# Cyclic Shift Binding & Bundling

## Introduction

These are the results for the cyclic shift unbinding using code:

Cyclic\_Shift\_V1\_comments.py

The code splits the functions into two Brian2 networks. Net1 performs the binding and stores the resulting sparse vector as P1\_timings.

Results presented below are from bundling 90 vectors with 1000 slots and 1024bits /slot – Threshold was set at Min\_Match-2 = 9. A memory size of 1000 random vectors is used to construct the P-Matrix.

## These are the measured bit positions following the argmax using the non-spiking code.

[38, 414, 638, 591, 230, 53, 916, 41, 946, 195, 984, 345, 7, 267, 10, 387, 216, 50, 603, 111, 31, 590, 4, 161, 411, 47, 8, 190, 157, 371, 473, 334, 168, 272, 119, 529, 121, 487, 12, 40, 545, 662, 250, 266, 200, 735, 72, 91, 145, 260, 301, 297, 197, 205, 289, 120, 576, 23, 450, 593, 377, 178, 330, 3, 328, 238, 389, 499, 211, 5, 21, 166, 50, 116, 235, 812, 209, 57, 51, 23, 118, 39, 502, 22, 53, 525, 241, 99, 32, 75, 78, 211, 806, 375, 143, 575, 30, 649, 296, 482, 440, 188, 673, 569, 259, 40, 179, 933, 146, 0, 26, 293, 809, 260, 586, 186, 382, 211, 109, 648, 53, 238, 20, 115, 427, 489, 56, 501, 268, 223, 202, 163, 230, 8, 71, 483, 432, 781, 945, 26, 116, 123, 46, 190, 7, 33, 646, 67, 283, 320, 293, 666, 188, 161, 74, 100, 339, 2, 491, 114, 37, 28, 29, 359, 33, 386, 86, 702, 1022, 367, 286, 175, 375, 400, 79, 65, 205, 560, 151, 36, 598, 20, 37, 741, 235, 255, 3, 613, 241, 32, 30, 33, 25, 204, 353, 16, 88, 55, 262, 1, 704, 29, 9, 894, 906, 209, 47, 167, 160, 303, 277, 34, 161, 19, 374, 175, 499, 559, 123, 84, 27, 35, 293, 297, 43, 47, 211, 321, 38, 75, 179, 223, 44, 179, 68, 99, 115, 14, 58, 60, 130, 377, 181, 543, 370, 352, 620, 270, 154, 94, 135, 77, 201, 74, 444, 81, 221, 79, 72, 353, 100, 194, 267, 609, 82, 439, 25, 56, 75, 178, 151, 50, 68, 38, 524, 20, 66, 218, 84, 198, 275, 774, 12, 578, 60, 993, 4, 56, 126, 389, 393, 220, 246, 103, 523, 51, 217, 54, 10, 375, 655, 272, 90, 257, 348, 239, 19, 839, 246, 71, 342, 485, 265, 372, 102, 385, 987, 574, 89, 192, 321, 86, 616, 111, 509, 344, 234, 133, 213, 555, 90, 558, 132, 16, 701, 59, 296, 241, 635, 13, 325, 507, 10, 789, 442, 353, 876, 33, 218, 77, 120, 17, 679, 131, 584, 747, 828, 114, 253, 211, 171, 332, 612, 221, 895, 829, 135, 272, 540, 225, 26, 358, 123, 108, 54, 745, 45, 233, 1, 766, 50, 627, 631, 556, 124, 443, 174, 130, 14, 252, 689, 18, 148, 857, 175, 794, 77, 60, 143, 923, 342, 505, 618, 840, 598, 147, 227, 80, 350, 398, 165, 380, 169, 850, 80, 170, 537, 23, 174, 171, 164, 542, 68, 676, 339, 458, 140, 155, 235, 81, 108, 708, 4, 4, 118, 50, 7, 300, 93, 284, 324, 35, 262, 107, 444, 242, 608, 273, 316, 103, 37, 4, 584, 630, 643, 884, 268, 259, 707, 84, 704, 319, 37, 206, 209, 71, 446, 111, 91, 594, 437, 443, 154, 61, 789, 193, 516, 31, 132, 540, 216, 77, 917, 108, 305, 61, 257, 62, 59, 159, 657, 460, 413, 239, 738, 26, 901, 146, 235, 27, 560, 622, 219, 501, 936, 270, 567, 230, 864, 410, 86, 216, 157, 8, 877, 19, 266, 150, 24, 555, 353, 231, 303, 29, 186, 132, 241, 318, 456, 0, 86, 28, 60, 180, 122, 6, 291, 459, 88, 924, 265, 103, 15, 469, 240, 394, 23, 253, 578, 698, 858, 174, 163, 677, 12, 805, 114, 371, 0, 477, 401, 183, 319, 491, 172, 411, 349, 399, 240, 251, 53, 432, 312, 397, 17, 141, 50, 224, 54, 342, 365, 656, 134, 435, 82, 174, 154, 21, 287, 855, 233, 217, 167, 695, 120, 360, 228, 577, 131, 52, 88, 248, 82, 897, 233, 41, 608, 8, 37, 97, 200, 656, 141, 514, 88, 330, 699, 442, 372, 241, 256, 79, 104, 318, 500, 981, 156, 450, 44, 67, 322, 331, 344, 78, 14, 51, 71, 298, 82, 28, 138, 692, 432, 375, 710, 4, 239, 78, 52, 134, 114, 455, 214, 16, 837, 101, 353, 525, 310, 43, 99, 86, 116, 587, 629, 639, 716, 1, 344, 608, 923, 737, 246, 49, 166, 386, 229, 111, 384, 10, 333, 648, 339, 88, 186, 559, 12, 549, 160, 379, 620, 505, 479, 169, 270, 587, 316, 261, 630, 422, 695, 812, 77, 289, 73, 68, 342, 91, 447, 63, 28, 155, 304, 4, 69, 629, 951, 44, 11, 682, 63, 152, 217, 66, 311, 127, 240, 734, 229, 557, 60, 235, 66, 76, 274, 156, 157, 172, 63, 41, 291, 545, 736, 60, 200, 593, 31, 39, 438, 818, 176, 38, 68, 127, 61, 288, 116, 329, 5, 167, 196, 67, 138, 356, 96, 96, 137, 187, 400, 333, 19, 101, 286, 364, 7, 599, 29, 100, 747, 294, 497, 35, 435, 604, 15, 91, 236, 39, 500, 830, 197, 101, 17, 395, 260, 133, 223, 32, 244, 713, 517, 70, 292, 319, 231, 533, 574, 55, 86, 810, 421, 229, 520, 355, 594, 84, 29, 158, 103, 64, 24, 156, 74, 99, 558, 226, 357, 82, 130, 111, 340, 245, 321, 820, 621, 51, 41, 254, 189, 193, 831, 332, 254, 128, 536, 408, 353, 227, 22, 664, 309, 529, 127, 176, 36, 87, 36, 122, 224, 17, 178, 553, 621, 118, 38, 161, 256, 260, 170, 18, 207, 430, 119, 431, 210, 12, 677, 338, 183, 229, 391, 917, 74, 68, 70, 361, 250, 218, 7, 872, 437, 94, 712, 734, 319, 58, 23, 36, 440, 176, 134, 345, 227, 652, 520, 74, 191, 310, 1007, 66, 248, 51, 193, 27, 95, 223, 484, 280, 613, 504, 301, 88, 98, 114, 302, 604, 115, 408, 290, 432, 321, 29, 139, 38, 13, 354, 929, 157, 348, 24, 91, 758, 285, 297, 157, 684, 20, 186, 411, 38, 594, 10, 814, 257, 20, 100, 90, 12, 29, 146, 629, 226, 340, 238, 256, 297, 937, 73, 46, 50, 48, 437, 323, 179, 351, 379, 257, 114, 47, 150, 40, 800, 149, 101, 243, 53, 21, 138, 51, 180, 16, 49, 268, 77, 735, 856, 403, 89, 227, 563]

