Case Study 1: Interfacing C++ with Python for Blahut-Arimoto Simulations

Kieran Morris

This case study will introduce an important algorithm for data transmission and channel capacity, then demonstrating how we have used both `C++` compatability and access to the University of Bristol `bluecrystal` supercomputer to optimise our computations over large dimensional (and sample size) data. Unfortunately we began work in python so this will be the main language we use, including its compatability with `C++`. However we hope that this will still be educational and should compliment the topic on implimenting `R` and `C++` with the `Rccp` package.

The Blahut Arimoto Algorithm

The Blahut-Arimoto algorithm is an iterative algorithm that computes the capacity of a channel, a channel can be thought of as a box which some signal can be parsed through and causes some sort of distortion. The capacity of a channel is the maximum rate at which information can be sent through the channel with an arbitrarily small error rate, i.e it gives a lower bound on how much redundancy we must add to guarantee (asymptotically) error free communication.

If we imagine the input and output signals X and Y as catagorical random variables on the sets $\{1, ..., m\}$ and $\{1, ..., n\}$, we can define the channel as a conditional probability distribution $P_{Y|X}$, or more concretely a matrix $P_{ij} = P(Y = j|X = i)$. Then we can write $Y = XP$. The channel coding theorem states that the capacity of a channel is given by the formula

$$C = \max_X I(X; Y) = \max_X I(X, XP)$$

where $I(X; Y)$ is the mutual information between the input and output signals. The Blahut-Arimoto algorithm is an iterative algorithm that computes the capacity of a channel by finding the optimal input distribution P_X that maximises the mutual information. We won't dive into the specifics here but the general idea is that this single max problem can be split into a maxmax problem, where the smaller maximisations correspond to analytic substitutions of parameters into equations. This boils down to the algorithm being a `for` loop. Below is my original python code which performs Blahut-Arimoto.

```
import numpy as np
import random as random

def arimito(prior,
            channel_matrix,
            log_base: float = 2,
            thresh: float = 1e-12,
            maxiter: int = 1e3,
            display_results = True):
    '''
    Consider I(X,Y;P), where P is the channel matrix, i.e P_ij = p(y=i|x=j)
```

```
    We attempt to maximise I with the blahut amirito algorithm
    -------------------------------------------------------
    prior: Inital guess for maximal X
    channel_matrix: is the matrix representing the transition from X to Y, i.e P
    log_base: base of logarithm, typically 2 or nats
    thresh: threshold to finish algorithm, when iterations are < thresh apart
    max_iter: maximum number of iterations before giving up
    -------------------------------------------------------
    BA is a maxmax algorithm, each maximisation has a closed form expression
    We perform the maximisation as a for loop
    '''
    if display_results:
        print("Arimoto: PYTHON EDITION")
    #Check we have a valid dimension for the prior
    assert prior.shape[0] > 1
    #Checkl prior is a valid probability distribution
    assert np.abs(prior.sum() - 1) < 1e-6
    #Check we have a valid dimension for the channel matrix
    n,m = channel_matrix.shape
    m_1 = prior.shape[0]
    assert m == m_1
    #Initialise the prior and Phi
    p = prior
    p_route = np.array([p])
    P = channel_matrix
    W = np.zeros((m,n))
    if display_results:
        print(f"Prior for p(x): {p}")
        print(f"Channel matrix p(y|x): \n {channel_matrix}")

    for iter in range(int(maxiter)):
        q = np.zeros(m)
        #Maximise I(p,W;P) over W
        for i in range(n):
            for j in range(m):
                W[j][i] = (P[i,j]*p[j])/np.dot(P[i,:],p)

        #Maximise I(p,W;P) over p
        r = np.zeros(m)
        for j in range(m):
            r[j] = np.exp(np.dot(P[:,j],np.log(W[j,:])))
        for i in range(m):
            q[i] = r[i]/np.sum(r)

        #Add to array of p values
        p_route = np.append(p_route,[q],axis=0)

        #Check if we have converged
        if np.linalg.norm(q-p) < thresh:
            break
        p = q

    #Calculate the capacity
```

```
    C = 0
    for i in range(n):
        for j in range(m):
            if p[j] > 0:
                C += p[j]*P[i][j]*np.log(W[j][i]/p[j])
    C = C/np.log(log_base)
    if display_results:
        print('Max Capacity: ', C)
        print('ArgMax: ', p)
        print("--------------------")
        print("--------------------")
    return C,p,p_route
#----------------------------------------
```

