```
1  #! /usr/bin/perl
2
3  # Author: Swapnil Bhatia (swapnilb at bu.edu)
4  # Thu Jan 19 15:06:54 EST 2017
5  # Computes angular difference between measured and signal vectors of
6  # BLADE circuits. Computes dynamic range, angles between all boolean
7  # functions, and measured circuit to all functions. Exports to CSV and
8  # JSON. Needs input as CSV in a specific format described in the
9  # comments.
10 #
11 # Send questions to swapnilb, bweinbe, or wilwong at bu dot edu.
12
13 # Tested on:
14 # perl 5, version 12, subversion 4 (v5.12.4) built for i386−linux−thread−multi
15 # Linux synbiotools.bu.edu 4.8.10−300.fc25.x86_64 #1 SMP Mon
16 # Nov 21 18:59:16 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
17
18 use Data::Dumper;
19 use Math::Trig;
20 use Storable;
21 use JSON;
22 use POSIX;
23
24 $Data::Dumper::Sortkeys = 1;
25 $Data::Dumper::Pair = ':';
26 $Data::Dumper::Quotekeys = 1;
27
28 my @bitVector;
29
30 $n = 2;  # n is the number of inputs to the circuits
31 $len = 2 * 2**$n;  # we have two outputs, so we have 2 x 2^n independent bits
32 #$circuitToPlot = $ARGV[0];
33
34 if ($#ARGV < 0) {
35     print STDERR "usage: computeAllAngulars <csv formatted data file>\n";
36     exit;
37 }
38
39 # This hash holds all the boolean functions on n input bits with two outputs.
40 my %allTT = ();
41
42 # Initialize the bit vector used to represent the boolean function to
43 # 0.
44 for($i=0; $i<$len; $i++) {
45    push(@bitVector, 0);
46 }
```

```
47
48  # Each boolean function gets a unique circuit identifier. This is used
49  # as a part of the hash key in allTT.
50  my $circuitId = 0;
51
52  # We first generate all possible boolean functions and initialize all
53  # the values that we can at this point. We do this by taking a vec 0
54  # bitvector and incrementing it, until we have covered all possible
55  # bit strings.
56  do {
57
58      # We use the following vars to also save the two outputs, i.e. two
59      # functions separately, for convenience and analysis later, in
60      # addition to saving the entire function, i.e. both outputs,
61      # together.
62      $evenTT = 0;
63      $oddTT = 0;
64
65      # Here, len is the size of the boolean function, that is, 2 x 2^n.
66      for($i=0; $i<$len; $i++) {
67
68          # Each function is initially called "Unbuilt-k", where k is
69          # some integer identifier. Once we read data that shows that
70          # function was tested in the laboratory, we move it into a new
71          # hash and fill in the data gathered in the laborator.
72
73          $allTT{"Unbuilt-$circuitId"}{truthTable}{"$i"}{truthTableOutput}
74              = $bitVector[$len-1-$i];
75
76          # Initialize other fields to reasonable values.
77          $allTT{"Unbuilt-$circuitId"}{truthTable}{"$i"}{mean} = "1.0";
78
79          # Add up the 1-bits to compute vector magnitude later.
80          $allTT{"Unbuilt-$circuitId"}{truthTableVectorMag} += $bitVector[$len-1-$i];
81
82          # Compute the integer value of each function, represented as a
83          # bit string. We use this later to have a concise way to
84          # describe a boolean function, just by this integer.
85          if ($i % 2 == 0) {
86              $evenTT += 2**(2**$n-1-$i/2) * $bitVector[$i];
87          } else {
88              $oddTT += 2**(2**$n-1-($i-1)/2) * $bitVector[$i];
89
90          }
91      }
92
```

```
93        # Initially, all boolean functions are "Unbuilt."
94        $allTT{"Unbuilt-$circuitId"}{built} = "No";
95
96        # We compute integer descriptors for each function.
97        $allTT{"Unbuilt-$circuitId"}{hexTruthTable} = sprintf("%X", $circuitId);
98        $allTT{"Unbuilt-$circuitId"}{hexGfpTruthTable} = sprintf("%X", $evenTT);
99        $allTT{"Unbuilt-$circuitId"}{hexMchTruthTable} = sprintf("%X", $oddTT);
100
101       # We initialize vector magnitudes for the binary function itself.
