Dauphin: A Signal Processing Language
Statistical Signal Processing made easy

Ross Kyprianou
Defence Science and Technology Group (DSTG)
Email: ross.kyprianou@defence.gov.au

Peter Schachte
Department of Computer Science and Software Engineering, University of Melbourne
Email: schachte@unimelb.edu.au

Bill Moran
Department of Electrical and Computer Engineering, RMIT University
Email: bill.moran@rmit.edu.au

***Abstract*—Dauphin is a new statistical signal processing language designed for easier formulation of detection, classification and estimation algorithms. This paper demonstrates the ease of developing signal processing algorithms in Dauphin. We illustrate this by providing exemplar code for two classifiers: Bayesian and K-means, and for an estimator: the Kalman filter. In all cases, and especially the last named, the code provides a more conceptually defined approach to these problems than other languages such as Matlab. Some Dauphin features under development are also highlighted, for instance a infinite list construct called *streams*, which is designed to be used as a natural representation of random processes.**

I. Introduction

Statistical signal processing applies statistical principles to make rigorous inferences about the world from sensor measurements corrupted by noise and imperfect measurement processes. Inferences informally fall into two general categories:

Estimation is the process of inferring continuous values from measurements, such as the mean and covariance of a population from a finite number of samples (parameter estimation), or estimating and predicting the state of a system with a well defined if uncertain dynamical model from a finite number of observations (trackers and filters).

Hypothesis tests are used to infer which of a finite number of alternatives best corresponds to an observation. Binary hypothesis tests or *detectors* decide between a *null* or *alternative hypothesis*, for example the presence or absence or a target against a significant clutter background. Non-binary hypothesis tests or *classifiers* decide between two or more categories or classes. Classification problems include face recognition, ship classification from ISAR images and converting text from speech.

There are many mathematical tools available for developing statistical signal processing algorithms, for instance Matlab [1], Octave [2], Mathematica [3], Maple [4] and R [5], but none cater specifically for statistical signal processing problems and most importantly, none are designed to assist the signal processor with problem formulation. All require significant work before a signal processing problem is expressed in the language in question. Translation from the natural domain-specific structure of problem description to the computer formulation is often a time consuming, unnatural and error-prone exercise. The greater the difficulty of translating mathematical formulations to software, the greater the likelihood that errors are introduced. This also necessitates the researcher spending time on the translation process rather than focusing on the problem itself, impacting productivity as well as program correctness. For instance, without direct support for primitives such as random variables and random processes, the user is forced to code signal processing problems in an unnatural way. There is a need to codify the basic tools of signal processing, thus allowing the researcher to spend more time on the challenges specific to a particular application.

Dauphin is a domain-specific programming language ultimately aiming to extend the power of signal processing researchers by allowing them to focus on their research problems while simplifying the process of implementing their ideas. In Dauphin, the basic algorithms of signal processing become standard function calls and are expressed naturally in terms of predefined signal processing primitives such as probability distributions, random variables and random processes.

Dauphin is designed to be easy to learn, to read and to write in. Its syntax is deliberately similar to C and Matlab or Octave — languages in popular use in signal processing. Dauphin is also a functional and declarative language. Declarative programming languages are closer to mathematics than imperative languages such as C++, Fortran and Matlab, making them ideal for the signal processing domain. Also, declarative languages focus on what is to be done rather than how it should be done, and work at a higher level of abstraction, which support the researcher to focus on the problem at hand.

The paper comprises of two parts. In the first part, we discuss classification and describe two classifiers, Bayes and K-means, comparing their implementation in Dauphin and other languages. In the second part, we discuss Dauphin Streams, a feature being added to the next version of Dauphin which will allow estimators such as the Kalman filter [6] to be more easily written. For all algorithms featured, we discuss both their mathematical formulation and show that their translation to Dauphin code is natural, succinct and reflects the underlying mathematics.

II. Classifiers

The goal of classification is to infer which hypothesis or class an observation corresponds to from a number of competing alternatives.

Classifiers or hypothesis tests can be interpreted geometrically as algorithms that partition an observation space into multidimensional regions, each region corresponding to a hypothesis or class. It is important to note that different classifiers can result in a different partitioning of the observation space. In general, the regions are inferred during a *training stage* from a limited number of *training points* and after training, each new observation is mapped to a particular region which corresponds to the class it is said to belong to.

In this section, two classifiers — Bayes and K-means — are presented, both mathematically and as Dauphin programs with the aim of demonstrating that Dauphin programs are easy to read and write while reflecting the underlying mathematical structure of their algorithms.

A. *THE BAYES CLASSIFIER*

The Bayes Classifier assumes prior probabilities or *priors*, $P(H_j)$ for each hypothesis H_j and partitions the observation space so that the average probability of error is minimised. This is achieved by selecting the hypothesis H_i that maximises the posterior probability $p(H_j|\boldsymbol{x})$. Now, according to Bayes Rule, $p(H_j|\boldsymbol{x}) = p(\boldsymbol{x}|H_j)\ P(H_j)/p(\boldsymbol{x})$ and since $p(\boldsymbol{x})$ is independent of j, the Bayes Classifier reduces to

$$\overset{ArgMax}{j}\ P(H_j)\, p(\boldsymbol{x}|H_j) \ \text{ for } \ 1 \le j \le M \tag{1}$$

where M is the number of hypotheses.

There is a direct translation[1] of Equation 1 to Dauphin, shown in Listing 1.

