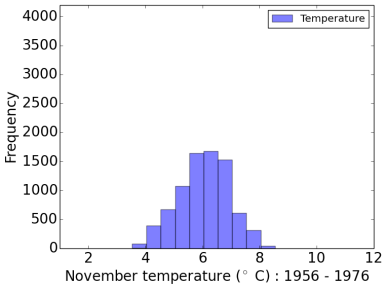
(h) November – SMC: 1896 - 1916



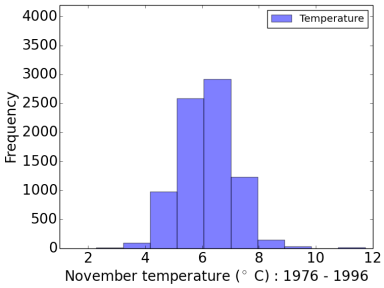
(i) November – SMC: 1916 - 1936



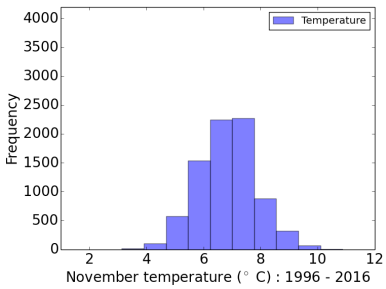
(j) November – SMC: 1936 - 1936



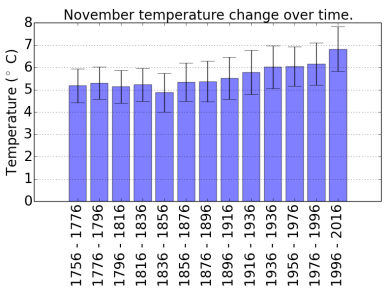
(k) November – SMC: 1956 - 1976



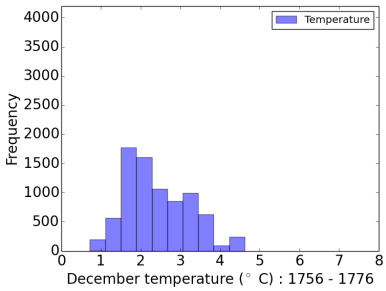
(l) November – SMC: 1976 - 1996



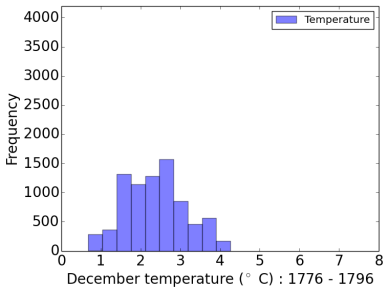
(m) November – SMC: 1996 - 2016



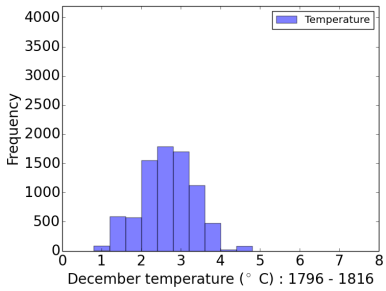
(n) November – SMC: Total



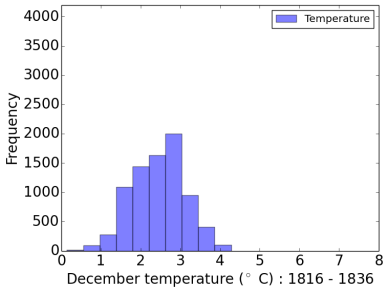
(a) December – SMC: 1756 - 1776



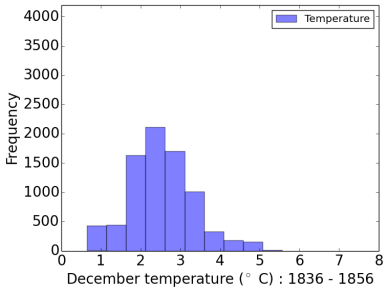
(b) December – SMC: 1776 - 1796



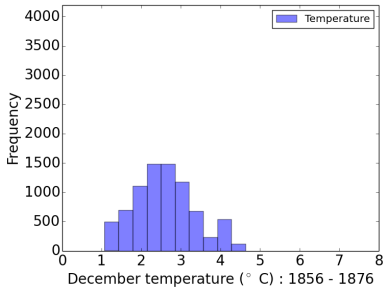
(c) December – SMC: 1796 - 1816



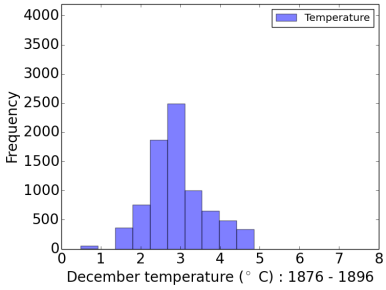
(d) December – SMC: 1816 - 1836



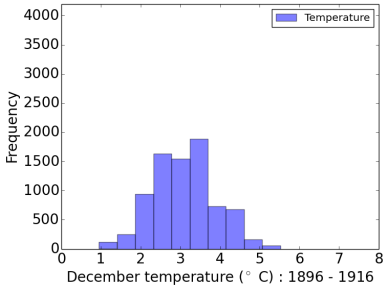
(e) December – SMC: 1836 - 1856



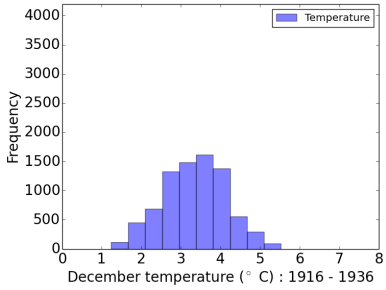
(f) December – SMC: 1856 - 1876



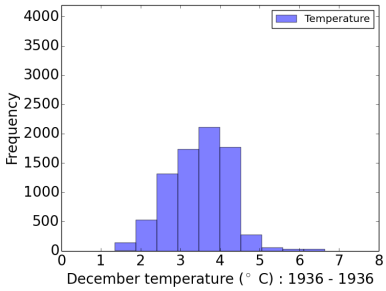
(g) December – SMC: 1876 - 1896



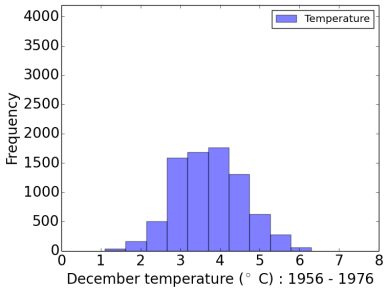
(h) December – SMC: 1896 - 1916



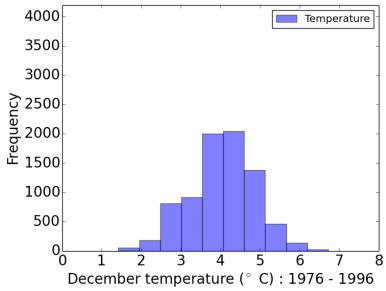
(i) December – SMC: 1916 - 1936



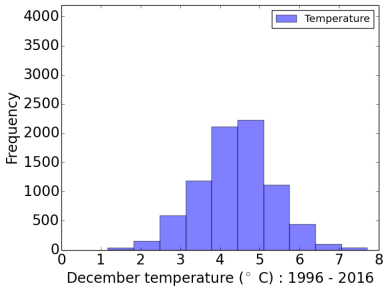
(j) December – SMC: 1936 - 1936



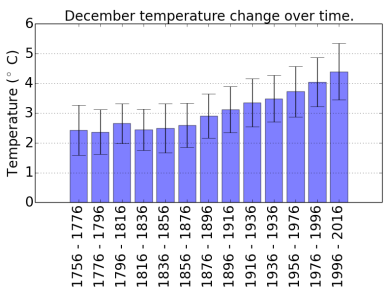
(k) December – SMC: 1956 - 1976



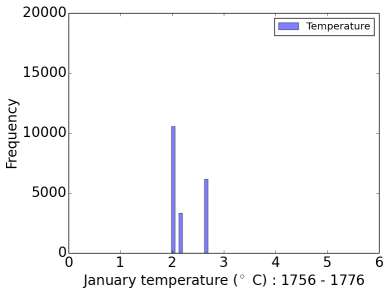
(l) December – SMC: 1976 - 1996



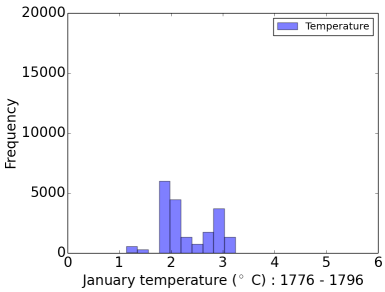
(m) December – SMC: 1996 - 2016



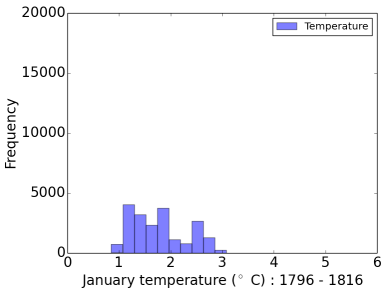
(n) December – SMC: Total



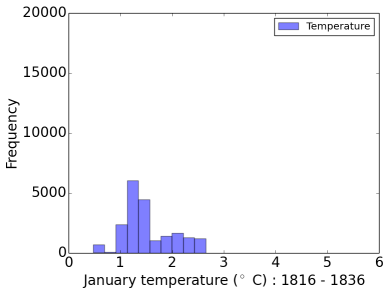
(a) January – TMCMC: 1756 - 1776



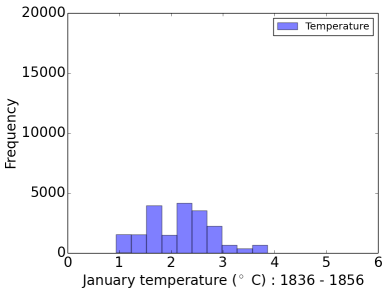
(b) January – TMCMC: 1776 - 1796



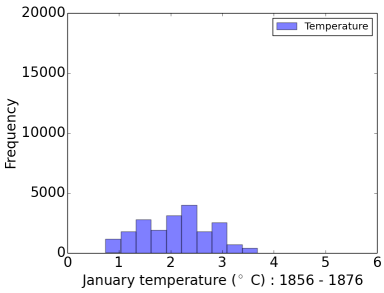