102       $allTT{"Unbuilt-$circuitId"}{meanVectorMag} = sqrt(8.0);
103       $allTT{"Unbuilt-$circuitId"}{truthTableVectorMag}
104           = sqrt($allTT{"Unbuilt-$circuitId"}{truthTableVectorMag});
105
106       # Move on to the next boolean function, and give it a new
107       # identifier.
108       $circuitId++;
109
110   # Continue, until there are no more bitstrings left to process.
111   } while (getNextTT() == 1);
112
113   # We explcitly ignore the trivial function Contradiction.
114   delete $allTT{"Unbuilt-0"};
115
116   # Count the total number of functions enumerated.
117   $count = scalar keys %allTT;
118
119   # Silly check point.
120   #print STDERR "Total circuits = $count\n";
121
122   # Print entire data structure with good formatting, thanks to Dumper::.
123   #print Dumper(\%allTT);
124
125   # Proceed after all of this looks correct.
126   #exit;
127
128   ## Generation of boolean functions is complete. We now move on to
129   #reading in laboratory data on some of these boolean functions.
130
131   ############## Reading in lab data ##################
132
133   $inFile = $ARGV[0];
134
135   open(IN, "<$inFile") or die;
136
137   # IMPORTANT: We assume that the first line of the file has clear
138   # column headings for human readability. We skip that line.
```

```
139 readline(IN);
140
141 # We initialize an empty hash table. This will hold the boolean
142 # functions which have been built in the lab and tested. We call these
143 # "circuits" -- implementations of abstract boolean functions.
144 my %circuit = ();
145
146 # This somewhat long loop reads in the data into this new hash table.
147 while(!eof(IN)) {
148
149     # Each line of the file contains data about a SINGLE circuit.
150     # Thus, data on each circuit is newline delimited. We follow *nix
151     # conventions for newline and convert the file (not shown here,
152     # and done externally) to this convention.
153     $line = readline(IN);
154
155     # The fields in each line of circuit data are delimited by commas.
156     # We split the line we just read into fields.
157     @fields = split(/,/, $line);
158
159     # We know by predefined convention that the first field (index 0) is a
160     # circuit identifier.
161     $circuitId = $fields[0];
162
163     # We know by predefined convention that the next four columns
164     # describe a BLADE device's composition. We save it with dot
165     # notation and use it as a human readable descriptor later.
166     $circuitStructure = "";
167     for($i=1; $i<5; $i++) {
168         $circuitStructure = $circuitStructure . "." . $fields[$i];
169     }
170
171     # Remove prefix dot and save into this device's entry in our hash
172     # table.
173     $circuitStructure =~ s/^.//;
174     $circuit{"$circuitId"}{structure} = $circuitStructure;
175
176     # We mark this device as having been built and tested in the
177     # laboratory.
178     $circuit{"$circuitId"}{built} = "Yes";
179
180     # We know from predefined convention that the next eight columns
181     # contain the intended boolean function. We read this in as a
182     # single boolean function, as well as separate it out into the two
183     # outputs below.
184
```

```
185     $startField = 5;
186     $endField = 13;
187
188     my @circuitOutputs;
189     $mag = 0;
190     $circuitBinToDec = 0;
191     for($i=$startField; $i<$endField; $i++) {
192         $row = $i-$startField;
193         $circuit{"$circuitId"}{truthTable}{"$row"}{truthTableOutput} = $fields[$i];
194
195         # In addition to reading in the boolean function, we also
196         # compute the vector magnitude of the function as a bit
197         # vector, and it's integer value, as a succinct descriptor,
198         # just as we had one in the list of unbuilt functions hash
199         # table above.
200         $mag += ($fields[$i] * $fields[$i]);
201         $circuitBinToDec += ((2**$row) * $fields[$i]);
202     }
203
204     $circuit{"$circuitId"}{truthTableVectorMag} = sqrt($mag);
205
206     # We will use the descriptor also for display to users as a hex
207     # value, to enable quick comprehension of the outputs for any
208     # combination of inputs.
209     $circuit{"$circuitId"}{hexTruthTable} = sprintf("%X", $circuitBinToDec);
210
211     # Finally, we delete this circuit from the list of unbuilt
212     # circuits.