Listing 1: The Bayes Classifier in Dauphin

```
      Bayes(X, Prior, H) = {
        ArgMax{j} Prior[j]*PDF(X, H[j]);
        }
```

Although the Bayes function body is merely a single line of code, it illustrates some important characteristics of Dauphin:

1) Preservation of mathematical structure: If the underlying mathematics of an algorithm can be translated directly and requires little effort to code and minimal lines to write then the likelihood of errors decreases. Comparing Equation 1 to Listing 1 we see the underlying mathematics corresponds closely to the resulting Dauphin code.

2) Dauphin is declarative: The declarative nature of Dauphin is highlighted in a function such as `ArgMax` which, although available in other languages, is not available in this form. `ArgMax` here is applied to a compound expression Prior[j]*PDF(X, H[j]) with an implied loop running over the index, j, which is listed inside the braces following `ArgMax`. This is an example of declarative languages letting the user express what needs to be done rather than how. In using this form of `ArgMax`, the user is only interested in the value of the index corresponding to the maximum value of the expression and is freed from many details in calculating this. The compiler can ascertain the lengths of the lists of priors Prior[j] and hypotheses H[j], calculate the value of the expression Prior[j]*PDF(X, H[j]) for each index j and return the value of the index corresponding to the maximum. There is no need for the user to provide lower and upper bounds of the loop (a common source of errors often called "off-by-one" errors) nor the boilerplate for the loop itself, making the code easier to write and clearer to read.

3) Type inferencing: Dauphin's type inferencing means the Bayes Classifier of Listing 1 works in both univariate and multivariate contexts. The observation point X, passed as an input argument to the Bayes function can be either a real number or a multidimensional vector. The code works without change for either case because the operations and functions are meaningful for both cases.

4) Dauphin is succint: The direct representation of common signal processing concepts and functions often allows the signal processsor to code one line of mathematics as one line of Dauphin code. This is also true of symbolic languages such as Mathematica [3] and Maple [4] but Dauphin's focus on signal processing makes it often more convenient to use in that domain. For example, Mathematica is a comprehensive mathematical language with many advantages over numerical languages (including Dauphin) such as the ability to multiply both numeric and symbolic matrices (Listing 2).

Listing 2: Mathematica session showing matrix multiplication

```
In[1]:= {{10, 0}, {0, 20}}.{{1, 2}, {3, 4}}
Out[1]= {{10, 20}, {60, 80}}

In[2]:= {a, b, c} . {x,y,z}
Out[2]= a x + b y + c z
```

Although Dauphin currently lacks symbolic capabilities, it uses a similar convenient matrix syntax as other numeric packages such as Matlab [1], Octave [2], Julia [7] and SciLab [8]. The numeric multiplication of Listing 2 is coded in Dauphin and these four languages succinctly as

```
[10 0; 0 20] * [1 2; 3 4].
```

Another simple example is matrix transposition where

```
Transpose[{{1, 2}, {3, 4}}]
```

in Mathematica corresponds to the simpler

```
[1 2; 3 4]'
```

in Dauphin (and Matlab, Octave, Julia and SciLab). It's important that languages used for signal processing make common signal processing operations easy to code.

A working example

To illustrate the Bayes classifier, we reproduce a multivariate Normal example from [9] using equal priors and the following mean and covariance values

$$\boldsymbol{\mu_1} = \begin{pmatrix} 3 \\ 6 \end{pmatrix} \qquad \boldsymbol{\Sigma_1} = \begin{pmatrix} 1/2 & 0 \\ 0 & 2 \end{pmatrix}$$

$$\boldsymbol{\mu_2} = \begin{pmatrix} 3 \\ -2 \end{pmatrix} \qquad \boldsymbol{\Sigma_2} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

[1] Traditionally mathematical formulas are terse, relying on conventions for their meaning whereas software programmers are encouraged to use meaningful names. Here, the j^{th} prior $P(H_j)$, corresponds to Prior[j]. Also, the probability density function or PDF of hypothesis j, evaluated at $\boldsymbol{x}$ *i.e.* $p(\boldsymbol{x}|H_j)$ corresponds to PDF(X, H[j]). The important point to note here is not that any names have changed but that the mathematical structure has been preserved.

This example illustrates how Dauphin can be interfaced with languages such as C and C++. Listing 3 contains the high-level, signal processing specific Dauphin code and Listing 4 shows C code that calls the Dauphin function `Classify`, which in turn calls the `Bayes` function with input X to classify X.

Listing 3: bayes.dau – the Bayes Classifier parameters

```
Bayes(X, Prior, H) = {
  ArgMax{j} Prior[j]*PDF(X, H[j]);
}

Classify(X) = {
 let Mu1    = [3; 6];
     Sigma1= [1/2 0; 0 2];
     Mu2    = [3; -2];
     Sigma2= [2 0; 0 2];
     Prior  = [1/2 1/2];
     H      = [ NormalRV(Mu1,Sigma1)
                NormalRV(Mu2,Sigma2) ]
 in Bayes(X, Prior, H)
}
```

Listing 4: main.c – Classifying a data point from C

```
#include "bayes.h"

void main() {
 //this defines X=[3,1.825] as 2x1 matrix
 double X[] = {2,1,  3,1.825};

 printf("X IS CLASSIFIED AS CLASS NO: %d\n",
        Classify( Dau_Matrix(X)) );
}
```

After compiling and linking main.c (Listing 4) with bayes.dau (Listing 3), the resultant binary displays

```
X IS CLASSIFIED AS CLASS NO: 1
```

The Bayes classifier of Listing 1 demonstrates that Dauphin succeeds in a number of design goals. First, Dauphin supports signal processing primitives such as random variables (represented by the array H[j]). Subsequently, this results in the code having the same structure as the mathematical formulation. Therefore, there is little effort in translating the mathematics into code which means the code is easy to write and read, is succinct and consequently is less likely to have errors.