(c) January – TMCMC: 1796 - 1816



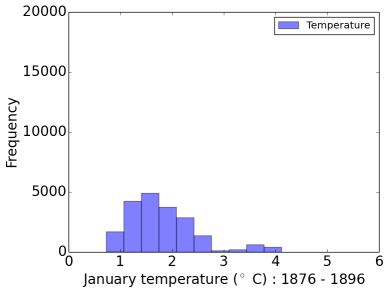
(d) January – TMCMC: 1816 - 1836



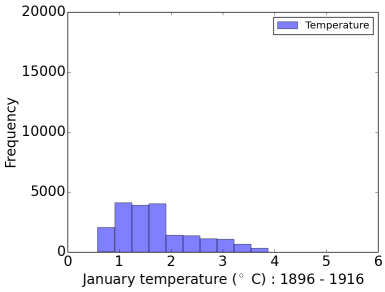
(e) January – TMCMC: 1836 - 1856



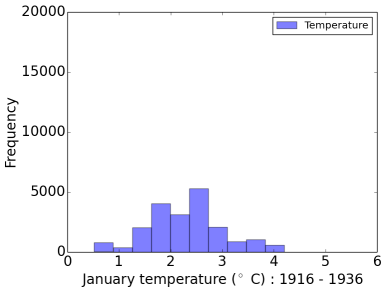
(f) January – TMCMC: 1856 - 1876



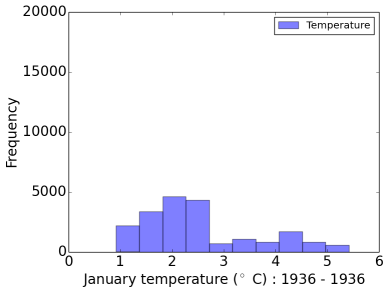
(g) January – TMCMC: 1876 - 1896



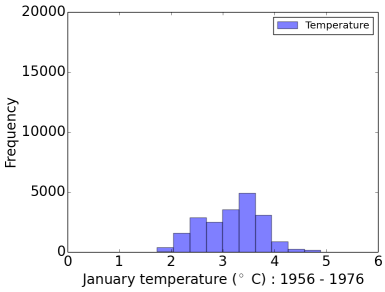
(h) January – TMCMC: 1896 - 1916



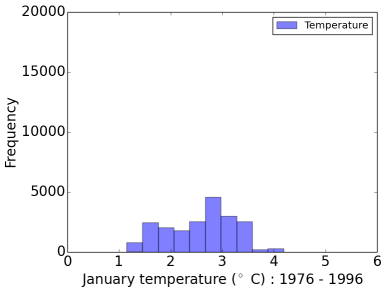
(i) January – TMCMC: 1916 - 1936



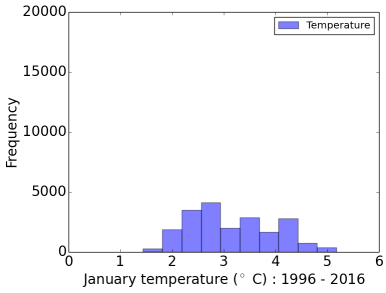
(j) January – TMCMC: 1936 - 1936



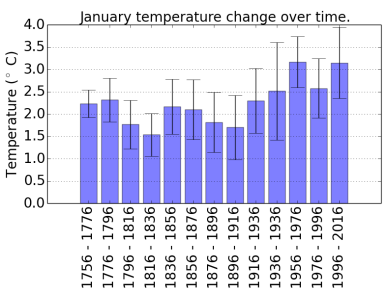
(k) January – TMCMC: 1956 - 1976



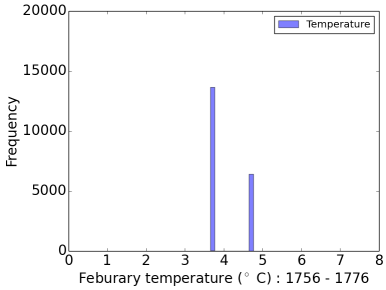
(l) January – TMCMC: 1976 - 1996



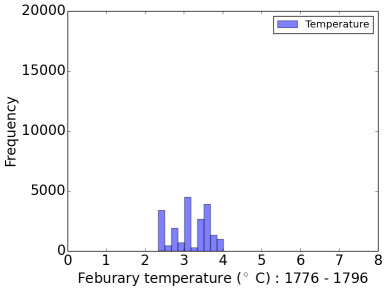
(m) January – TMCMC: 1996 - 2016



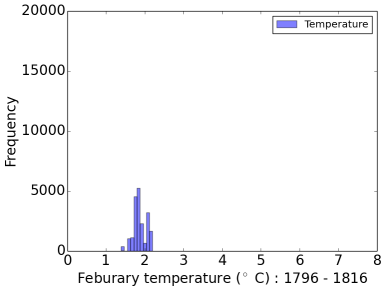
(n) January – TMCMC: Total



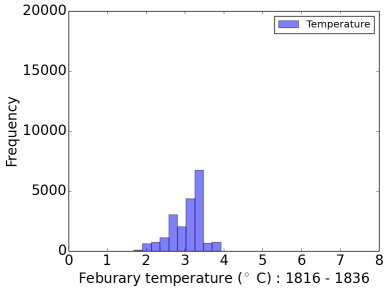
(a) Feburary – TMCMC: 1756 - 1776



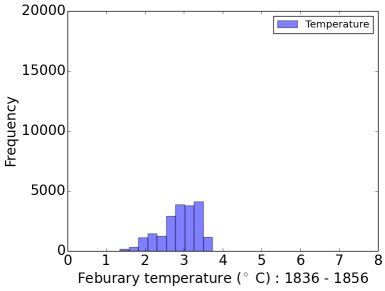
(b) Feburary – TMCMC: 1776 - 1796



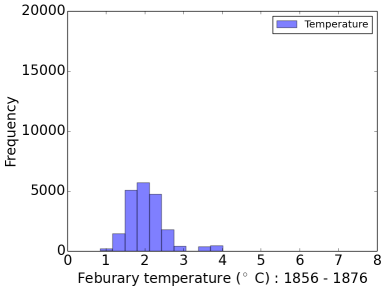
(c) Feburary – TMCMC: 1796 - 1816



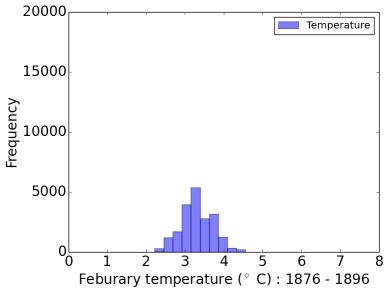
(d) Feburary – TMCMC: 1816 - 1836



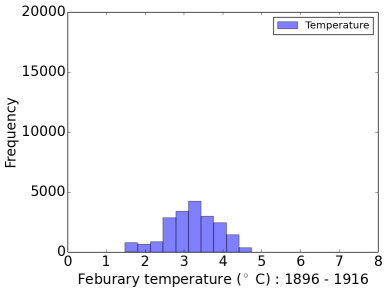
(e) Feburary – TMCMC: 1836 - 1856



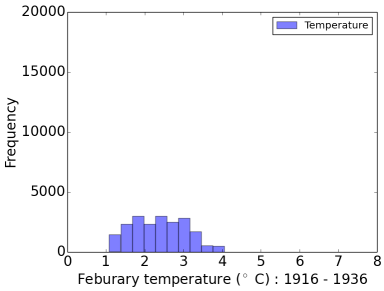
(f) Feburary – TMCMC: 1856 - 1876



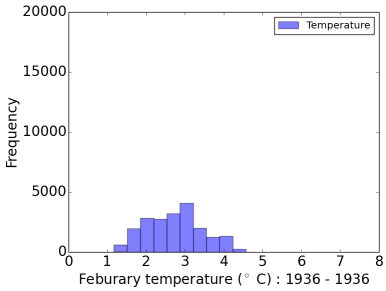
(g) Feburary – TMCMC: 1876 - 1896



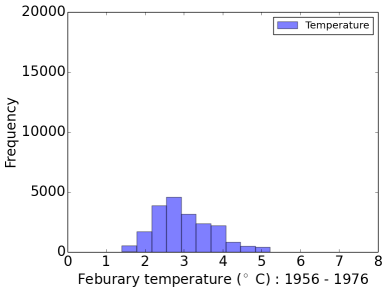
(h) Feburary – TMCMC: 1896 - 1916



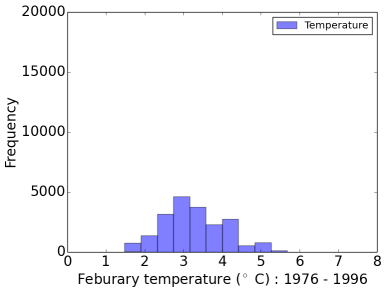
(i) Feburary – TMCMC: 1916 - 1936



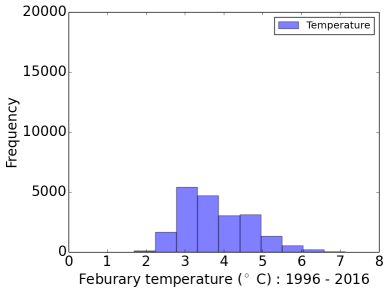
(j) Feburary – TMCMC: 1936 - 1936



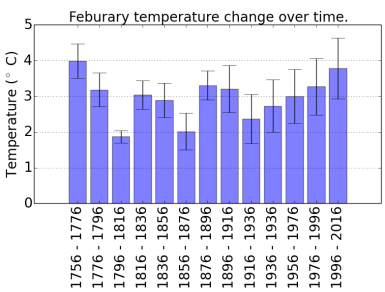
(k) Feburary – TMCMC: 1956 - 1976



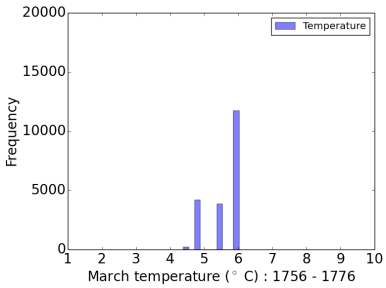