213     delete $allTT{"Unbuilt-$circuitBinToDec"};
214
215     # We also separate the boolean function by output into two
216     # separate but related functions: one reported by GFP and another
217     # by mCherry, as we know from out-of-band agreement.
218
219     # We first separate out the GFP output. We compute vector
220     # magnitude and the number of "true" values to help determine
221     # later if our dot product metric would be computable for this
222     # function. We also compute and save a succinct integer
223     # descriptor.
224     $magGFP = 0;
225     $cnt1sGFP = 0;
226     $circuitBinToDec = 0;
227     for($i=$startField; $i<$endField; $i+=2) {  # skip mCherry values
228         $row = ($i-$startField)/2;
229         $circuit{"$circuitId"}{truthTableGfp}{"$row"}{truthTableOutput} = $fields[$i];
230         $magGFP += ($fields[$i] * $fields[$i]);
```

```
231             if ($fields[$i] =~ /1/) {
232                 $cnt1sGFP++;
233             }
234             $circuitBinToDec += ((2**$row) * $fields[$i]);
235         }
236         $circuit{"$circuitId"}{truthTableVectorMagGfp} = sqrt($magGFP);
237         $circuit{"$circuitId"}{truthTableTotalOnesGfp} = $cnt1sGFP;
238         $circuit{"$circuitId"}{hexGfpTruthTable} = sprintf("%X", $circuitBinToDec);
239
240         # We work similarly on the mCherry output.
241         $magMCh = 0;
242         $cnt1sMCh = 0;
243         $circuitBinToDec = 0;
244         for($i=$startField+1; $i<$endField; $i+=2) {  # skip GFP values
245             $row = ($i-($startField + 1))/2;  #  truth table row of just mcherry output
246             $circuit{"$circuitId"}{truthTableMch}{"$row"}{truthTableOutput} = $fields[$i];
247             $magMCh += ($fields[$i] * $fields[$i]);
248             if ($fields[$i] =~ /1/) {
249                 $cnt1sMCh++;
250             }
251             $circuitBinToDec += ((2**$row) * $fields[$i]);
252         }
253         $circuit{"$circuitId"}{truthTableVectorMagMch} = sqrt($magMCh);
254         $circuit{"$circuitId"}{truthTableTotalOnesMch} = $cnt1sMCh;
255         $circuit{"$circuitId"}{hexMchTruthTable} = sprintf("%X", $circuitBinToDec);
256
257
258         # By predefined convention, we know that the next eight columns
259         # contain the output values measured for this device in the
260         # laboratory. We record these together and separately for each
261         # output. We also record them as capped and uncapped, as described
262         # below. And we read or compute basic summaries like mean and vector
263         # magnitude, treating the tuple of values as a real vector.
264
265         $startField = 13;
266         $endField = 21;
267
268         # We "cap" any measured value above 20,000 a.u. to 20,000 a.u., as
269         # at this level any such signal is considered sufficiently robust.
270         # Please see manuscript for a discussion of this capping or
271         # contact Ben Weinberg for specific questions.
272         my $CAP_VALUE = 20000.0;
273
274         # Note that we also store the uncapped values.
275
276         my @circuitMean;
```

```
277         $mag = 0;
278         for($i=$startField; $i<$endField; $i++) {
279             $row = $i-$startField;
280             $circuit{"$circuitId"}{truthTable}{"$row"}{uncappedMean} = $fields[$i];
281             $tmp = $fields[$i];
282             if ($tmp > $CAP_VALUE) {
283                 $tmp = $CAP_VALUE;
284             }
285             $circuit{"$circuitId"}{truthTable}{"$row"}{mean} = $tmp;
286             $mag += ($tmp * $tmp);
287         }
288         $circuit{"$circuitId"}{meanVectorMag} = sqrt($mag);
289
290
291         # We repeat the process for the two outputs separately.