## These are the measured number of matches following the unbinding using non-spiking code

0 0 30

1 1 23

2 2 28

3 3 25

4 4 25

5 5 22

6 6 17

7 7 23

8 8 14

9 9 21

10 10 22

11 11 21

12 12 26

13 13 23

14 14 20

15 15 26

16 16 29

17 17 12

18 18 18

19 19 27

20 20 26

21 21 24

22 22 33

23 23 18

24 24 26

25 25 19

26 26 27

27 27 21

28 28 24

29 29 21

30 30 20

31 31 27

32 32 19

33 33 17

34 34 27

35 35 23

36 36 24

37 37 28

38 38 20

39 39 20

40 40 23

41 41 28

42 42 22

43 43 22

44 44 25

45 45 23

46 46 19

47 47 17

48 48 27

49 49 27

50 50 24

51 51 17

52 52 25

53 53 31

54 54 21

55 55 20

56 56 22

57 57 16

58 58 17

59 59 21

60 60 21

61 61 26

62 62 28

63 63 24

64 64 20

65 65 27

66 66 22

67 67 19

68 68 27

69 69 18

70 70 22

71 71 26

72 72 28

73 73 19

74 74 30

75 75 23

76 76 31

77 77 22

78 78 24

79 79 30

80 80 25

81 81 33

82 82 20

83 83 35

84 84 14

85 85 27

86 86 30

87 87 21

88 88 19

89 89 22

Min\_match= 12

## Brian2 SNN Results

The following diagram shows the neuromorphic circuit for cyclic shift binding/bundling.