As a demonstration of the algorithm, we can consider a simple binary channel with a transition matrix P given by:

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{bmatrix}$$

and a prior distribution $p(x)$ given by: $p(x=0) = 0.8$ and $p(x=1) = 0.2$. In fact the maximisation for this type of channel (a binary symmetric channel) has an analytic solution for the capacity, which is given by:

$$C = 1 - H(0.9)$$

where $H(p)$ is binary entropy function, and the maximising distribution is given by $p(y=0) = p(y=1) = 0.5$. We can then run the algorithm with the following code:

```
random.seed(123)
import Arimito as Arimito
import time

start = time.time()
Arimito.binary_symmetric_channel_simulation(prior = np.array([0.8,0.2]), e = 0.1)
```

```
## --||Binary Symmetric Channel Simulation||--
## --------------------
## Python Capacity: 0.5310044064107188
## Python ArgMax: [0.5 0.5] :: <class 'numpy.ndarray'>
## True Capacity:  0.5310044064107188
## --------------------
```

```
end = time.time()
print(f"Time taken: {end-start} seconds")
```

```
## Time taken: 0.005585908889770508 seconds
```

The function is defined also in the `Arimito.py` file, which interfaces with other files which we will use in this case study, the link to the repo containing them can be found here.

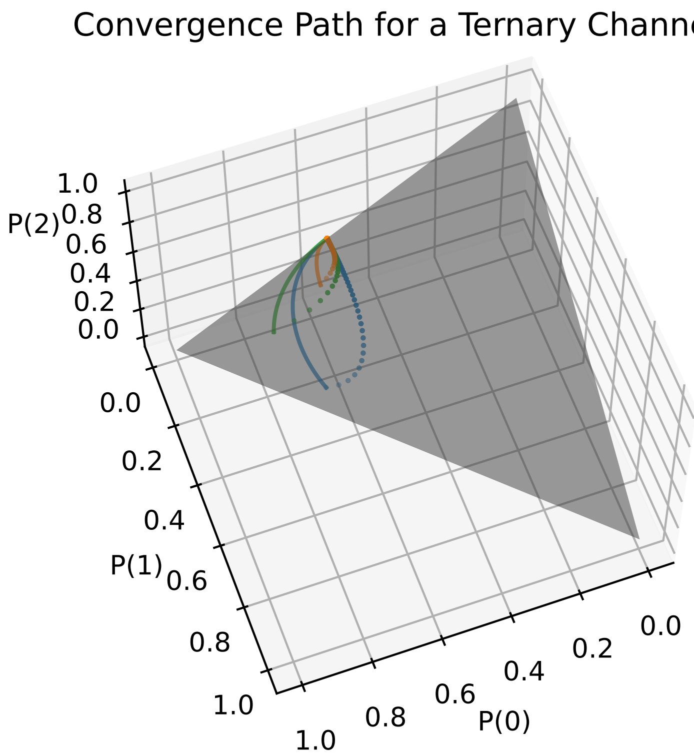
Our search space, for an n-ary input space, is an $(n-1)$-simplex, and of course as the dimension of the input space increases, the number of iterations required to converge increases too. Below we run the algorithm for a 3-ary input space, with 3 different priors and a random channel matrix, and plot the path taken by the algorithm to converge.

```
import Arimito_Generator as Generator
import Arimito_Convergence as Convergence
```

```
## INFO: Using numpy backend
```

```
random.seed(123)

channel_Matrix = Generator.random_channel_matrix(xdim = 3,ydim = 3)
start = time.time()
Convergence.plot_path_ternary(n=3,include_geodesic = True)
```

```
## Arimoto: PYTHON EDITION
## Prior for p(x): [0.65299733 0.28387738 0.06312529]
## Channel matrix p(y|x):
##  [[0.70641094 0.26179374 0.20259432]
##  [0.2864822  0.5621832  0.44053478]
##  [0.00710686 0.17602306 0.3568709 ]]
## Max Capacity:  0.2771501279968305
## ArgMax:  [5.44873594e-01 9.69231957e-12 4.55126406e-01]
## -------------------
## -------------------
## Arimoto: PYTHON EDITION
## Prior for p(x): [0.59342997 0.0795325  0.32703753]
## Channel matrix p(y|x):
##  [[0.70641094 0.26179374 0.20259432]
##  [0.2864822  0.5621832  0.44053478]
##  [0.00710686 0.17602306 0.3568709 ]]
## Max Capacity:  0.2771501279968468
## ArgMax:  [5.44873594e-01 9.55556949e-12 4.55126406e-01]
## -------------------
## -------------------
## Arimoto: PYTHON EDITION
## Prior for p(x): [0.74260096 0.10357653 0.15382251]
## Channel matrix p(y|x):
##  [[0.70641094 0.26179374 0.20259432]
##  [0.2864822  0.5621832  0.44053478]
##  [0.00710686 0.17602306 0.3568709 ]]
## Max Capacity:  0.2771501279968335
## ArgMax:  [5.44873594e-01 9.66662768e-12 4.55126406e-01]
## -------------------
## -------------------
```