(l) Feburary – TMCMC: 1976 - 1996



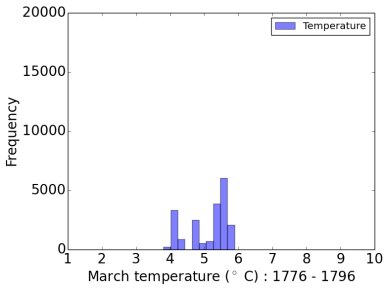
(m) Feburary – TMCMC: 1996 - 2016



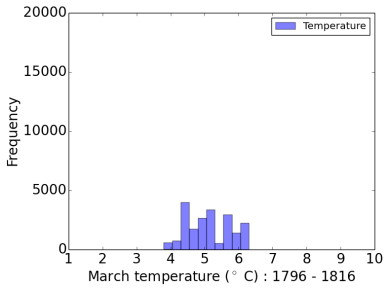
(n) Feburary – TMCMC: Total



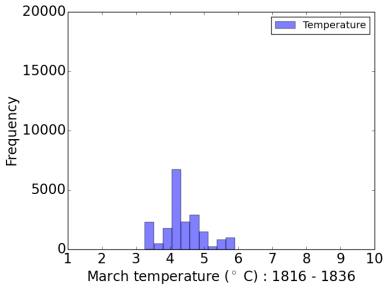
(a) March – TMCMC: 1756 - 1776



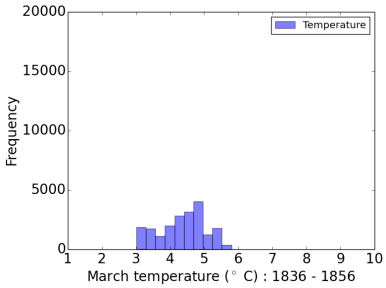
(b) March – TMCMC: 1776 - 1796



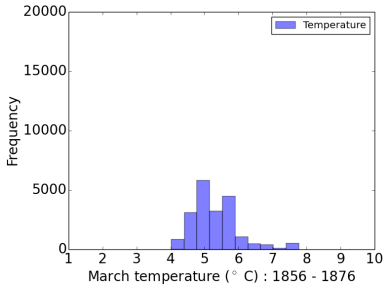
(c) March – TMCMC: 1796 - 1816



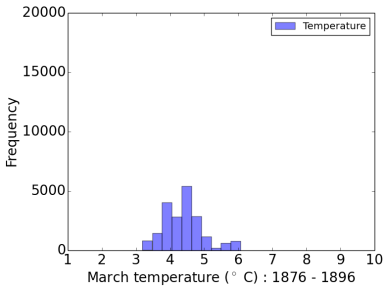
(d) March – TMCMC: 1816 - 1836



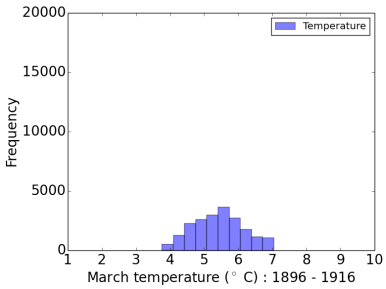
(e) March – TMCMC: 1836 - 1856



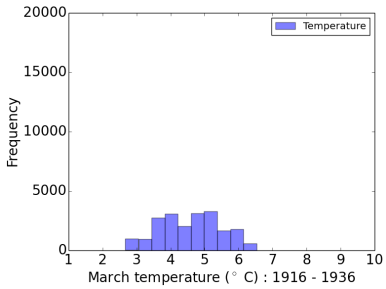
(f) March – TMCMC: 1856 - 1876



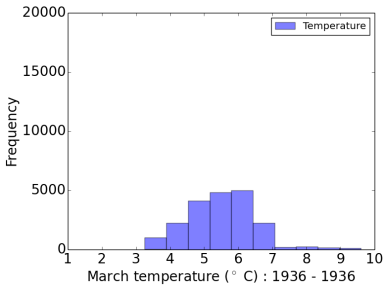
(g) March – TMCMC: 1876 - 1896



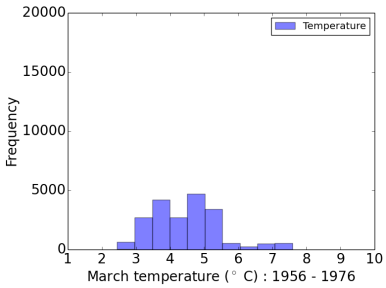
(h) March – TMCMC: 1896 - 1916



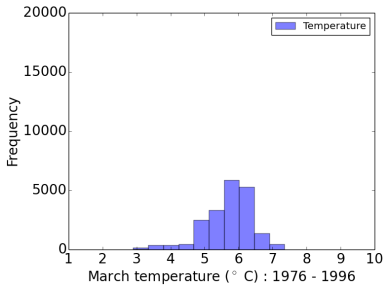
(i) March – TMCMC: 1916 - 1936



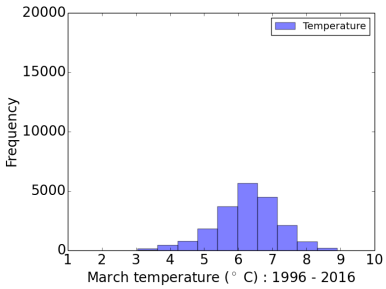
(j) March – TMCMC: 1936 - 1936



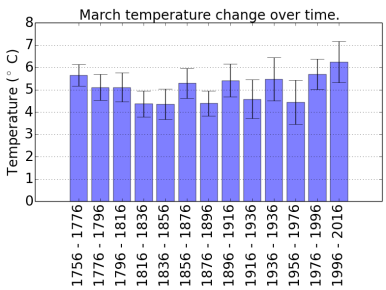
(k) March – TMCMC: 1956 - 1976



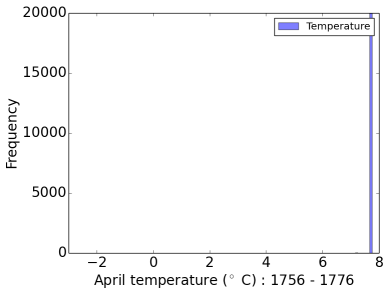
(l) March – TMCMC: 1976 - 1996



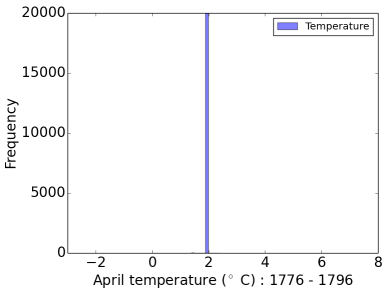
(m) March – TMCMC: 1996 - 2016



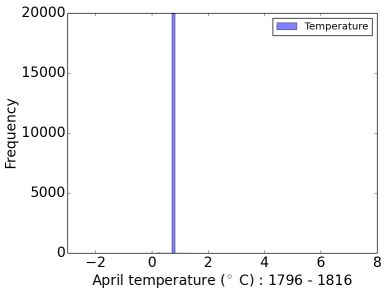
(n) March – TMCMC: Total



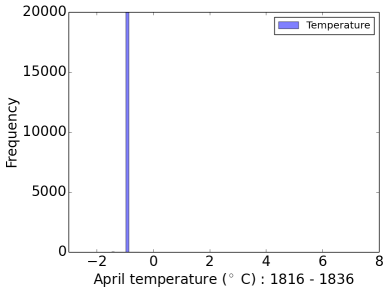
(a) April – TMCMC: 1756 - 1776



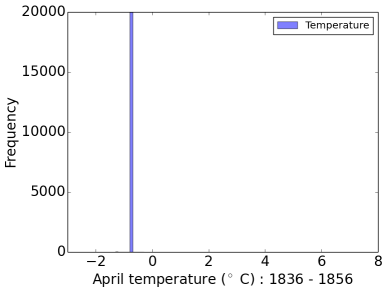
(b) April – TMCMC: 1776 - 1796



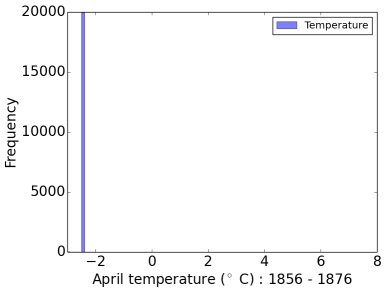
(c) April – TMCMC: 1796 - 1816



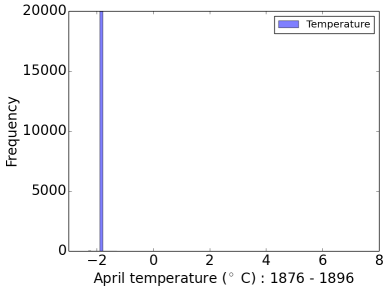
(d) April – TMCMC: 1816 - 1836



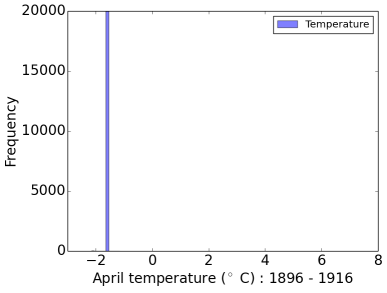
(e) April – TMCMC: 1836 - 1856



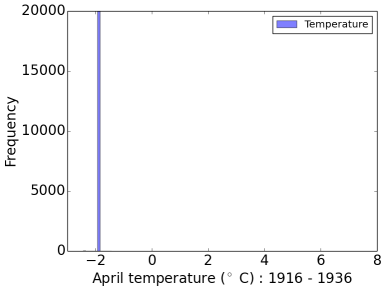
(f) April – TMCMC: 1856 - 1876



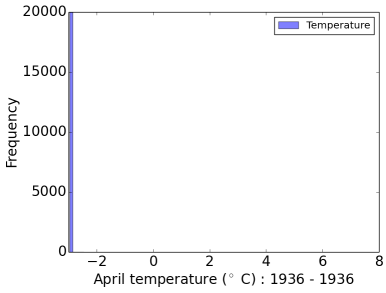
(g) April – TMCMC: 1876 - 1896