B. *THE K-MEANS CLASSIFIER*

The Bayes classifier is a supervised classifier, in that it infers class regions during training from class-labelled observations. In contrast, K-means is an unsupervised classifier, grouping unlabelled observations into regions or clusters.

The goal of K-means is to group a set of N observations $\boldsymbol{X} = \{\boldsymbol{x}_n : n = 1, \dots N\}$ from an observation space S into K clusters, S_k, $k = 1, \dots K$ such that $X = \bigcup_{k=1}^{K} S_k$ and each $\boldsymbol{x}_n$ is a member of exactly one cluster S_k. Following the discussion in [10], membership can be represented with the indicator function r_{nk}, defined as

$$r_{nk} = \begin{cases} 1 & \text{if } \boldsymbol{x}_n \in S_k \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

One way a cluster S_k can be considered to be a set of points "close to each other" is to define a representative point $\boldsymbol{\mu}_k$ for each S_k and then define S_k as the subset of $\boldsymbol{x}_n$ that are closer to $\boldsymbol{\mu}_k$ than any other representative point. The points $\boldsymbol{x}_n$ are partitioned into such clusters optimally by finding J such that

$$J = \underset{\boldsymbol{\mu}_k; r_{nk}}{\text{ArgMin}} \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2 \tag{3}$$

where for each n, $r_{nk} = 1$ for exactly one k and $r_{nk} = 0$ otherwise.

The $\boldsymbol{\mu}_k, r_{nk}$ of Equation 3 can be found by iteratively applying two steps: (a) minimising with respect to r_{nk} while keeping the $\boldsymbol{x}_n$ fixed then (b) minimising with respect to $\boldsymbol{x}_n$ while keeping r_{nk} fixed.

For the first step, we note J is linear in r_{nk} and the terms involving n are independent so for a fixed set of $\boldsymbol{\mu}_k$, J is minimal when $r_{nk}\|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|$ is minimal for each n. This is achieved if we set $r_{nk} = 1$ for the k that minimises $\|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|$ and set it to 0 for all other values of k. Formally,

$$r_{nk} = \begin{cases} 1 & \text{if } k = \underset{j}{\text{ArgMin}} \|\boldsymbol{x} - \boldsymbol{\mu}_j\|^2 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The second step minimises J over $\boldsymbol{\mu}_k$ with fixed $\boldsymbol{x}_n$. Since J is a quadratic function of $\boldsymbol{\mu}_k$, J has a minimum where its derivative with respect to $\boldsymbol{\mu}_k$ is zero *i.e.*

$$\sum_{n=1}^{N} r_{nk}(\boldsymbol{x}_n - \boldsymbol{\mu}_k) = 0 \tag{5}$$

and solving for $\boldsymbol{\mu}_k$ gives

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} r_{nk}\boldsymbol{x}_n}{\sum_{n=1}^{N} r_{nk}} \tag{6}$$

Since the denominator is the number of points in cluster k and the numerator is the vector sum of the points in cluster k then $\boldsymbol{\mu}_k$ is the mean of the points of cluster k.

Once the optimal values of $\boldsymbol{\mu}_k$ and r_{nk} are established, the K-means classifier is said to be "trained" (for the given set of training points) and then K-means classifies a new point $\boldsymbol{x}$ as belonging to cluster m if

$$m = \underset{k}{\text{ArgMin}} \|\boldsymbol{x} - \boldsymbol{\mu}_k\|^2 \tag{7}$$

Next, a simple one dimension example will show the K-means algorithm at work. We will see the algorithm alternating between two steps: (a) finding clusters and (b) updating centers. A cluster is defined to be the set of points X_j closest to a particular center. A center C_i of a cluster is the mean of all the points of the cluster.

The one dimension example in its initial configuration is shown in Figure 1. There are 5 points $\{X_1, X_2, X_3, X_4, X_5\}$ at fixed positions. We will choose to find 2 clusters for this example, therefore we place 2 centers C_1 and C_2 at random starting positions as shown in Figure 1.

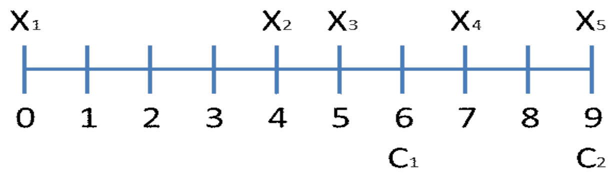

Fig. 1: 1D K-means example (Initial configuration)

For the first step of Iteration 1 — "find clusters" — we find the points that are nearest to each center. The first cluster, containing the points that are closer to C_1 than C_2, are seen to be the points $\{X_1, X_2, X_3, X_4\}$, colored green below. The only point that is closer to C_2 than C_1 is X_5 (colored red) which defines the second cluster.