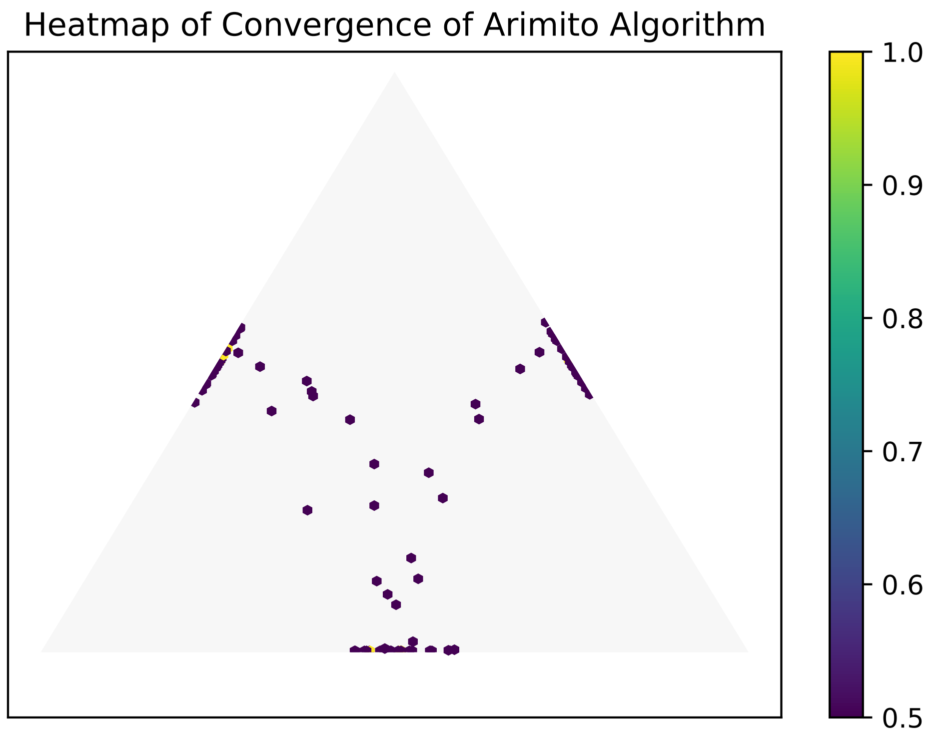

```
end = time.time()
print(f"Time taken: {end-start} seconds")
```

```
## Time taken: 0.4546070098876953 seconds
```