This circuit requires 3\*slots + memory\_size neurons which in this example is 4000 neurons.

In the following diagram we show the brian2 monitors as the binding/bundling progresses.

Diagram

Description automatically generated

The following are the measured spasre vector bit positions (integer values) after running the Brian2 SNN code i.e. following the argmax. They match exactly the expected values.

[ 38 414 638 591 230 53 916 41 946 195 88 345 7 267 10 387 216 50

603 111 31 590 4 161 411 47 8 190 157 371 243 334 168 272 119 529

121 487 12 40 545 662 250 266 200 450 72 91 145 260 301 297 197 205

289 120 576 23 136 593 377 178 330 3 328 238 389 499 211 5 21 166

50 116 235 2 61 57 51 23 118 39 502 22 53 525 241 99 32 75

78 211 806 375 143 575 30 649 296 12 440 188 673 91 218 40 179 250

146 0 26 293 809 260 227 152 382 211 109 648 53 238 20 115 427 489

56 122 268 223 202 163 230 8 71 483 432 781 51 26 116 123 46 190

7 33 431 67 283 320 293 234 188 161 74 100 339 2 491 114 37 28

29 359 33 386 86 442 231 144 286 175 375 234 79 65 205 92 151 36

598 20 37 741 235 255 3 297 241 32 30 33 25 204 353 16 88 55

262 1 704 9 9 894 906 209 47 167 160 303 277 34 161 19 374 175

499 559 123 84 27 35 293 297 43 47 211 321 38 75 179 223 44 179

68 99 115 14 58 60 130 377 181 543 370 352 620 270 154 94 135 77

201 74 444 81 221 79 72 353 100 194 267 609 82 439 25 56 75 178

151 50 68 38 524 20 66 218 84 198 275 774 12 578 60 993 4 56

126 389 290 220 246 103 523 51 217 54 10 375 655 272 90 257 348 239

19 839 246 71 342 485 18 372 102 385 987 574 89 192 153 86 616 111

509 344 234 133 213 555 90 558 132 16 701 59 296 167 635 13 325 507

10 789 442 353 268 33 218 77 120 17 128 131 584 747 177 114 253 211

171 332 612 221 20 273 135 272 540 225 26 358 123 108 54 117 45 149

1 270 50 627 265 556 124 443 174 130 14 252 689 18 148 857 175 794

77 60 143 923 342 505 618 19 598 147 227 80 350 398 165 380 169 207

80 170 109 23 174 171 164 542 68 676 339 458 140 155 235 81 108 708

4 4 118 50 7 300 93 232 324 35 262 107 444 242 608 273 316 103

37 4 584 630 643 13 268 259 707 84 704 319 37 206 209 71 446 111