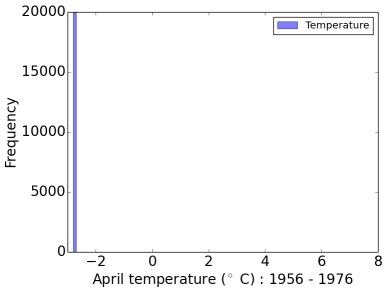
(h) April – TMCMC: 1896 - 1916



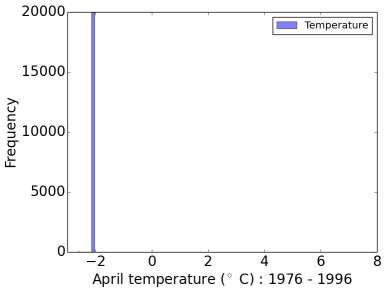
(i) April – TMCMC: 1916 - 1936



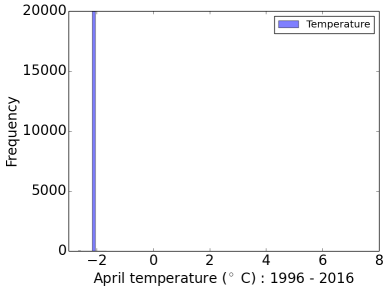
(j) April – TMCMC: 1936 - 1936



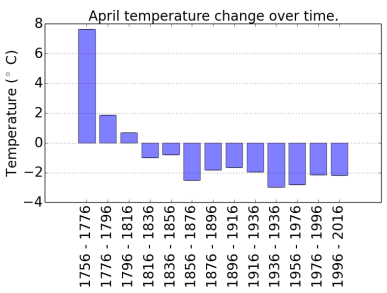
(k) April – TMCMC: 1956 - 1976



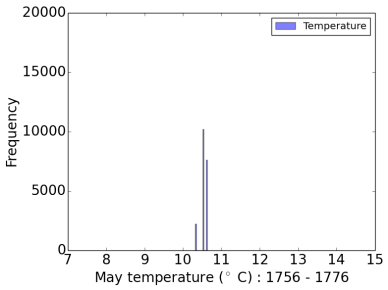
(l) April – TMCMC: 1976 - 1996



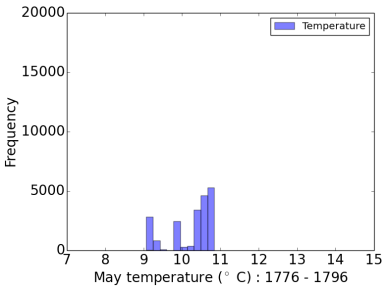
(m) April – TMCMC: 1996 - 2016



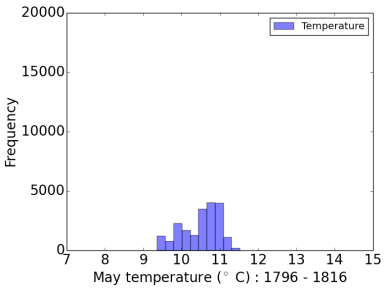
(n) April – TMCMC: Total



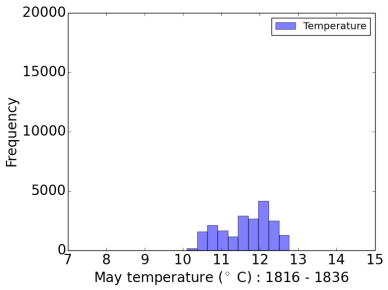
(a) May – TMCMC: 1756 - 1776



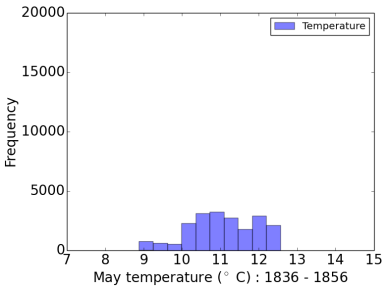
(b) May – TMCMC: 1776 - 1796



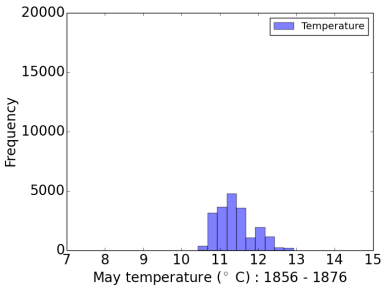
(c) May – TMCMC: 1796 - 1816



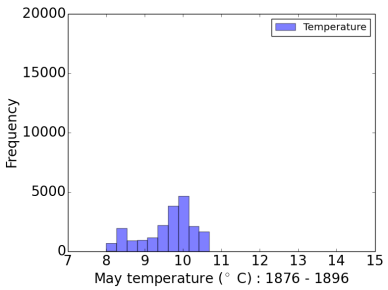
(d) May – TMCMC: 1816 - 1836



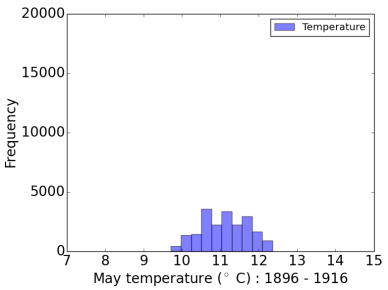
(e) May – TMCMC: 1836 - 1856



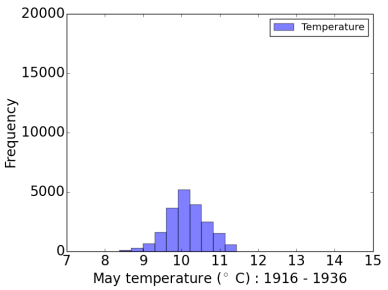
(f) May – TMCMC: 1856 - 1876



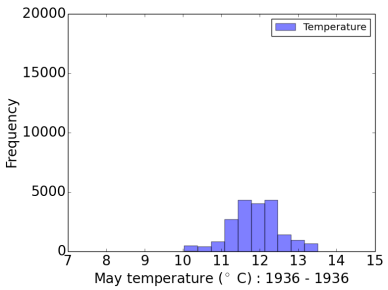
(g) May – TMCMC: 1876 - 1896



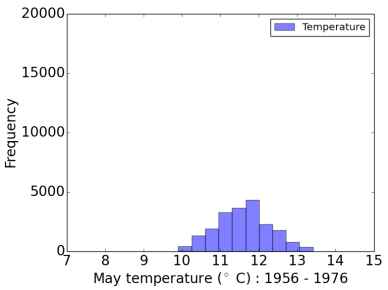
(h) May – TMCMC: 1896 - 1916



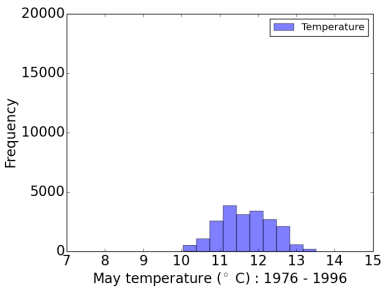
(i) May – TMCMC: 1916 - 1936



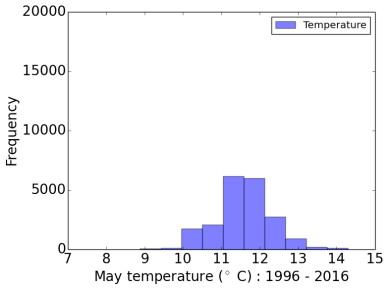
(j) May – TMCMC: 1936 - 1936



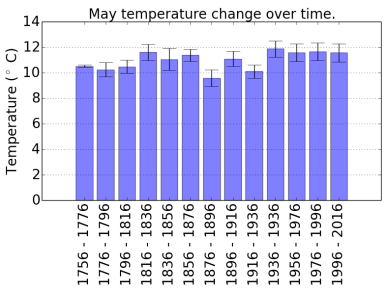
(k) May – TMCMC: 1956 - 1976



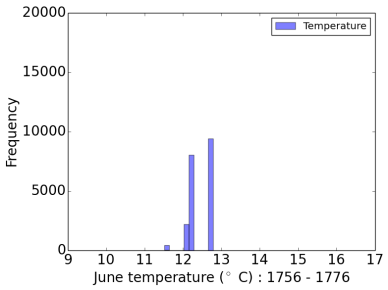
(l) May – TMCMC: 1976 - 1996



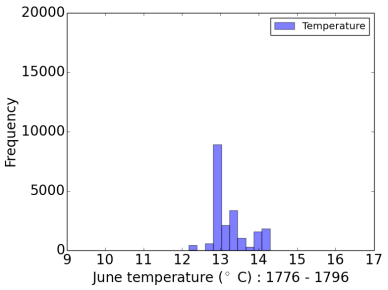
(m) May – TMCMC: 1996 - 2016



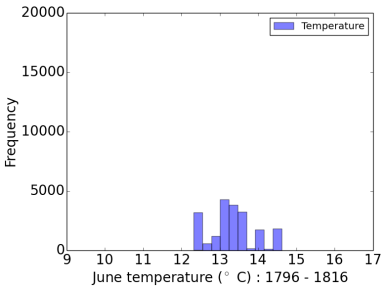
(n) May – TMCMC: Total



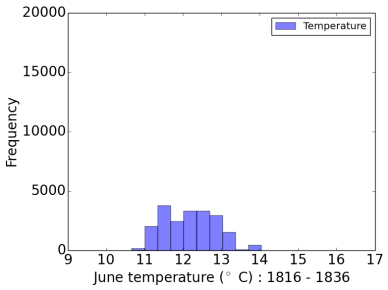
(a) June – TMCMC: 1756 - 1776



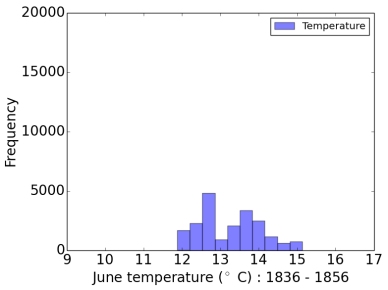
(b) June – TMCMC: 1776 - 1796



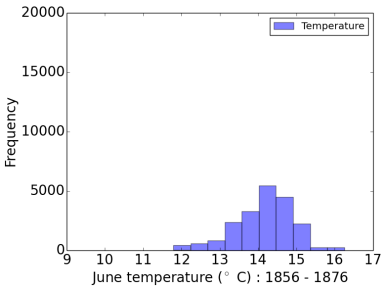
(c) June – TMCMC: 1796 - 1816



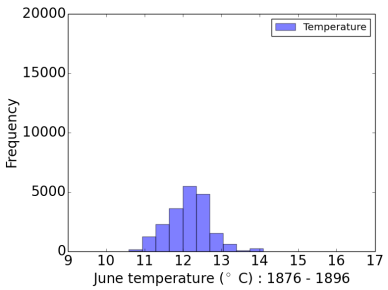