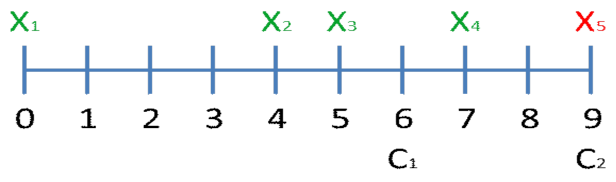

Fig. 2: Iteration 1: (a) Find clusters

The second step of Iteration 1 — "update centers" — updates the centers' positions to the mean of the points for each cluster. The mean of the positions of the first cluster $\{X_1, X_2, X_3, X_4\}$ is $(0+4+5+7)/4 = 4$ and the mean of the second cluster (with only one point) is $9/1 = 9$ so the centers for Iteration 1 are updated to positions: 4 for C_1 and 9 for C_2 as shown in Figure 3.

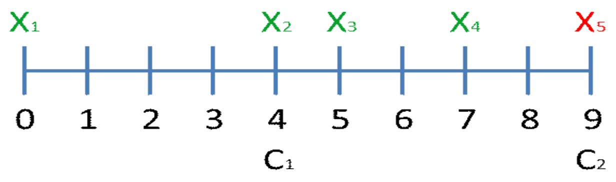

Fig. 3: Iteration 1: (b) Update centers

We note that the centers' positions were updated from their initial positions in Iteration 1 so we repeat the two steps. We continue until the centers do not change. This is known as a *fixed point solution*. For the next iteration, we once again first find the clusters for the updated centers (Figure 4) and then we update the center positions (Figure 5). This completes Iteration 2.

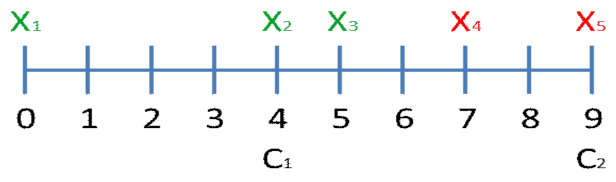

Fig. 4: Iteration 2: (a) Find clusters

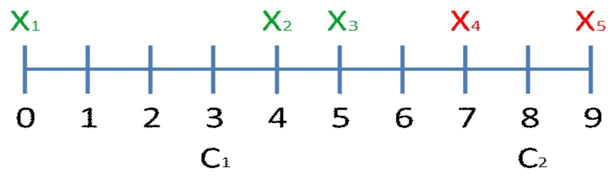

Fig. 5: Iteration 2: (b) Update centers

The next iteration (and any subsequent iteration) results in no change to the clusters and consequently there will be no change to the position of the centers. Having arrived at a fixed point solution, we stop.

We next introduce some Dauphin functions that are used to produce the above results. Following this, a complete Dauphin implementation of the K-means algorithm using these functions will be shown, together with implementations in other languages for comparison.

Let X and C be row matrices[2] denoting the positions of the points X_j for $j = 1, \ldots, 5$ and C_i for $i = 1, 2$. Initially, $X = [0\ 4\ 5\ 7\ 9]$ and $C = [6\ 9]$ (Figure 1).

Let Dist denote the Euclidean distance function. For example, $\text{Dist}(C_1, X_1) = |6 - 0| = 6$, $\text{Dist}(C_1, X_2) = |6 - 4| = 2$ and so on. To calculate the centre that each point is closest to, we first form a matrix D with entries $d_{ij} = \text{Dist}(C_i, X_j)$. This results in the matrix

$$D = \begin{array}{c|ccccc|} & X_1 & X_2 & X_3 & X_4 & X_5 \\ \hline C_1 & 6 & 2 & 1 & 1 & 3 \\ C_2 & 9 & 5 & 4 & 2 & 0 \\ \hline \end{array} \qquad (8)$$

where we have labelled the rows and columns with their corresponding centers C_i and points X_j.

The Dauphin function, `MapCol` can directly produce the matrix D. `MapCol` is one of Dauphin's *higher order functions* — a function that takes a function as an argument. It produces matrix D with the call

```
MapCol(Dist,C,X)
```

by applying `Dist` pairwise on each column C_i of C and X_j of X to form D with entries $d_{ij} = \text{Dist}(C_i, X_j)$.

[2]If the problem were M-dimensional rather than 1-dimensional, the point X_j would be the j^{th} column of X with length M. Nevertheless, the discussion will remain unchanged since the functions being discussed work in the same way for the M-dimensional case as for the 1D case.

In general, MapCol(f,A,B) forms the matrix F with entries $f_{ij} = f(A_i, B_j)$ where: A_i is the i^{th} column of A; B_j is the j^{th} column of B; and f_{ij} is a either a scalar or matrix. If f_{ij} is a matrix then F is defined by the concatenation of the submatrices, f_{ij}. Dauphin also has `MapRow`, which applies a function to corresponding rows of A and B. Similarly, `Map` applies a function to corresponding elements of two matrices.

Returning to our example, the j^{th} column of D stores the distances of X_j to each C_i. Applying the Dauphin function `ArgMin` to D returns the row index of the minimum value of each column *i.e.* the index of the closest center for each X_j. This can be achieved in a single step:

ArgMin(MapCol(Dist,C,X)).

The matrices produced by `MapCol` (of the distances from each C_i to each X_j) are shown below for the Initial configuration (see Figure 1) and for Iterations 1 and 2 (Figure 3 and Figure 5). Below each of these matrices is the row matrix produced after applying `ArgMin` to it. A value of 1 (or 2) in column j of the row matrix means X_j is closer to C_1 (or C_2). For example, the `ArgMin` row matrix of the "Initial configuration" below has the first 4 entries equal to 1 therefore X_1, X_2, X_3 and X_4 are closest to C_1 and belong to one of the clusters and the remaining point X_5 is the only member of the other cluster. Note that the unchanged `ArgMin` row matrix from Iteration 1 to Iteration 2 satisfies the stopping condition.