We also plotted the geodesic path (with respect to the fisher information matrix) along the simplex to study how well the algorithm converges. One thing you may notice in the above example is that the optimal solution lies on the boundary of the simplex, i.e one of the probabilities is 0. This might seem kind of weird, since we previously discussed that we have an infinite family of solutions at $p = (1/3, 1/3, 1/3)$. The question became, what does a typical solution look like? We answer this question by simulating a collection of random channel matrices and priors, and then plotting the heatmap of solutions. This is where computational complexity begins to creep in, as the below simulation took my computer ~2 minutes, and the number of simulations is only 10000, with `xdim = 3`. For the below code we use the package `ternary` which helps with heatmaps on simplexes.

```
import ternary

start = time.time()
#Create function which performs arimito multiple times and outputs the distribution
def arimito_multiple(prior,log_base: float = 2, thresh: float = 1e-12, maxiter: int = 1e3, n: int = 1):
    '''
    Perform the arimito algorithm n times and output the distribution
    '''

    p = []
```

```
    for i in range(n):
        random_binary_channel = Generator.random_channel_matrix(xdim = 3,ydim = 3)
        p.append(arimito(prior,random_binary_channel,log_base=log_base,thresh=thresh,maxiter=maxiter,di
    return p


#Perform the arimito algorithm multiple times
p = arimito_multiple(prior = np.array([0.2,0.3,0.5]),n=100)

p = np.array(p)*100

heatmap_data = {}

for i in range(p.shape[0]):
    # Round the points to 2 decimal places and convert to a tuple
    point = tuple(np.round(p[i, :], 1))
    # Add the point to the heatmap data
    if point in heatmap_data:
        heatmap_data[point] += 1
    else:
        heatmap_data[point] = 1

# Create the figure
fig, tax = ternary.figure(scale=100)
# Normalize the values in the heatmap data
max_value = max(heatmap_data.values())
heatmap_data = {key: value / max_value for key, value in heatmap_data.items()}
# Plot the heatmap
tax.heatmap(heatmap_data, style="hexagonal", use_rgba=False)

# Set labels
tax.set_title("Heatmap of Convergence of Arimito Algorithm")
tax.left_corner_label("$p_0$")
tax.right_corner_label("$p_1$")
tax.top_corner_label("$p_2$")

# Remove default Matplotlib Axes
tax.clear_matplotlib_ticks()

ternary.plt.show()
```

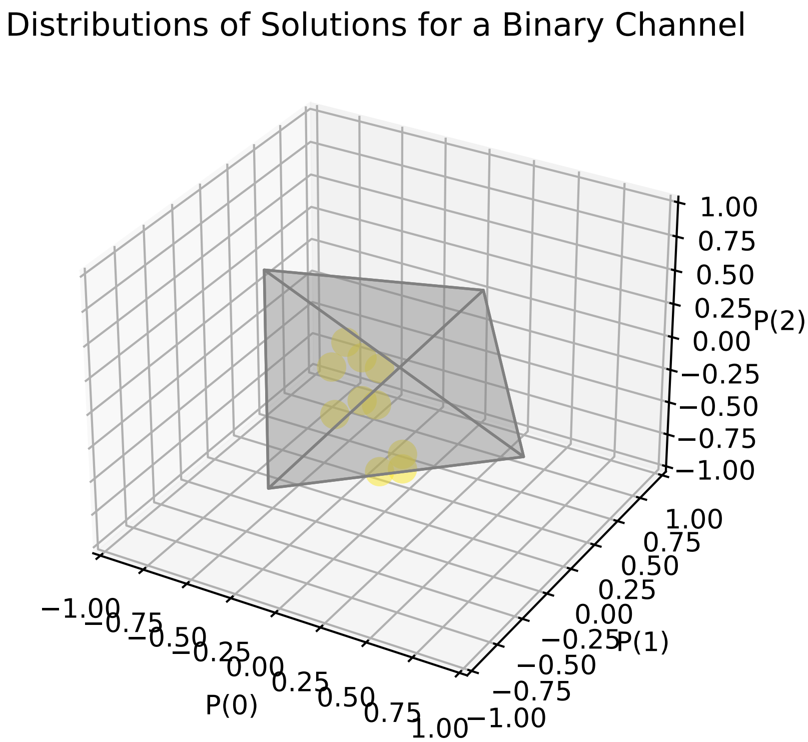



```
end = time.time()

print(f"Time taken: {end-start} seconds")
```

```
## Time taken: 1.4777138233184814 seconds
```

As you can see, counterintuitively, the optimal solution is often on the boundary of the simplex. Which is a great observation, however as it took so long it's hard to get a good measure of the behaviour. Below is our attempt to also plot the density plot for a tertiary input space (m=4), which took even longer. Also note that its hard to plot a 3D heatmap.

```
Convergence.simulate_distribution_quarterly(n=10)
```



This took even longer, and is beginning to slow my computer down. Since the arimito algorithm is fairly rudamental (its just a for loop where we update probabilities) we rewrote the `arimito` function in C instead. We also used python to interface with C to preserve our codebase.