91 594 270 443 154 61 252 193 516 31 132 540 216 77 154 108 305 61

257 62 59 159 657 108 413 208 738 26 363 146 235 27 560 622 219 27

936 270 97 230 299 410 86 29 157 8 877 19 266 150 24 555 353 231

303 29 186 132 241 318 456 0 86 28 60 180 122 6 291 459 88 924

265 103 15 469 240 394 23 253 92 43 858 174 163 677 12 805 114 371

0 10 401 183 319 491 172 40 349 399 240 251 53 432 312 397 17 141

50 224 54 342 365 656 134 435 82 174 154 21 287 843 233 217 167 695

120 360 228 577 131 52 88 248 82 100 233 41 608 8 37 97 200 656

141 514 88 330 699 272 372 241 256 79 104 318 500 322 156 450 44 67

322 331 344 78 14 51 71 298 82 28 138 692 432 375 710 4 239 78

52 134 114 455 214 16 837 101 353 525 102 43 99 86 116 587 629 639

716 1 344 608 923 149 246 49 166 386 229 111 384 10 333 648 339 88

186 559 12 549 160 98 620 505 479 169 270 36 316 261 630 422 695 812

77 289 73 68 342 91 447 63 28 155 304 4 69 629 134 44 11 682

63 152 217 66 311 127 45 734 229 557 60 235 66 76 274 156 157 172

63 41 291 545 736 60 200 593 31 39 438 281 176 38 68 127 61 288

116 329 5 167 196 67 138 356 96 96 137 187 400 333 19 101 286 364

7 599 29 100 747 294 497 35 435 604 15 91 236 39 500 830 197 101

17 95 260 133 223 32 244 713 50 70 292 319 231 533 574 55 86 810

421 229 520 355 594 84 29 158 103 64 24 156 74 99 558 226 357 82

130 111 340 245 321 820 621 51 41 254 189 193 831 332 254 128 536 408

353 227 22 664 309 42 127 176 36 87 36 122 224 17 178 538 621 118

38 161 256 260 170 18 207 430 119 431 210 12 677 338 183 229 391 178

74 68 70 361 250 218 7 574 437 94 621 279 319 58 23 36 440 176

134 345 227 120 520 74 191 310 37 66 248 51 193 27 95 223 484 280

613 504 301 88 98 114 302 604 115 268 290 432 321 29 139 38 13 354

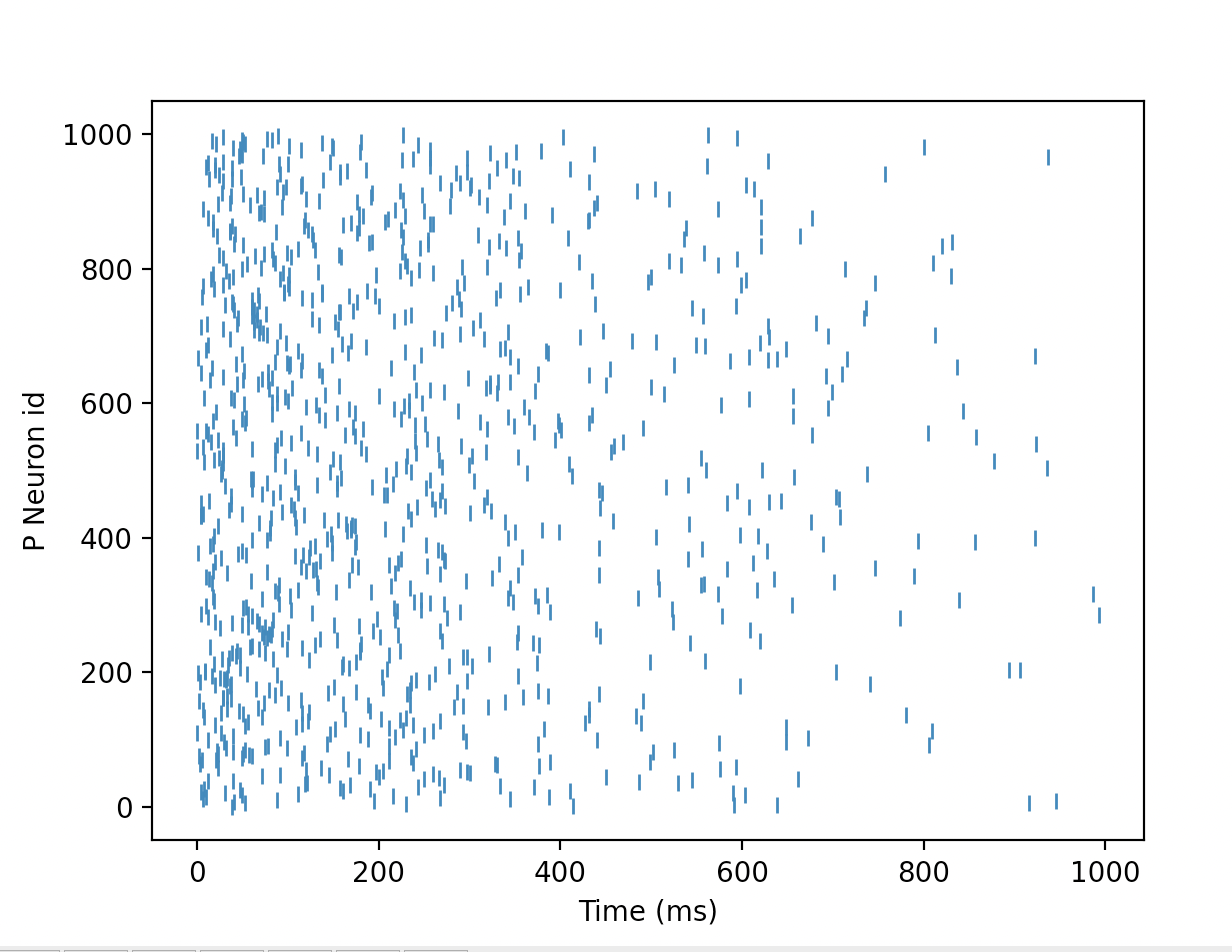
48 157 348 24 91 758 285 297 157 165 20 186 411 38 330 10 561 257

20 100 90 12 29 146 629 226 340 238 256 297 937 73 46 50 48 437

323 179 351 379 257 114 47 150 40 800 149 101 243 53 21 138 51 180

16 49 83 77 594 28 403 89 227 563]

The following diagram shows the Brian2 sparse vector monitor.