Initial configuration

$$\begin{array}{r|ccccc|} \text{MapCol} & X_1 & X_2 & X_3 & X_4 & X_5 \\ \hline C_1 & 6 & 2 & 1 & 1 & 3 \\ C_2 & 9 & 5 & 4 & 2 & 0 \\ \hline \end{array}$$

$$\begin{array}{r|ccccc|} \hline \text{ArgMin} & 1 & 1 & 1 & 1 & 2 \\ \hline \end{array}$$

Iteration 1

$$\begin{array}{r|ccccc|} \text{MapCol} & X_1 & X_2 & X_3 & X_4 & X_5 \\ \hline C_1 & 4 & 0 & 1 & 3 & 5 \\ C_2 & 9 & 5 & 4 & 2 & 0 \\ \hline \end{array}$$

$$\begin{array}{r|ccccc|} \hline \text{ArgMin} & 1 & 1 & 1 & 2 & 2 \\ \hline \end{array}$$

Iteration 2

$$\begin{array}{r|ccccc|} \text{MapCol} & X_1 & X_2 & X_3 & X_4 & X_5 \\ \hline C_1 & 3 & 1 & 2 & 4 & 6 \\ C_2 & 8 & 4 & 3 & 1 & 1 \\ \hline \end{array}$$

$$\begin{array}{r|ccccc|} \hline \text{ArgMin} & 1 & 1 & 1 & 2 & 2 \\ \hline \end{array}$$

A complete Dauphin implementation of K-means is in Listing 5.

We see `MapCol` and `ArgMin` on Line 1 of Listing 5 being used to find clusters as discussed in detail above. The centers are calculated on Lines 3–4 by calculating the mean (along rows) of the X_j corresponding to each cluster. Each cluster's X_js are selected using the indices stored in the row matrix produced by `ArgMin`. The `KMeans` function (Lines 6–11) repeats these two steps until the fixed point solution is found.

Listing 5: K-means in Dauphin

```
1  ClustersOf(X,C) = { ArgMin(MapCol(Dist,C,X)) }

3  CentersOf(X,C) =
4   {[for i=1:cols(C), MeanRows(X[:,find(C==i)])]}

6  KMeans(X,C, interations) = {
7    newC = CentersOf(ClustersOf(X,C))
8     if iterations > 0 && newC != C
9       then KMeans(X, newC, iterations -1)
10      else newC
11 }
```

Implementations of K-means in various languages can be found at http://ilnumerics.net/blog/fast-faster-performance-comparison-c-ilnumerics-fortran-matlab-and-numpy-part-ii. The Matlab/Octave and Python versions are reproduced in Listings 6–7 for comparison with the Dauphin implementation.

Listing 6: K-means in Matlab/Octave

```
function classes=kmeans(X, centers, iterations)
  n = size(X,2);
  k = size(centers,2);
  oldCenters = centers;
  while (iterations > 0)
    iterations = iterations - 1;
    for i = 1:k
      dist(i,:)=
        sum(bsxfun(@minus,X, centers(:,i)).^2);
    end
    [~, classes] = min(dist);
    for i = 1:k
      class_i = X(:,classes == i);
      centers(:,i) = mean(class_i,2);
    end
    if (all(all(oldCenters == centers))),
      break;
    end
    oldCenters = centers;
  end
```

Listing 7: K-means in Python

```
from numpy import *
def kmeans(X,k):
 n = size(X,1)
 maxit = 20
 centers = X[:,0:k].copy()
 classes = zeros((1.,n))
 oldCenters = centers.copy()
 for it in range(maxit):
  for i in range(n):
   dist=sum(abs(centers-X[:,i,newaxis]),axis=0)
   classes[0,i] = dist.argmin()

   for i in range(k):
    inClass = X[:,nonzero(classes == i)[1]]
    if inClass.size == 0:
     centers[:,i] = np.nan
    else:
     centers[:,i] = inClass.mean(axis=1)

    if all(oldCenters == centers):
     break
    else:
     oldCenters = centers.copy()
```

III. Future work: Dauphin Streams and the Kalman Filter

The version of Dauphin currently under development is introducing a *Streams* feature. One driver for this is to express more simply, algorithms such as the Kalman filter [6] that are naturally framed in terms of random processes.

The Kalman filter is an estimator of the states of a Linear Dynamical System where a Linear Dynamical System is defined as a sequence of (hidden) states $\boldsymbol{x}_k$, a sequence of noisy measurements of each state $\boldsymbol{z}_k$, a linear state transition model

$$\boldsymbol{x}_k = \boldsymbol{F}_k \boldsymbol{x}_{k-1} + \boldsymbol{w}_k \tag{9}$$

and a linear observation model relating the hidden state $\boldsymbol{x}_k$ to the measurement $\boldsymbol{z}_k$ according to

$$\boldsymbol{z}_k = \boldsymbol{H}_k \boldsymbol{x}_k + \boldsymbol{v}_k \tag{10}$$

where $\boldsymbol{x}_k, \boldsymbol{z}_k, \boldsymbol{w}_k$ and $\boldsymbol{v}_k$ are Normally distributed random variables with densities

$$\begin{aligned} p(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k) &= \mathcal{N}(\boldsymbol{F}_k\boldsymbol{x}_k, \boldsymbol{Q}_k) \\ p(\boldsymbol{z}_k|\boldsymbol{x}_k) &= \mathcal{N}(\boldsymbol{H}_k\boldsymbol{x}_k, \boldsymbol{R}_k) \\ p(\boldsymbol{w}_k) &= \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_k) \\ p(\boldsymbol{v}_k) &= \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_k). \end{aligned}$$

The $\boldsymbol{F}_k, \boldsymbol{H}_k, \boldsymbol{Q}_k$ and $\boldsymbol{R}_k$ are known matrices and $\boldsymbol{w}_k$ and $\boldsymbol{v}_k$ are independent.