(d) June – TMCMC: 1816 - 1836



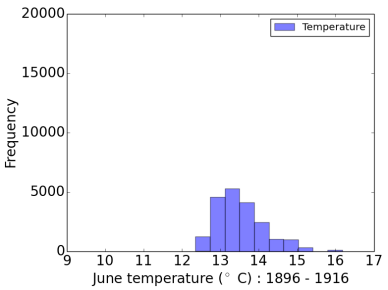
(e) June – TMCMC: 1836 - 1856



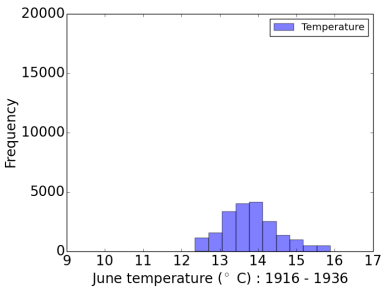
(f) June – TMCMC: 1856 - 1876



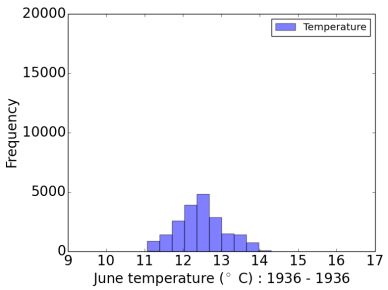
(g) June – TMCMC: 1876 - 1896



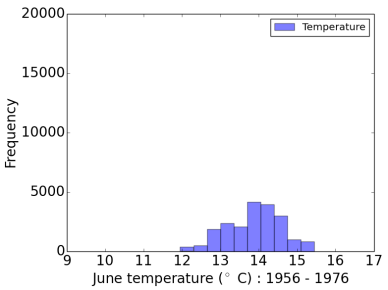
(h) June – TMCMC: 1896 - 1916



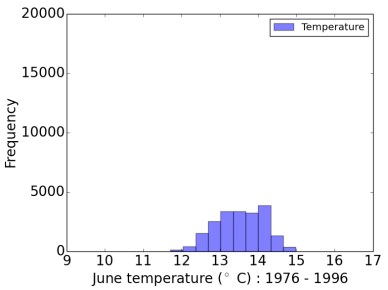
(i) June – TMCMC: 1916 - 1936



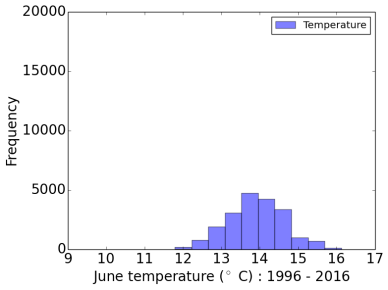
(j) June – TMCMC: 1936 - 1936



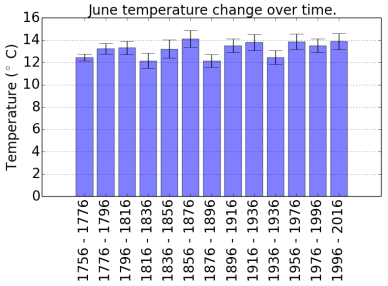
(k) June – TMCMC: 1956 - 1976



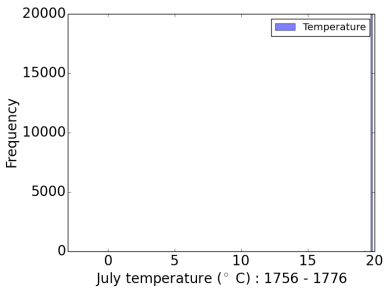
(l) June – TMCMC: 1976 - 1996



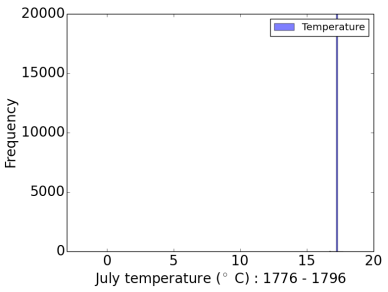
(m) June – TMCMC: 1996 - 2016



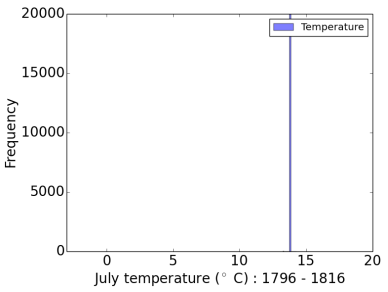
(n) June – TMCMC: Total



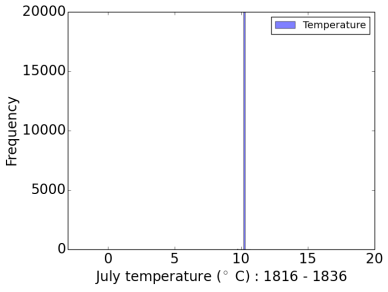
(a) July – TMCMC: 1756 - 1776



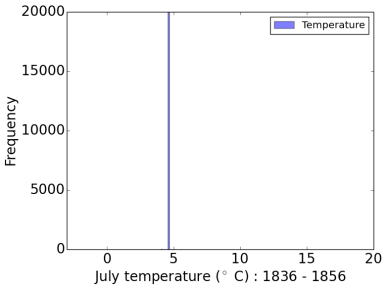
(b) July – TMCMC: 1776 - 1796



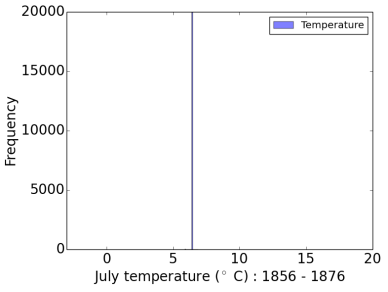
(c) July – TMCMC: 1796 - 1816



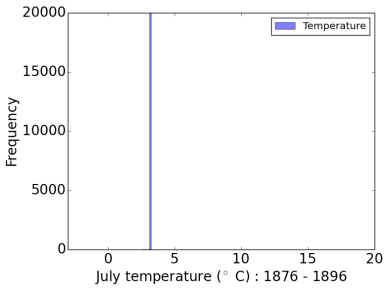
(d) July – TMCMC: 1816 - 1836



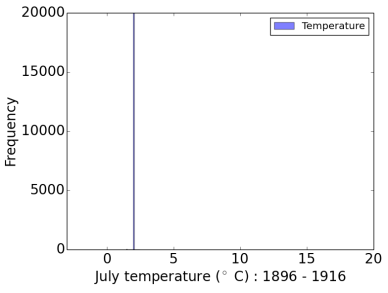
(e) July – TMCMC: 1836 - 1856



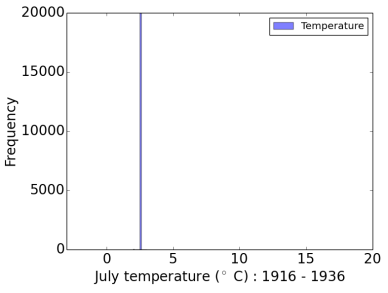
(f) July – TMCMC: 1856 - 1876



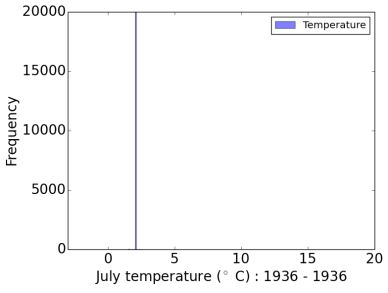
(g) July – TMCMC: 1876 - 1896



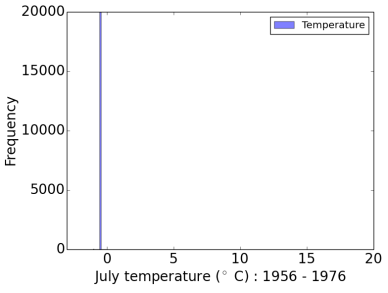
(h) July – TMCMC: 1896 - 1916



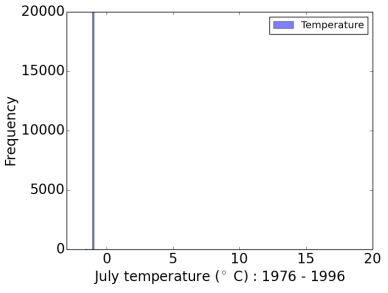
(i) July – TMCMC: 1916 - 1936



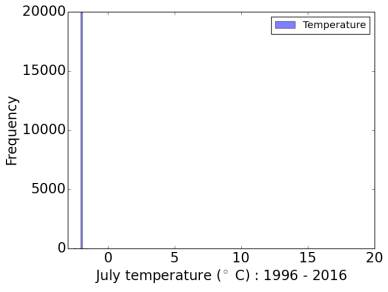
(j) July – TMCMC: 1936 - 1936



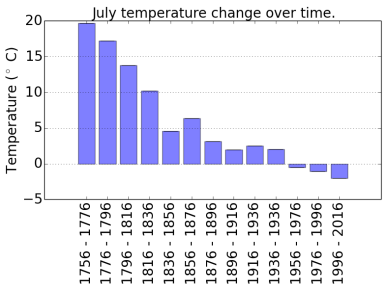
(k) July – TMCMC: 1956 - 1976



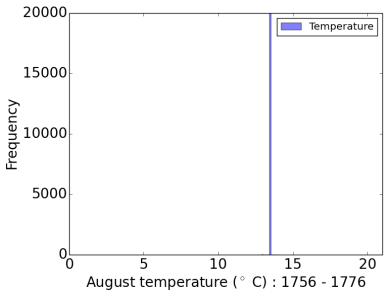
(l) July – TMCMC: 1976 - 1996



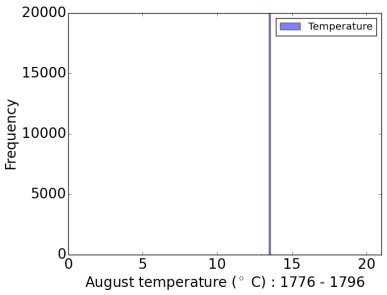
(m) July – TMCMC: 1996 - 2016



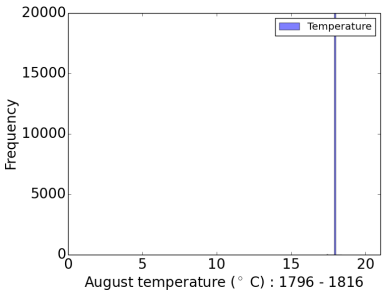