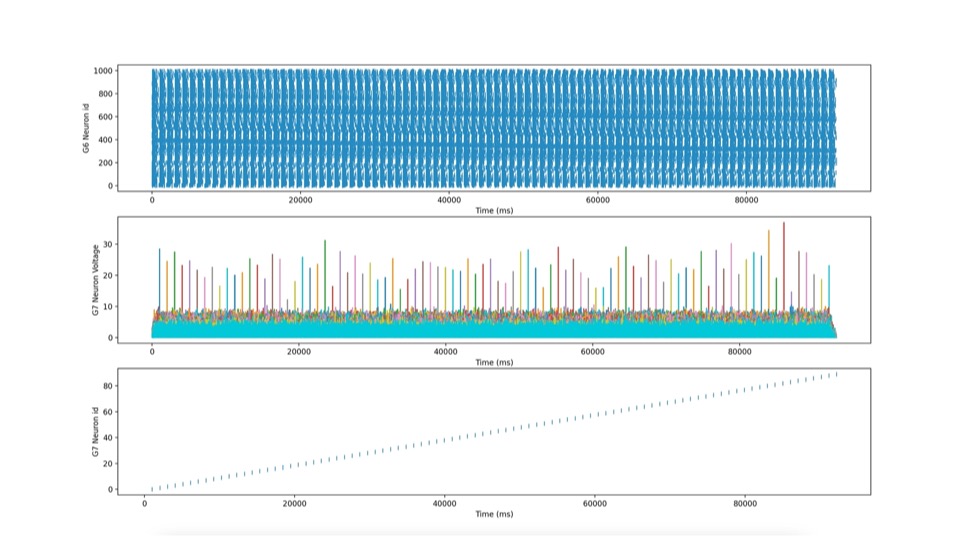


This sparse vector is now the input to the Net2 unbinding operations using the circuit shown in the following diagram.



This circuit requires slots + memory\_size neurons, which in this case is 2000 neurons.

In the following diagram we show the output as the G4 neurons unbind the input vector and the clean-up memory threshold voltage and spikes produced. Note that the order is the reverse of the input order.



## Conclusion

The results show that the binding/bundling capacity of the SNN is the same as that for the dense vectors using the equivalence of 1000 slots/neurons and 1024 bit positions per slot such that:

Slots \* log(bit\_positions) = 1000\*10 = 10,000.

## Annex A – The Code

from brian2 import \*

from brian2.equations import refractory

from brian2.monitors import spikemonitor

#import random

import matplotlib.pyplot as plt

import numpy as np

def visualise\_connectivity(S):

Ns = len(S.source)

Nt = len(S.target)

figure(figsize=(10, 4))

subplot(121)

plot(zeros(Ns), arange(Ns), 'ok', ms=10)

plot(ones(Nt), arange(Nt), 'ok', ms=10)

for i, j in zip(S.i, S.j):

plot([0, 1], [i, j], '-k')

xticks([0, 1], ['Source', 'Target'])

ylabel('Neuron index')

xlim(-0.1, 1.1)

ylim(-1, max(Ns, Nt))

subplot(122)

plot(S.i, S.j, 'ok')

xlim(-1, Ns)

ylim(-1, Nt)

xlabel('Source neuron index')

ylabel('Target neuron index')

#This code uses the Brian2 neuromorphic simulator code to implement

# a version of cyclic shift binding and unbinding based on the

# paper :High-Dimensional Computing with Sparse Vectors" by Laiho et al 2016.

# The vector representation is a block structure comprising slots

# where the number of slots is the vector dimension. In each slot there are a

# number of possible bit positions with one bit set per slot.

# In this implementation we implement the cyclic shift binding and unbinding

# operations in Brian2 by representing each slot as a neuron and the time delay

# of the neuron's spike as the bit position.

# To ensure that the Brian2 network is performing correctly the first section of the code

# computes the expected sparse bound vector.

# The neuromorphic equivalent is implemented as two Brian2 networks. The first network (net1) implements

# the cyclic binding and the second netwok (net2) implements the cyclic shift unbinding and the clean-up memory

# operation which compares the unbound vector with all the memory vectors to find the best match.

# The sparse bound vector resulting from net1 is passed to net2.

# Initialise the network parameters

slots = 100 # This is the number of neurons used to represent a vector

bits = 100 # This is the number of bit positions

mem\_size = 1000 # The number of vectors against which the resulting unbound vector is compared

Num\_bound = 5 # The number of vectors that are to be bound

input\_delay = bits # Time delay between adding cyclically shifted vectors to construct the bound vector is set to 'bits' milliseconds.