We wish to estimate the state $\boldsymbol{x}_k$ at time k based on the measurements $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k\}$.

In adaptive filtering, the information about the current state of the system under consideration is contained in the "belief state". This is the probabililty density that describes the state and its uncertainty. This density at time k is the posterior density based on all previous measurements.

The particular case of the Kalman filter is a straightforward consequence of the Gaussian and linearity constraints on the one hand, and the Chapman-Kolmogorov equation

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1}) = \int_{\boldsymbol{X}} p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}) p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1})\, d\boldsymbol{x}_k \tag{11}$$

and Bayes rule

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) = \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})}{\int_{\boldsymbol{X}} p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})\, d\boldsymbol{x}_k} \tag{12}$$

on the other. As in the general situation, the Chapman-Kolmogorov equation is used to update the posterior to a prior for the application of Bayes rule involving the next measurement. What makes the Kalman Filter particularly easy to compute is that the assumptions and the use of these equations show that all of the densities under consideration are Gaussian, so that the information about the prior and posterior densities is entirely encoded in their means and covariance matrices. The Chapman-Kolmogorov equation and Bayes rule then provide a collection of equations:

Noise

$$\boldsymbol{w}_k \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_k) \tag{13}$$

$$\boldsymbol{v}_k \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_k) \tag{14}$$

Predict

$$\hat{\boldsymbol{x}}_{k|k-1} = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1|k-1} \tag{15}$$

$$\hat{\boldsymbol{\Sigma}}_{k|k-1} = \boldsymbol{F}_k \hat{\boldsymbol{\Sigma}}_{k-1|k-1} \boldsymbol{F}_k^{\mathrm{T}} + \boldsymbol{Q}_k \tag{16}$$

$$\hat{\boldsymbol{z}}_k = \boldsymbol{H}_k \hat{\boldsymbol{x}}_{k|k-1} \tag{17}$$

$$\hat{\boldsymbol{S}}_k = \boldsymbol{H}_k \hat{\boldsymbol{\Sigma}}_{k|k-1} \boldsymbol{H}_k^{\mathrm{T}} + \boldsymbol{R}_k \tag{18}$$

Update

$$\boldsymbol{K}_k = \hat{\boldsymbol{\Sigma}}_{k|k-1} \boldsymbol{H}_k^{\mathrm{T}} \hat{\boldsymbol{S}}_k^{-1} \tag{19}$$

$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k) \tag{20}$$

$$\hat{\boldsymbol{\Sigma}}_{k|k} = (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k) \hat{\boldsymbol{\Sigma}}_{k|k-1} \tag{21}$$

for the recursive update of these parameters. The apparent complexity of these equations hides the elegant simplicity of the Kalman Filter, but often in engineering treatments of the topic these equations are emphasized rather than the simple conceptual framework inherent in adaptive filtering.

The formulation in Dauphin, on the contrary emphasizes the concepts rather than the computational complexity of these equations. Our algorithm has two components, a prediction step that is just the application of the Chapman-Kolmogorov equation and an update step that is an application of Bayes rule.

Dauphin has been designed to represent random variables and processes directly as primitives and to apply a number of well known statistical rules automatically. For instance, by defining the two random variables $\boldsymbol{x}$ and $\boldsymbol{z}|\boldsymbol{x}$ as in Equations 22–23, the compiler automatically applies Bayes rule to the expression $\boldsymbol{x}|\boldsymbol{z}$ (Equation 24) to calculate the mean and covariance according to Equations 25–26.

$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{x}},\, \boldsymbol{\Sigma}_{\boldsymbol{x}}) \tag{22}$$

$$\text{and} \qquad \boldsymbol{z}|\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}, \boldsymbol{\Sigma}_{\boldsymbol{z}}) \tag{23}$$

$$\text{then} \qquad \boldsymbol{x}|\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}_{x|z}, \boldsymbol{\Sigma}_{x|z}) \tag{24}$$

$$\text{where} \qquad \boldsymbol{\Sigma}_{x|z} = \boldsymbol{\Sigma}_x^{-1} + \boldsymbol{A}^T \boldsymbol{\Sigma}_z^{-1} \boldsymbol{A} \tag{25}$$

$$\boldsymbol{\mu}_{x|z} = \boldsymbol{\Sigma}_{x|z}[\boldsymbol{A}^T \boldsymbol{\Sigma}_z^{-1}(\boldsymbol{z} - \boldsymbol{b}) + \boldsymbol{\Sigma}_x^{-1} \boldsymbol{\mu}_x] \tag{26}$$