Python Interface to C++

Python has a slightly different method of interfacing with `C++`, which come from a few different packages, we chose to use `pybind11` as that's what google said was good, plus it works so that's good. All it requires is installation and inclusion of a few lines of code in your `C++` code. Firstly we pip install.

```
pip install pybind11
```

The technique for mixing C++ with python is very simple, we essentially make a local package which we can import like any other python package - this is probably a side affect of python being built on C. We only need to include the following code in our preamble:

```
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
#include <pybind11/stl.h>


namespace py = pybind11;
```

to tell the compiler that we are using `pybind11`. This plays a similar role to `#include <Rcpp.h>` in in R. Then at the bottom we included the command:

```
PYBIND11_MODULE(CArimito, m) {
    m.def("arimito", &arimito, "Arimoto algorithm");
}
```

In our context we specified `CArimito` to be the name of our 'package' and the function `arimito` to be the function we want to export to python. With this setup, pybind does all the work for us, so long as we compile. Which we do by going to the working directory of our `.cpp` file and running:

```
c++ -O3 -Wall -shared -std=c++11 -fPIC `python3
-m pybind11 --includes` Arimito.cpp
-o CArimito`python3-config --extension-suffix`
```

in the terminal. This creates `CArimito.cpython-310-x86_64-linux-gnu.so` which can be imported into python via:

```
import CArimito
```

We can then call our function `arimito` as if it were a python function, so specifically we can run the following code:

```
CArimito.arimito(channel_matrix_list,prior_list,1000,1e-12)
```

Although to make the function more user friendly we created the following wrapped to match it up with or python edition:

```
def Carimito(prior,
             channel_matrix,
             log_base: float = 2,
             thresh: float = 1e-12,
             maxiter: int = 1e3,
             display_results = True):
    '''
    Consider I(X,Y;P), where P is the channel matrix, i.e P_ij = p(y=i|x=j)
    We attempt to maximise I with the blahut amirito algorithm
    This is a wrapper for the C implementation of the Arimito algorithm, it is dependent on the CArimit
    ---------------------------------------------------------
    prior: Inital guess for maximal X
    channel_matrix: is the matrix representing the transition from X to Y, i.e P
    log_base: base of logarithm, typically 2 or nats
    thresh: threshold to finish algorithm, when iterations are < thresh apart
    max_iter: maximum number of iterations before giving up
    ---------------------------------------------------------
    BA is a maxmax algorithm, each maximisation has a closed form expression
    We perform the maximisation as a for loop
    '''
    if display_results:
        print("Arimoto: C EDITION")
    #Check we have a valid dimension for the prior
    assert prior.shape[0] > 1
    #Checkl prior is a valid probability distribution
    assert np.abs(prior.sum() - 1) < 1e-6
```

```
    #Check we have a valid dimension for the channel matrix
    n,m = channel_matrix.shape
    m_1 = prior.shape[0]
    assert m == m_1
    if display_results:
        print(f"Prior for p(x): {prior}")
        print(f"Channel matrix p(y|x): \n {channel_matrix}")

    prior_list = prior.tolist()
    channel_matrix_list = channel_matrix.tolist()
    p,C,p_route = CArimito.arimito(channel_matrix_list,prior_list,1000,1e-12)
    if display_results:
        print('Max Capacity: ', C)
        print('ArgMax: ', p)
        print("--------------------")
        print("--------------------")
    return C,p,p_route
```

As a demonstration, lets recreate the output from our intial symmetric examples and three random prior examples.

```
Channel_Matrix = Generator.symmetric_channel_matrix(xdim = 2,ydim =2,e= 0.1)
print("Python Edition")
```

```
## Python Edition
```

```
start = time.time()
C,p,p_route = Arimito.Carimito(prior = np.array([0.8,0.2]),channel_matrix = Channel_Matrix)
```

```
## Arimoto: C EDITION
## Prior for p(x): [0.8 0.2]
## Channel matrix p(y|x):
##  [[0.9 0.1]
##  [0.1 0.9]]
## Max Capacity:  0.5310044064107188
## ArgMax:  [0.5 0.5]
## --------------------
## --------------------
```