(n) July – TMCMC: Total



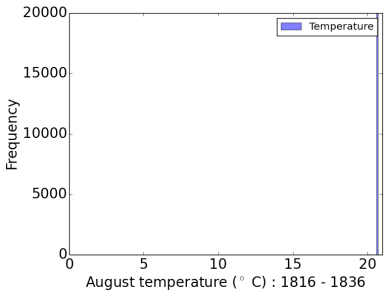
(a) August – TMCMC: 1756 - 1776



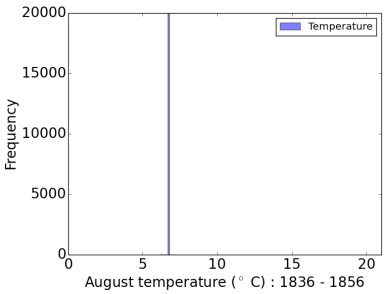
(b) August – TMCMC: 1776 - 1796



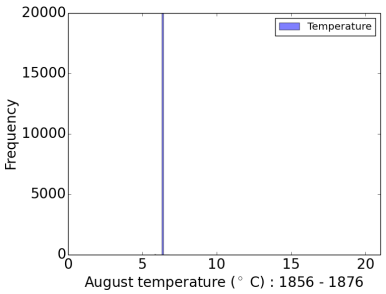
(c) August – TMCMC: 1796 - 1816



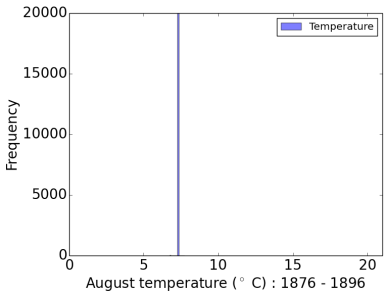
(d) August – TMCMC: 1816 - 1836



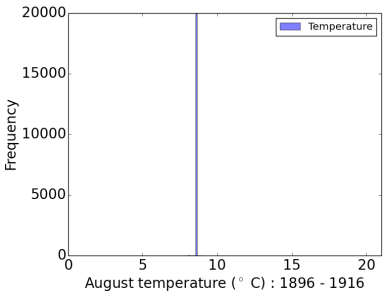
(e) August – TMCMC: 1836 - 1856



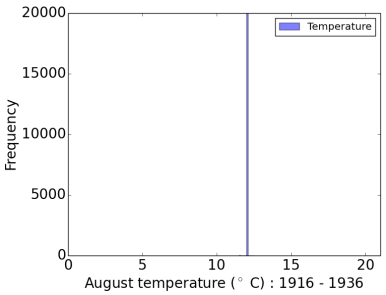
(f) August – TMCMC: 1856 - 1876



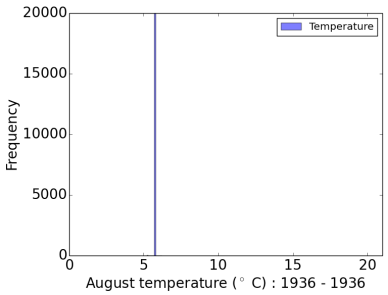
(g) August – TMCMC: 1876 - 1896



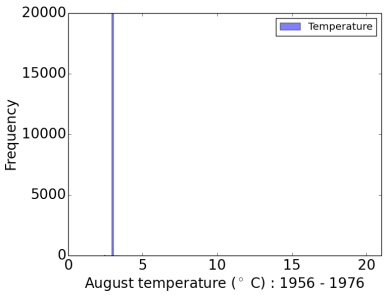
(h) August – TMCMC: 1896 - 1916



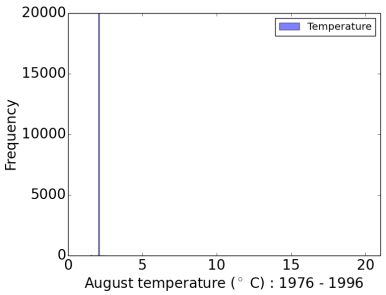
(i) August – TMCMC: 1916 - 1936



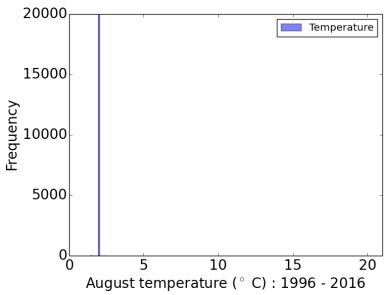
(j) August – TMCMC: 1936 - 1936



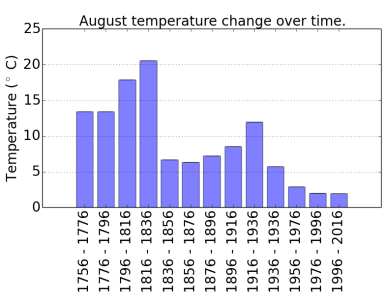
(k) August – TMCMC: 1956 - 1976



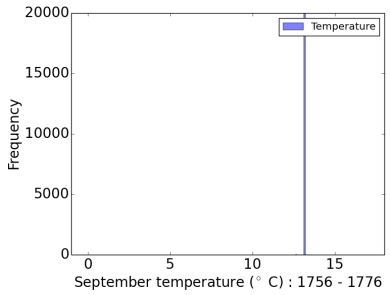
(l) August – TMCMC: 1976 - 1996



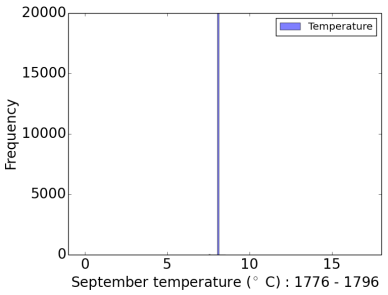
(m) August – TMCMC: 1996 - 2016



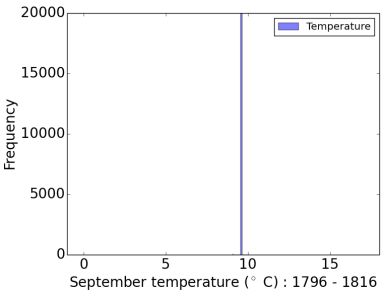
(n) August – TMCMC: Total



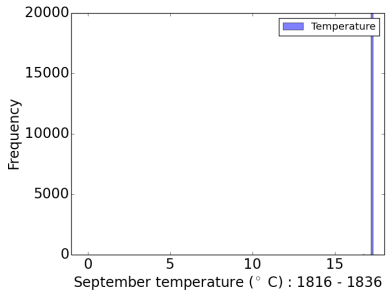
(a) September – TMCMC: 1756 - 1776



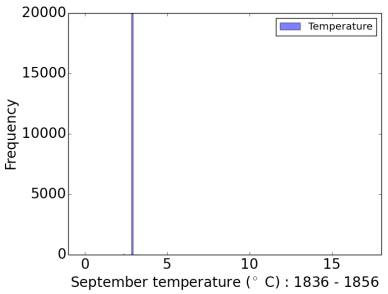
(b) September – TMCMC: 1776 - 1796



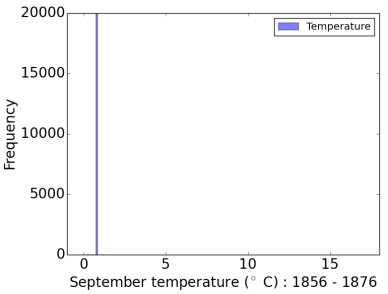
(c) September – TMCMC: 1796 - 1816



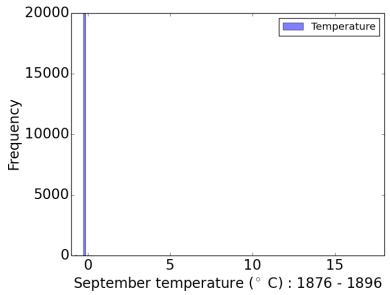
(d) September – TMCMC: 1816 - 1836



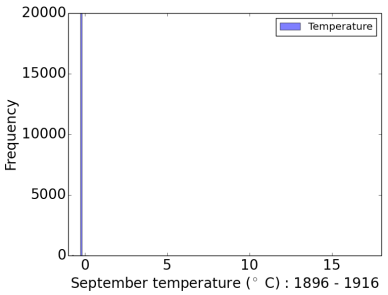
(e) September – TMCMC: 1836 - 1856



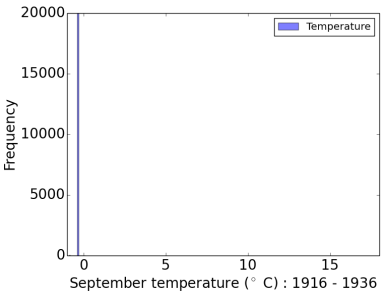
(f) September – TMCMC: 1856 - 1876



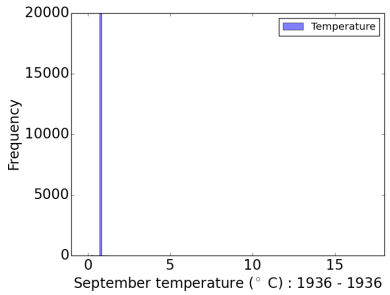
(g) September – TMCMC: 1876 - 1896



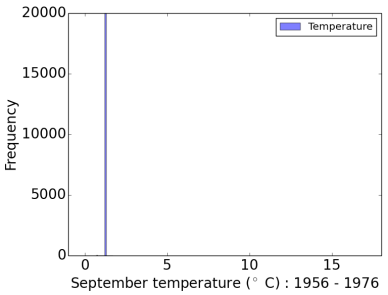
(h) September – TMCMC: 1896 - 1916



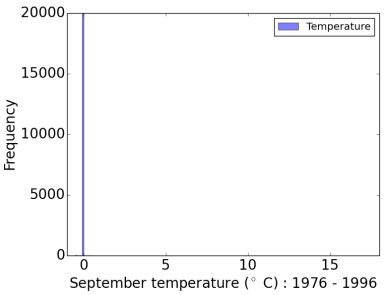