Dauphin is designed to also automatically apply well-known facts such as: if $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}_x, \boldsymbol{\Sigma}_x)$ and $\boldsymbol{Y} \sim \mathcal{N}(\boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y)$ are two independent Normal random variables then $\boldsymbol{X} + \boldsymbol{Y} \sim \mathcal{N}(\boldsymbol{\mu}_x + \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_x + \boldsymbol{\Sigma}_y)$ and the affine transformation $\boldsymbol{W} = \boldsymbol{A}\boldsymbol{X} + \boldsymbol{b}$ where matrix $\boldsymbol{A}$ is the size of $\boldsymbol{\Sigma}_x$ and vector $\boldsymbol{b}$ is size of $\boldsymbol{\mu}_x$, results in another Normal random variable $\boldsymbol{W} \sim \mathcal{N}(\boldsymbol{A}\boldsymbol{\mu}_x + \boldsymbol{b}, \boldsymbol{A}\boldsymbol{\Sigma}_x\boldsymbol{A}^{\mathrm{T}})$.

In freeing the user from explicitly coding these results, Dauphin supports the writing of a more natural and succinct version of the Kalman filter. In Dauphin, we need only define the random processes and their relationships

$$\boldsymbol{w}_k \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_k) \tag{27}$$
$$\boldsymbol{v}_k \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_k) \tag{28}$$
$$\hat{\boldsymbol{X}}_k = \boldsymbol{F}_k \hat{\boldsymbol{Y}}_{k-1} + \boldsymbol{w}_k \tag{29}$$
$$\hat{\boldsymbol{Z}}_k | \hat{\boldsymbol{X}}_k = \boldsymbol{H}_k \hat{\boldsymbol{X}}_k + \boldsymbol{v}_k \tag{30}$$
$$\hat{\boldsymbol{Y}}_k = \hat{\boldsymbol{X}}_k | \hat{\boldsymbol{Z}}_k \tag{31}$$

Using Bayes rule, the Dauphin compiler automatically calculates the *a posteriori* state $\hat{\boldsymbol{Y}}_k \sim \mathcal{N}(\boldsymbol{\mu}_{k|k}, \boldsymbol{\Sigma}_{k|k})$ from the measurement and the *a priori* state $\hat{\boldsymbol{X}}_k \sim \mathcal{N}(\hat{\boldsymbol{x}}_{k|k-1}, \boldsymbol{\Sigma}_{k|k-1})$ according to

$$\boldsymbol{\Sigma}_{k|k} = (\boldsymbol{\Sigma}_{k|k-1}^{-1} + \boldsymbol{H}^T \boldsymbol{R}^{-1} \boldsymbol{H})^{-1} \tag{32}$$
$$\boldsymbol{\mu}_{k|k} = \boldsymbol{\Sigma}_{k|k} (\boldsymbol{H} \boldsymbol{R}^{-1} \boldsymbol{z} + \boldsymbol{\Sigma}_{k|k} \boldsymbol{\Sigma}_{k|k-1}^{-1} \boldsymbol{\mu}_{k|k-1}) \tag{33}$$

This means rather than encoding Equations 13–21, the signal processor need only encode the high level description of the Kalman filter (Equations 27–31) as in Listing 8.

Listing 8: The Kalman filter in Dauphin

```
1 KalmanFilter(Z, Y0, F, w, H, v) = {
2   Y where
3     Y{0}: Y0;                              //Initial
4     Y{k>0}: X{k} = F*Y{k-1} + w{k},       //Predict
5             Z{k}|X{k} = H*X{k} + v{k},
6             Y{k} = X{k}|Z{k};              //Update
7   }
```

Here, Line 2 starts the definition of a Dauphin stream (*i.e.* infinite sequence) of random variables named Y. Line 3 initialises the stream at index 0 with Y{0} = Y0. Note that Y0 is passed to the `KalmanFilter` function as its second argument on Line 1.

Line 4 begins the stream definition for indexes k>0. The definitions on Lines 4–6 conform to a pattern that invokes the compiler to apply Bayes rule automatically to implicitly calculate the mean and covariance of Y{k} (using Equations 32–33). This is repeated, producing an element of stream Y from each element of stream Z, until the measurements from stream Z are exhausted.

To illustrate one use of the `KalmanFilter` function, assume a stream of measurements arrive from a real-time source available to the user via standard input.

Listing 9: Processing measurements from standard input in C

```
1 Dau_Stream Z = Dau_OpenStream(stdin);
2 Dau_Stream Y = KalmanStream(Z);
3 RV Yn;
4 for(;;) {
5   Yn = Dau_StreamNext( Y );
6   if (!Yn) break;
7   display(Yn);
8   }
```

Listing 9 shows code to read a stream of measurements and pass them through the Kalman filter which returns a stream of state estimates to be displayed. The code reflects the high level abstraction of the Kalman filter as a function that transforms one stream into another: in this case, the measurement stream into a stream of state measurements.

Variable Z is set to the measurement stream on Line 1. The Kalman filter transforms the measurement stream into the stream of state measurements Y on Line 2.

Each state estimate is a Normal random variable (Line 3) and each call to `Dau_StreamNext` (Line 5) requests the next state estimate Yn from the stream Y. Since Y is, by the definition on Line 2, dependent on the stream of measurements Z, the next measurement is requested automatically and passed to the `KalmanStream` function which returns the next state estimate. Finally, the state estimate is displayed (Line 7).

The whole process runs in a loop (Line 4) until the measurement stream terminates which in turn terminates the state estimate stream (Line 6).

The setting up of the Kalman filter parameters can be done in Dauphin as indicated by the simple example of Listing 10.