#NB all timings use milliseconds and we can use a random seed if required.

#np.random.seed(654321)

y\_low=0 # This is used to select the lowest index of the range of neurons that are to be displayed

y\_high=3 # This is used to select the highest index of the range of neurons that are to be displayed

delta = (Num\_bound+1)\*bits #This determins the time period over which the Brian2 simulation is to be run.

# Generate a random matrix (P\_matrix) which represents all of the sparse vectors that are to be used.

# This matrix has columns equal to the number of slots in each vector with the number of rows equal to the memory size (mem\_size)

P\_matrix = np.random.randint(0, bits, size=(mem\_size,slots))

#print(P\_matrix)

for n in range(0,Num\_bound):

print(P\_matrix[n])

print()

#--------------------------------------------------------------------------------------------------------------

#This section of the code computes the theoretical values for the sparse vector (which can then be compared with

# the output of the net1 neuromorphic circuit. It then computes the expected number of bits that will align in the

# clean-up memory operation (which can then be compared with the net2 neuromorphic circuit output).

#print the cyclically shifted version of the vectors that are to be bound

for n in range (0,Num\_bound):

print(np.roll(P\_matrix[n],n))

#Create sparse bound vector (s\_bound) of zeros

s\_bound=[]

for s in range(0,slots):

s\_bound.append(np.zeros(bits))

print()

#Create sparse representation of the bound vector

# - i.e. read the bit position for the shifted vector from the rows of the P\_matrix

# and add the vectors together to get s\_bound

for n in range(0,Num\_bound):

for s in range(0,slots):

#b = np.roll(P\_matrix[n],Num\_bound-1-n)[s]

b = np.roll(P\_matrix[n],n)[s]

s\_bound[s][b] += 1

#make s\_bound sparse using the argmax function - NB this will take the first bit position if

#non of the bits adds to greater than 1.

sparse\_bound=[]

for s in range(0,slots):

sparse\_bound.append(np.argmax(s\_bound[s]))

print()

print(sparse\_bound)

print()

#The following unbinds the sparse\_bound vector and compare with each of the vectors in the P\_matrix couting the

#number of slots that have matching bit positions. This gives the number of spikes that should line up

#in the clean up memory operation.

min\_match=slots

for n in range(0,Num\_bound):

for m in range(0,Num\_bound):

match=0

for s in range(0,slots):

# if P\_matrix[n][s] == np.roll(sparse\_bound,-(Num\_bound-1-m))[s]:

if P\_matrix[n][s] == np.roll(sparse\_bound,-m)[s]:

match +=1

if n==m:

print(n,m,match)

if match<=min\_match:

min\_match = match

#When we print the maximum value of match should occur when m=n

print('Min\_match=',min\_match)

#---------------------------------------------------------------------------------------------------------------

#This section of the code implements the cyclic shift binding in the Brian2 network (net1)

net1=Network()

#We first create an array of time delays which will be used to select the first Num\_bound vectors from

# the P\_matrix with a time delay (input\_delay) between each vector.

array1 = np.ones(mem\_size)\*slots\*bits

for b in range(0,Num\_bound):

array1[b] = (Num\_bound-b-1)\*input\_delay

# print (array1[b])

#We use the array1 timedelay matrix to trigger a SpikeGeneratorGroup of neurons that generates the

# required spike triggers and add this to the network.

P = SpikeGeneratorGroup(mem\_size,np.arange(mem\_size), (array1)\*ms)

net1.add(P)

#We now define the set of equation and reset definitions that will be used to generate the neuron action

#potentials and spike reset operations. Note that we make use of the Brian2 refractory operation.

equ1 = '''

dv/dt = -v/tau : 1 (unless refractory)

vt : 1

tau : second

'''

equ2 ='''

dv/dt = (I)/tau : 1 (unless refractory)

vt :1

I : 1

tau : second

'''

reset1 = '''

vt += v

v=0.0

'''

# The G2 neuron group are the neurons that generate the sparse vectors tht will be bound. To do this each neuron represents

# one slot of the sparse vector and the synaptic connections (S2) on the dendrite represent the time delay of the corresponding spike.