(i) September – TMCMC: 1916 - 1936



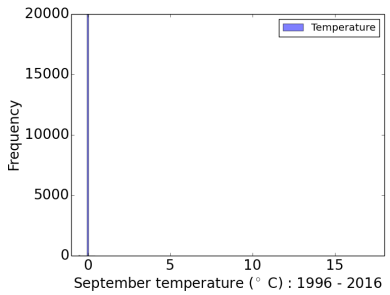
(j) September – TMCMC: 1936 - 1936



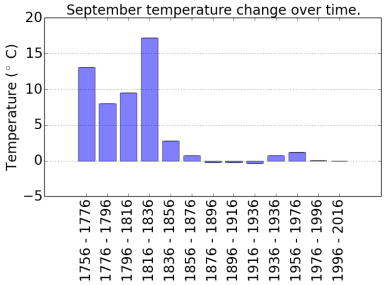
(k) September – TMCMC: 1956 - 1976



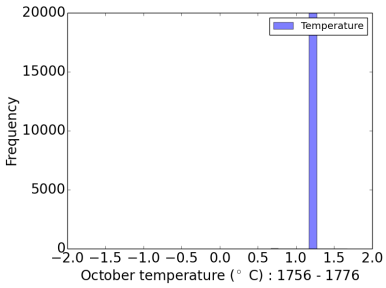
(l) September – TMCMC: 1976 - 1996



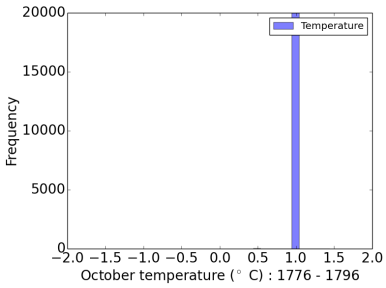
(m) September – TMCMC: 1996 - 2016



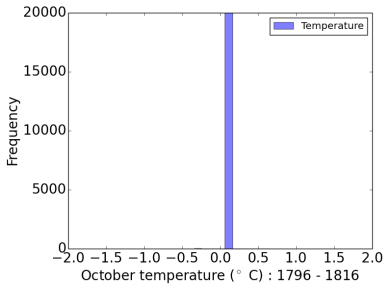
(n) September – TMCMC: Total



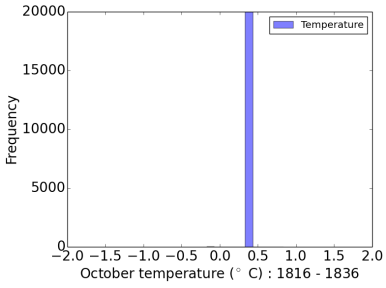
(a) October – TMCMC: 1756 - 1776



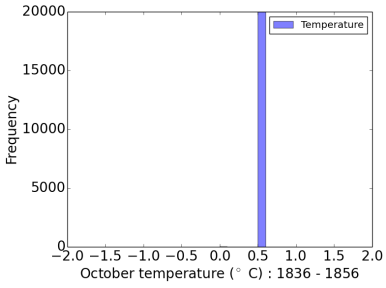
(b) October – TMCMC: 1776 - 1796



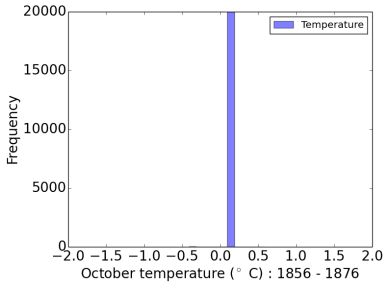
(c) October – TMCMC: 1796 - 1816



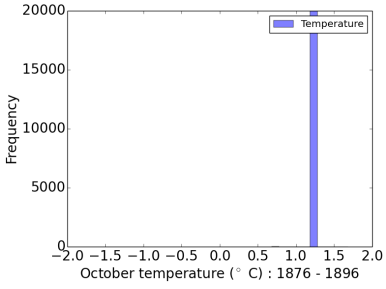
(d) October – TMCMC: 1816 - 1836



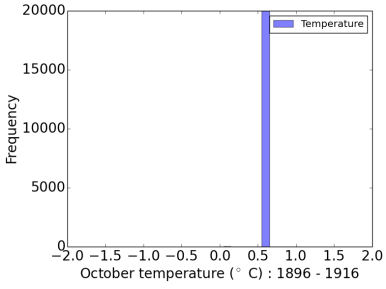
(e) October – TMCMC: 1836 - 1856



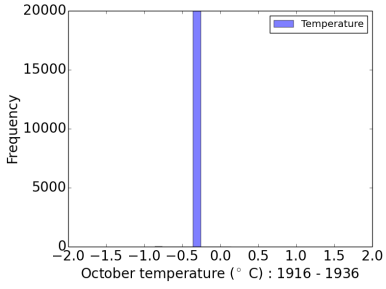
(f) October – TMCMC: 1856 - 1876



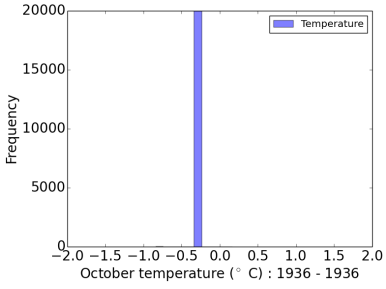
(g) October – TMCMC: 1876 - 1896



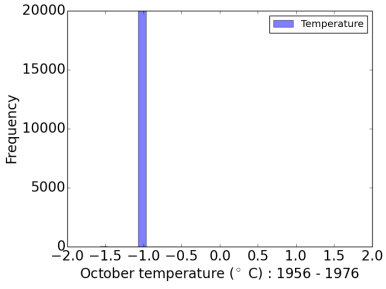
(h) October – TMCMC: 1896 - 1916



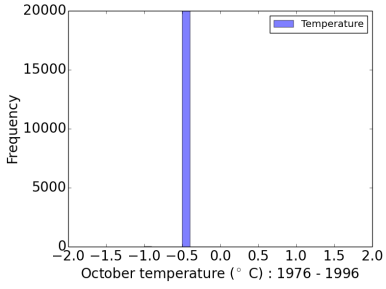
(i) October – TMCMC: 1916 - 1936



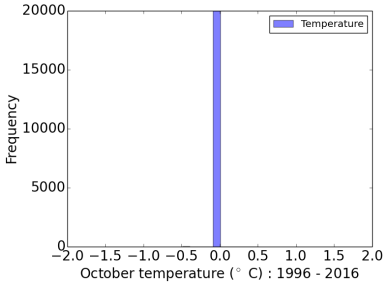
(j) October – TMCMC: 1936 - 1936



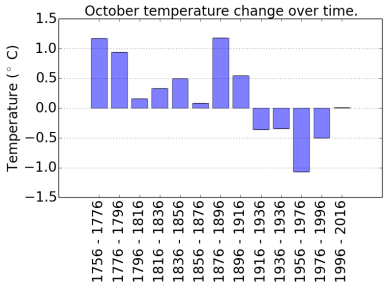
(k) October – TMCMC: 1956 - 1976



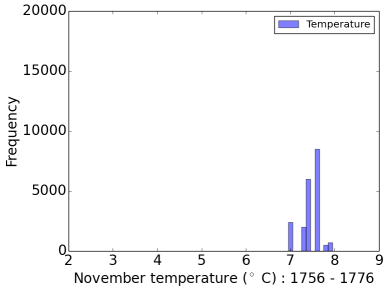
(l) October – TMCMC: 1976 - 1996



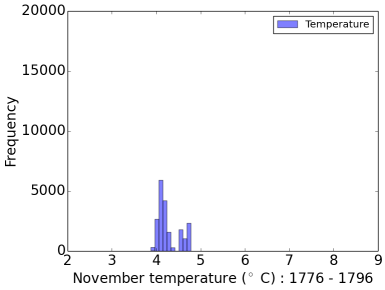
(m) October – TMCMC: 1996 - 2016



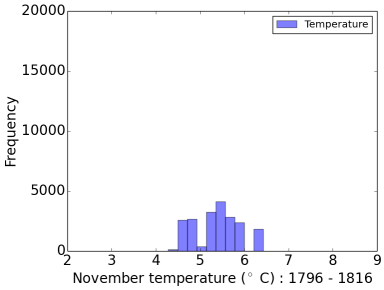
(n) October – TMCMC: Total



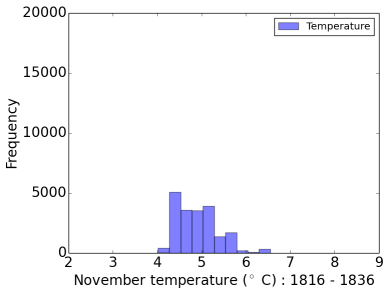
(a) November – TMCMC: 1756 - 1776



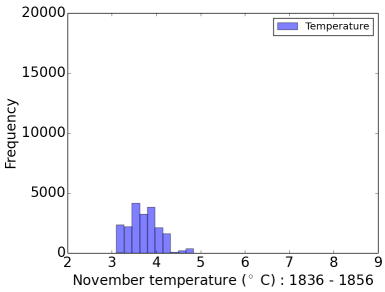
(b) November – TMCMC: 1776 - 1796



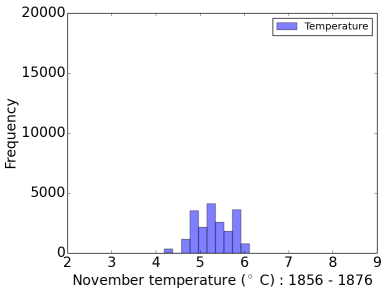
(c) November – TMCMC: 1796 - 1816



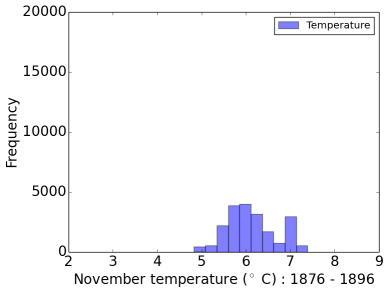