Listing 10: The Kalman filter parameters in Dauphin

```
KalmanStream(Z) = {
  let Q  = [1 1; 1 1];
      w  = NormalRV(0, Q);
      R  = [1 1; 1 1];
      v  = NormalRV(0, R);
      F  = [1 0; 1 1];
      H  = [1 0];
      Y0 = [0; 1];
  in KalmanFilter(Z, Y0, F, w, H, v);
}
```

IV. Implementation

Dauphin was introduced as a Python signal processing library in [11]. It has evolved into a compiled, strongly-typed, declarative language with type inferencing. Type inferencing is the Dauphin compiler's ability to infer, in the absence of type declarations, the types of variables and expressions as well as the signatures of functions from the source code, freeing the user from explicit type declarations as is necessary in C. Many modern languages employ type inferencing. The Dauphin compiler is written in Haskell [12], a functional language with a rich type system and powerful pattern matching capabilities, ideal for implementing compilers for new languages. A Dauphin program is compiled to standard C (the C99 standard) which is then compiled to object files. This means Dauphin is code multiplatform, able to run on any system for which there is a C compiler.

Extensive use is made of the GNU Scientific Library (GSL) [13], for matrix operations and the extensive library of mathematical and statistical functions. Using GSL, Dauphin has implementations of the same binary matrix operators (and their dot-prefix counterparts) that are available in Matlab and Octave $(+, -, *, /, \backslash$ and $\hat{})$.

Dauphin augments these operators with a variant of Matlab's and Octave's *binary singular expansion* or *broadcasting* functionality (`bsxfun`). For example, Matlab makes it possible to add a $1 \times M$ (row) matrix A to every row of a $N \times M$ matrix B with the call

```
bsxfun(plus,A,B)
```

The function `bsxfun` makes N copies of A to ensure it is a compatitible size for matrix addition with B and then does the addition. Octave also has the `bsxfun` function but with its version of broadcasting, allows standard operators such as $+$ (addition) to also implicitly resize A before adding[3]. So bsxfun(plus,A,B) can be achieved in Octave simply as A+B. In Dauphin, a user must explicitly indicate that they want matrix expansion by prefixing operators with an @, as in `A @+ B`. By being explicit, the compiler can support the user to produce correct programs by reporting when matrix sizes are nonconformable for the matrix operation. In future versions, Dauphin will extend broadcasting from operators to any function with two arguments.

In addition to GSL, initial experiments in using open parallelism software such as OpenMP suggest only minor modifications to the compiler will result in speed gains for some application code on multicore systems.

Interfaces from Dauphin to C and from Dauphin to Octave exist now but more interfaces are under development to transparently access other libraries that are useful for signal processing. The popularity of Python and the comprehensive mathematical libraries of Sage [14], particularly its symbolic capabilities, makes interfaces from Dauphin to both Python and Sage the next priorities. Interfaces to Dauphin from languages other than C are important to develop as well, so code that is written more naturally in Dauphin can be used by systems written in popular signal processing languages such as Python, Matlab and Octave.

V. Conclusion

Software languages play a significant role in the ability of researchers to express and solve problems in their particular problem domain. Our aim is to develop a language that focuses on the algorithmic structures of signal processing. Probability distributions, random variables and random processes are as fundamental and easy to use in Dauphin as numeric types are in other languages. Dauphin allows you to code these algorithms directly, so they can be coded once and put into libraries for future use. Ultimately, Dauphin aims to extend the power of the researcher by allowing them to focus on the real problems of signal processing and simplify the process of implementing their ideas.

[3]Octave allows the user to choose whether implicit resizing should throw an error, be allowed with a warning, or to proceed silently. A user risks errors if they allow broadcasting to proceed without warnings. Dauphin circumvents the issue by making broadcasting explicit by prefixing operators with the '@' symbol.

References

[1] The MathWorks Inc., *Matlab 8.5 (2015a)*, Natick, Massachusetts, 2015.

[2] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring, *GNU Octave version 3.8.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform, 2015.

[3] Wolfram Research Inc., *Mathematica Version 10*. Champaign, Illinois: Wolfram Research, Inc., 2015.

[4] Maplesoft, *Maple 2015*. Waterloo, Ontario: a division of Waterloo Maple Inc.

[5] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2010, iSBN 3-900051-07-0.

[6] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[7] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *ArXiv e-prints*, Sep. 2012.

[8] Scilab Enterprises, *Scilab: Free and Open Source software for numerical computation*, Scilab Enterprises, Orsay, France, 2012. [Online]. Available: http://www.scilab.org

[9] R. Duda, P. Hart, and D. Stork, *Pattern Classification (2nd Ed.)*. John Wiley and Sons Inc., 2000.

[10] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[11] R. Kyprianou, B. Moran, and P. Schachte, "Dauphin: A new statistical signal processing language," in *Radar (Radar), 2013 International Conference on*, Sept 2013, pp. 464–469.

[12] S. Peyton Jones *et al.*, "The Haskell 98 language and libraries: The revised report," *Journal of Functional Programming*, vol. 13, no. 1, pp. 0–255, Jan 2003, http://www.haskell.org/definition/.

[13] M. Galassi, *GNU Scientific Library Reference Manual (3rd Ed.)*, 2009.

[14] W. Stein and D. Joyner, "Sage: System for algebra and geometry experimentation," *ACM SIGSAM Bulletin*, vol. 39, no. 2, p. 64, 2005.