# The time delays are obtained from the P\_matrix (S2.delay). The input to this part of the neuromorphic circuit are the

# sequence of spikes from the 'P' spike generator group. A 'P' spike excites an axon which is connected to all the G2 neurons

# (S2.connect).

G2 = NeuronGroup(slots, equ1,

threshold='v >= 1.0', reset='v=v', method='euler',refractory = 't >=((Num\_bound)\*bits)\*ms')

G2.v = 0.0

G2.tau = 0.5\*ms

net1.add(G2)

S2 = Synapses(P, G2, 'w : 1', on\_pre='v += 1.25' )

range\_array1 = range(0,slots)

for n in range(0,mem\_size):

S2.connect(i=n,j=range\_array1)

S2.delay = np.reshape(P\_matrix,mem\_size\*slots)\*ms

net1.add(S2)

# To perform the cyclic shift and superposition operations the output from G2 is recurrently fed back such that the output from neuron\_0

# feeds to the input of neuron\_1 etc. Because Brian2 introduces a time delay of 0.1ms when performing this operation the delay for this

# feedback is the input\_delay minus 0.1ms (S3.delay)

S3 = Synapses(G2, G2, 'w : 1', on\_pre='v +=1.25' )

for n in range(0,slots):

S3.connect(i=n,j=(n+1)%(slots))

S3.delay = (input\_delay-0.1)\*ms

net1.add(S3)

# The resulting vector from the recurrent superposition operation is a dense vector.

# To create the corresponding sparse vector the G4 and G5 neuron groups work together to perform the Argmax operation.

# The G4 neurons perform part of this operation by using a variable spike threshold such that if spikes from the superposed vectors

# have the same time delay then they will only exceed the threshold if the same number of aligne spikes has not occured earlier.

# The G5 neurons then use a linear decaying neuron potential to create a single spike per slot which is the required sparse bound vector.

G4 = NeuronGroup(slots, model=equ1, reset=reset1, threshold='v>=vt',method='euler',refractory = 't <=((Num\_bound-1)\*bits)\*ms')

G4.v = 1.0

G4.vt = -0.5

G4.tau =0.25\*ms

net1.add(G4)

S4 = Synapses(G2, G4, 'w : 1', on\_pre='v +=1.0')

S4.connect(j='i')

net1.add(S4)

G5 = NeuronGroup(slots, equ2,

threshold='v <= 0.0', reset='v=1.0', method='euler',refractory = 't< (Num\_bound-1)\*bits\*ms or t> (Num\_bound)\*bits\*ms')

G5.v = 0.0

G5.I = -1.0

G5.tau = bits\*ms

net1.add(G5)

argmax\_synapse = Synapses(G4, G5, 'w : 1', on\_pre='v = 1.0')

argmax\_synapse.connect(j='i')

net1.add(argmax\_synapse)

# The following spike and state monitors are defined.

SMP = SpikeMonitor(P)

net1.add(SMP)

M2 = StateMonitor(G2, 'v', record=True)

net1.add(M2)

SM2= SpikeMonitor(G2)

net1.add(SM2)

SM4 = SpikeMonitor(G4)

net1.add(SM4)

M4 = StateMonitor(G4, 'vt', record=True)

net1.add(M4)

M5 = StateMonitor(G4, 'v', record=True)

net1.add(M5)

SM5 = SpikeMonitor(G5)

net1.add(SM5)

M6 = StateMonitor(G5, 'v', record=True)

net1.add(M6)

# Network 1 is now run for delta milliseconds.

net1.run(delta\*ms)

# Obtain the sparse vector timings from the SM5 monitor and print the timings so that they can be compared with the theoretical values.

array2 = np.array([SM5.i,SM5.t/ms])

sub\_array2 = array2[0:2,slots:]

print()

sorted\_sub\_array2 = sub\_array2[:,sub\_array2[0].argsort()].astype(int) - Num\_bound\*bits

P1\_timing = sorted\_sub\_array2[1]

print(P1\_timing)

print()

# The following plots output from the different monitors

subplot(4,2,1)

plot(SMP.t/ms, SMP.i,'|')

xlabel('Time (ms)')

ylabel('P Neuron id')

#plt.ylim(0,10)

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.xlim(9700,10800)

subplot(4,2,2)

plot(SM2.t/ms, SM2.i,'|')

xlabel('Time (ms)')

ylabel('G2 Neuron id')

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.ylim(y\_low,y\_high)

#plt.xlim(690,810)

subplot(4,2,3)

plot(M4.t/ms, M4.vt.T)

xlabel('Time (ms)')

ylabel('G4 Threshold Voltage')

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.xlim(690,810)

subplot(4,2,4)

plot(M5.t/ms, M5.v.T)

xlabel('Time (ms)')

ylabel('G4 Neuron Voltage')

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.xlim(690,810)

subplot(4,2,5)

plot(SM4.t/ms, SM4.i,'|')

xlabel('Time (ms)')

ylabel('G4 Neuron id')

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.ylim(y\_low,y\_high)

#plt.xlim(690,810)

#plt.ylim(29,31)

subplot(4,2,6)

plot(M6.t/ms, M6.v.T)

xlabel('Time (ms)')

ylabel('G5 Neuron Voltage')

#plt.xlim(0,22000)