(d) November – TMCMC: 1816 - 1836



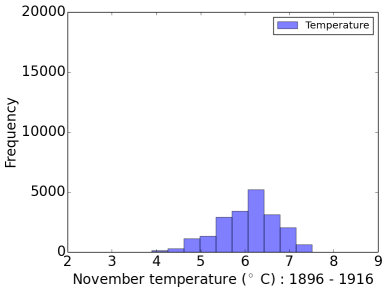
(e) November – TMCMC: 1836 - 1856



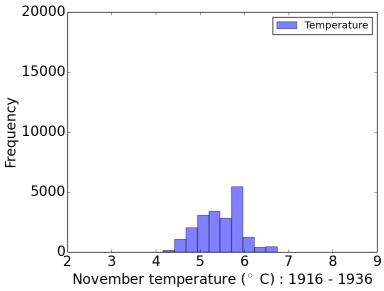
(f) November – TMCMC: 1856 - 1876



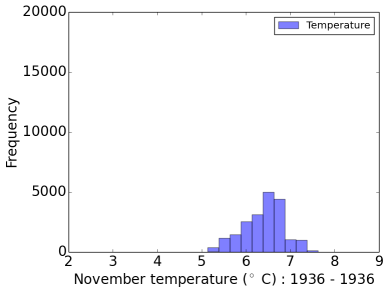
(g) November – TMCMC: 1876 - 1896



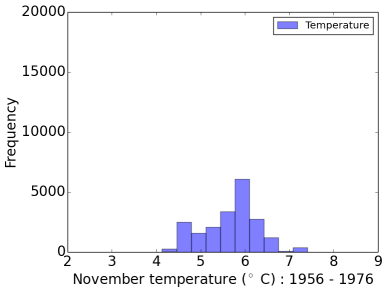
(h) November – TMCMC: 1896 - 1916



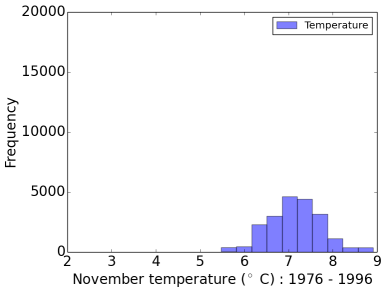
(i) November – TMCMC: 1916 - 1936



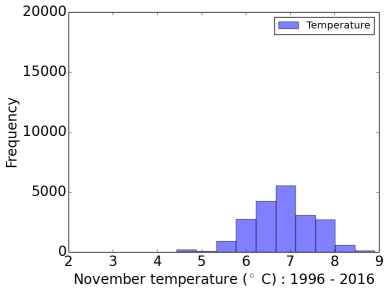
(j) November – TMCMC: 1936 - 1936



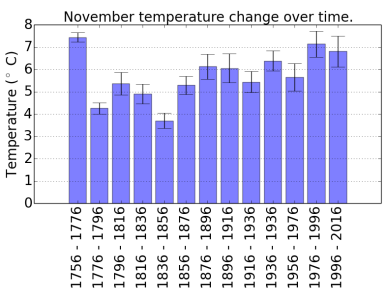
(k) November – TMCMC: 1956 - 1976



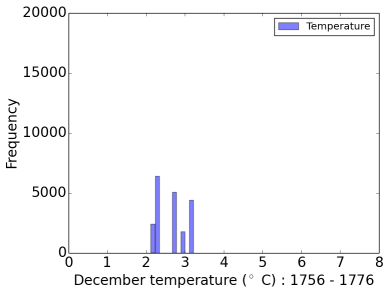
(l) November – TMCMC: 1976 - 1996



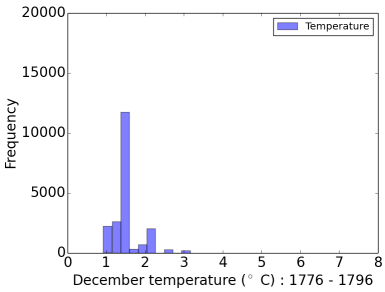
(m) November – TMCMC: 1996 - 2016



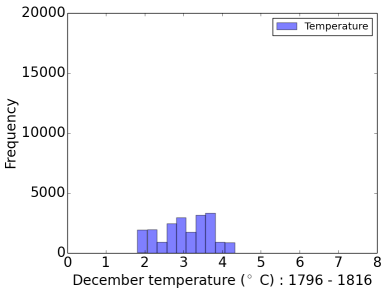
(n) November – TMCMC: Total



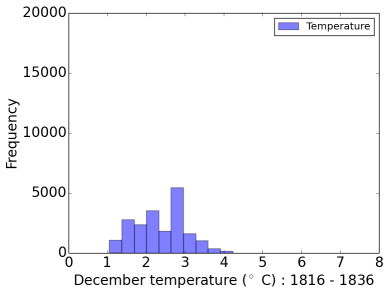
(a) December – TMCMC: 1756 - 1776



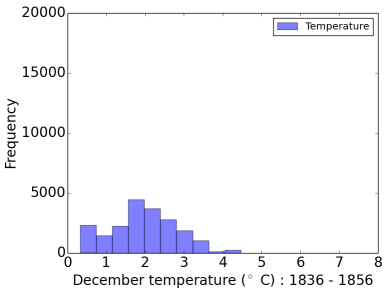
(b) December – TMCMC: 1776 - 1796



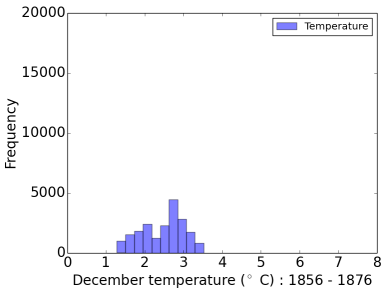
(c) December – TMCMC: 1796 - 1816



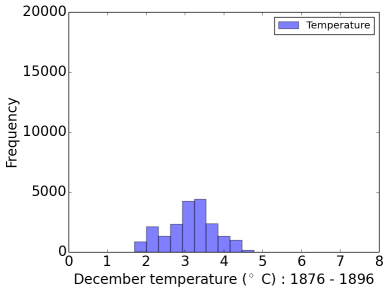
(d) December – TMCMC: 1816 - 1836



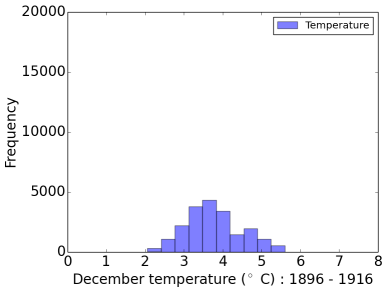
(e) December – TMCMC: 1836 - 1856



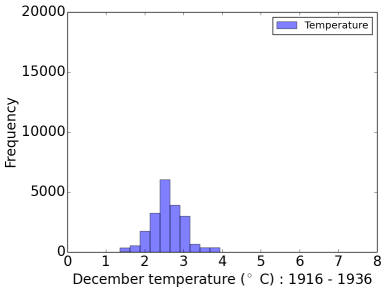
(f) December – TMCMC: 1856 - 1876



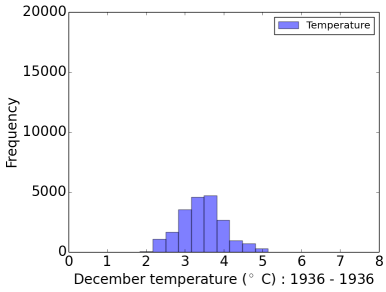
(g) December – TMCMC: 1876 - 1896



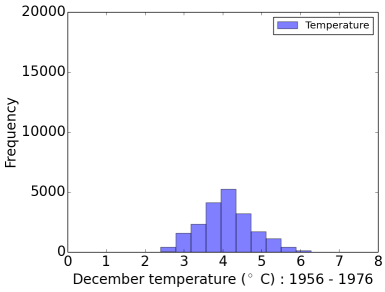
(h) December – TMCMC: 1896 - 1916



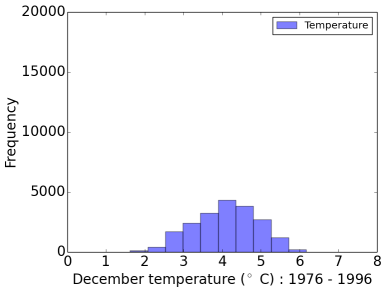
(i) December – TMCMC: 1916 - 1936



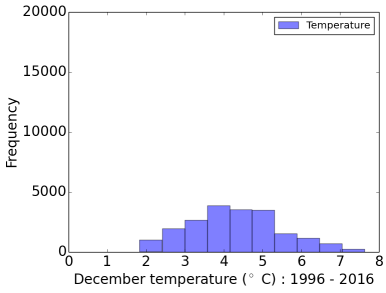
(j) December – TMCMC: 1936 - 1936



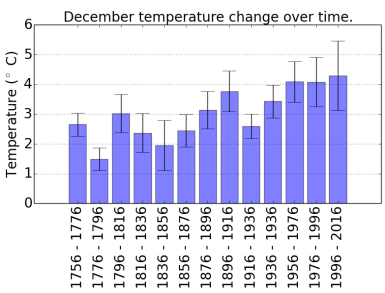
(k) December – TMCMC: 1956 - 1976



(l) December – TMCMC: 1976 - 1996



(m) December – TMCMC: 1996 - 2016



(n) December – TMCMC: Total