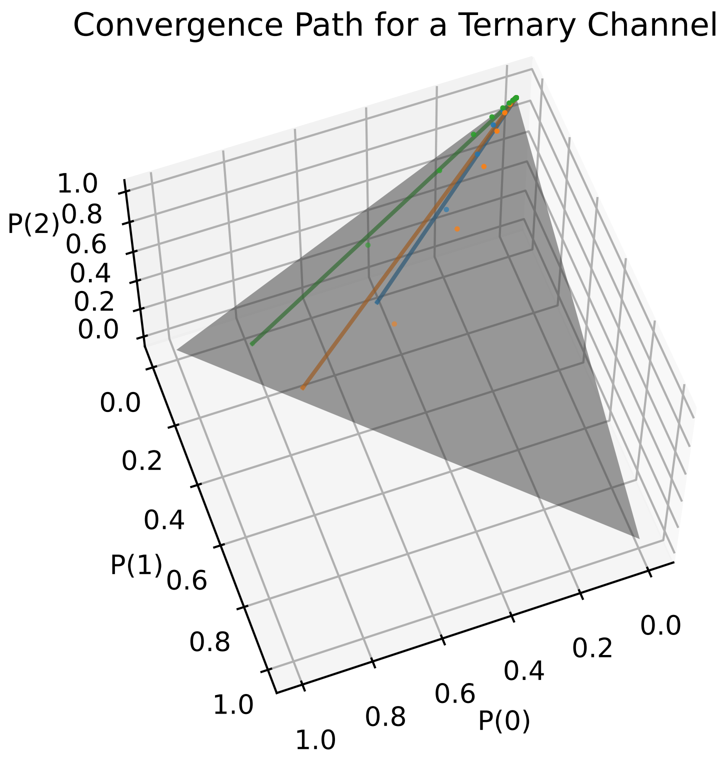
```
end = time.time()
print(f"Time taken: {end-start} seconds")
```

```
## Time taken: 0.005236387252807617 seconds
```

Hilariously, in this simple case the difference is by a measily 0.001, lets try our 3 random priors examples and see the result:

```
random.seed(123)
start = time.time()
Convergence.plot_path_ternary(n=3,include_geodesic = True,lang = "C++")
```

```
## Arimoto: C EDITION
## Prior for p(x): [0.4676667  0.19148926 0.34084404]
## Channel matrix p(y|x):
##  [[0.70641094 0.26179374 0.20259432]
##  [0.2864822  0.5621832  0.44053478]
##  [0.00710686 0.17602306 0.3568709 ]]
## Max Capacity:  2.1819892261872954e-14
## ArgMax:  [2.05274355e-14 3.60433549e-86 1.00000000e+00]
## -------------------
## -------------------
## Arimoto: C EDITION
## Prior for p(x): [0.71110665 0.25219101 0.03670234]
## Channel matrix p(y|x):
##  [[0.70641094 0.26179374 0.20259432]
##  [0.2864822  0.5621832  0.44053478]
##  [0.00710686 0.17602306 0.3568709 ]]
## Max Capacity:  1.2480006645133402e-14
## ArgMax:  [1.17437228e-14 9.24902601e-74 1.00000000e+00]
## -------------------
## -------------------
## Arimoto: C EDITION
## Prior for p(x): [0.80440212 0.09607603 0.09952185]
## Channel matrix p(y|x):
##  [[0.70641094 0.26179374 0.20259432]
##  [0.2864822  0.5621832  0.44053478]
##  [0.00710686 0.17602306 0.3568709 ]]
## Max Capacity:  1.6693377511492987e-14
## ArgMax:  [1.57837716e-014 6.55776656e-130 1.00000000e+000]
## -------------------
## -------------------
```




```python
end = time.time()
print(f"Time taken: {end-start} seconds")
```

```
## Time taken: 0.07237744331359863 seconds
```

We're working at around about 4x as fast. Which should be useful with large simulations.

C++ Interface to bluecrystal

Results