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.xlim(690,900)

subplot(4,2,7)

plot(SM5.t/ms, SM5.i,'|')

xlabel('Time (ms)')

ylabel('G5 Neuron id')

#plt.xlim(0,22000)

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.ylim(y\_low,y\_high)

#plt.xlim(790,900)

show()

#--------------------------------------------------------------------------------------------------------

#This section of the code implements the Brian2 neuromorphic circuit which unbinds the vector and then compares the

#unbound vector with each vector in the memory to find the best match (i.e. the clean-up memory operation)

# We first generate the time delay data\_matrix which will be used in the 'clean-up memory' so that the input vector

# time delay in each slot plus the delay matrix line up at the number of bits per slot

# (e.g. a time delay in slot 0 of the input vector of say 10 will have a corresponding delay of 90 in the corresponding

# data\_matrix so that if this vector is received then the match condition is an input potential to the neuron at 100)

data\_matrix = bits - P\_matrix

net2=Network()

# To pass the sparse vector from Net1 into Net2 we create a SpikeGeneratorGroup that uses the P1\_timing from Net1 to generate

# the sparse bound vector which is the input to NeuronGroup G6 (S6).

P1 = SpikeGeneratorGroup(slots,np.arange(slots), P1\_timing\*ms)

net2.add(P1)

equ3 ='''

dv/dt = -v / tau : 1

tau : second

ts:second

'''

# The G6 NeuronGroup performs the recurrent unbinding of the bound sparse vector in this case Neuron\_1 connects to Neuron\_0 etc. (S7.connect)

# Again the recurrent delay depends on the number of bit positions.

G6 = NeuronGroup(slots, equ3, threshold='v >= 0.5', reset='v=0.0', method='euler',refractory = 't>= (Num\_bound-1)\*bits\*ms')

G6.v = 0.0

G6.tau = 1\*ms

net2.add(G6)

S6 = Synapses(P1, G6, 'w : 1', on\_pre='v = 1.0')

S6.connect(j='i')

net2.add(S6)

# Recursively unbind with a delay of 'input\_delay' between unbound vectors

S7 = Synapses(G6, G6, 'w : 1', on\_pre='v = 1.0')

for n in range(0,slots):

S7.connect(i=n,j=(n-1)%slots)

S7.delay = (input\_delay)\*ms

net2.add(S7)

# The unbound vector is fed directly into NeuronGroup G7 which performs the 'Clean-Up' memory operation by comparing it in parallel

# with all of the vectors in the memory. This operation relies on the alignemnt of the unbound vector spikes delayed by the transpose

# of the data\_matrix (S8.delay). N.B. we have used the predicted min\_match to set the threshold for the clean-up memory and just outout

# the index of the best matching vector. Also note that the output order is the reverse of the input order.

G7 = NeuronGroup(mem\_size, equ3, threshold='v > min\_match-2.0', reset='v=0.0', method='euler')

G7.v = 1.0

G7.tau = 1.0\*ms

net2.add(G7)

range\_array1 = range(0,mem\_size)

S8 = Synapses(G6, G7, on\_pre='v += 1.0')

for n in range(0,slots):

S8.connect(i=n,j=range\_array1)

data\_matrix2 = np.transpose(data\_matrix)

S8.delay = np.reshape(data\_matrix2,mem\_size\*slots)\*ms

net2.add(S8)

# Create the required monitors

SMP1 = SpikeMonitor(P1)

net2.add(SMP1)

SM6 = SpikeMonitor(G6)

net2.add(SM6)

SM7 = SpikeMonitor(G7)

net2.add(SM7)

M7 = StateMonitor(G7, 'v', record=True)

net2.add(M7)

#Run Network2 for delta milliseconds

net2.run(delta\*ms)

# Plot the sparse bound vector

plot(SMP1.t/ms, SMP1.i,'|')

xlabel('Time (ms)')

ylabel('P Neuron id')

show()

# Plot the other monitors

subplot(3,1,1)

plot(SM6.t/ms, SM6.i,'|')

xlabel('Time (ms)')

ylabel('G6 Neuron id')

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.xlim(bits\*Num\_bound-100,2\*bits\*(Num\_bound+1))

#plt.ylim(y\_low,y\_high)

subplot(3,1,2)

plot(M7.t/ms, M7.v.T)

xlabel('Time (ms)')

ylabel('G7 Neuron Voltage')

#plt.xlim(0,2\*bits\*Num\_bound)

#plt.xlim(bits\*Num\_bound-100,2\*bits\*(Num\_bound+1))

subplot(3,1,3)

plot(SM7.t/ms, SM7.i,'|')

xlabel('Time (ms)')

ylabel('G7 Neuron id')

show()