292         $magGFP = 0;
293         for($i=$startField; $i<$endField; $i+=2) {
294             $row = ($i-$startField)/2;
295             $circuit{"$circuitId"}{truthTableGfp}{"$row"}{uncappedMean} = $fields[$i];
296             $tmp = $fields[$i];
297             if ($tmp > $CAP_VALUE) {
298                 $tmp = $CAP_VALUE;
299             }
300             $circuit{"$circuitId"}{truthTableGfp}{"$row"}{mean} = $tmp;
301             $magGFP += ($tmp * $tmp);
302         }
303         $circuit{"$circuitId"}{meanVectorMagGfp} = sqrt($magGFP);
304
305         $magMCh = 0;
306         for($i=$startField+1; $i<$endField; $i+=2) {
307             $row = ($i-($startField + 1))/2;
308             $circuit{"$circuitId"}{truthTableMch}{"$row"}{uncappedMean} = $fields[$i];
309             $tmp = $fields[$i];
310             if ($tmp > $CAP_VALUE) {
311                 $tmp = $CAP_VALUE;
312             }
313             $circuit{"$circuitId"}{truthTableMch}{"$row"}{mean} = $tmp;
314             $magMCh += ($tmp * $tmp);
315         }
316         $circuit{"$circuitId"}{meanVectorMagMch} = sqrt($magMCh);
317
318
319
320
321         # Finally, we read the last eight columns which contain the
322         # standard error of mean for each measured output value. We save
```

```
323         # them together and separately for each output. This is currently
324         # unused for any computation.
325
326         $startField = 21;
327         $endField = 29;
328         for($i=$startField; $i<$endField; $i++) {
329             $row = $i-$startField;
330             $circuit{"$circuitId"}{truthTable}{"$row"}{standardErrorOfUncappedMean}
331                 = $fields[$i];
332         }
333
334         ## GFP standard error of means separately recorded.
335         for($i=$startField; $i<$endField; $i+=2) {
336             $row = ($i-$startField)/2;
337             $circuit{"$circuitId"}{truthTableGfp}{"$row"}{standardErrorOfUncappedMean}
338                 = $fields[$i];
339         }
340
341         ## MCh standard error of means separately recorded.
342         for($i=$startField+1; $i<$endField; $i+=2) {
343             $row = ($i-($startField+1))/2;
344             $circuit{"$circuitId"}{truthTableMch}{"$row"}{standardErrorOfUncappedMean}
345                 = $fields[$i];
346         }
347
348     }
349
350     # This marks the end of the loop that reads in data from the file.
351
352     # We may pause and use this as a check point to confirm the number of
353     # built and unbuilt circuits against their true values.
354
355     $count = keys %allTT;
356     #print "Total Unbuilt circuits = $count\n";
357     #print Dumper(\%allTT);
358     $count = keys %circuit;
359     #print "Total Built circuits = $count\n";
360
361
362     # We now move all the *unbuilt* boolean functions into the circuits
363     # hash table. Thus, our circuits hash table now comprises all boolean
364     # functions, with the unbuilt ones marked with a "No" under the
365     # "built" field.
366
367     @circuit{keys %allTT} = values %allTT;
368     $count = keys %circuit;
```

```
369 #print "Total Built circuits = $count\n";
370
371 # Here, we may pause and check that all the data read in appears to be
372 # correctly recorded.
373
374 ##print Dumper(\%circuit);
375 ##exit;
376
377
378 # We now proceed to compute the dot product metric.
379
380 # We also wish to compute the number of circuits whose closest boolean
381 # function is NOT the intended function. We loosely refer to them as
382 # "dysfunctional." We keep a count of such circuits.
383
384 $dysfunc = 0;
385
386 # This is the main loop that computes the dot product metric. We
387 # proceed in a sorted order by circuit identifier.
388
389 for my $circuitId1 (sort {$a <=> $b} keys %circuit) {
390
391     # We only consider functions that have been tested in the
392     # laboratory, skipping any function that has not been built.
393     if ($circuit{"$circuitId1"}{built} =~ /No/) {
394         #print "Skipping circuit $circuitId1\n";
395         next;
396     }
397
398     # We may print each circuit being analyzed to confirm that all
399     # built, and only those circuits, are being included in the
400     # analysis.
401     # print "Analyzing circuit $circuitId1: $circuit{\"$circuitId1\"}{hexTruthTable}\n";
402
403
404     # Below, we use the dot product to obtain the angle of the
405     # measured output to the intended boolean function.
406
407     # We begin with the full function taking both outputs together. We
408     # will compute this using the mean values of the measured outputs.
409     # Hence we refer to this as the mean dot product.
410     $ttMeanDotProduct = 0.0;
411
412     # We get a reference to the full function, from our data
413     # structure.
414     my $c1 = $circuit{"$circuitId1"}{truthTable};
```

```
415
416         # Then, for each input−output relation for this function, we
417         # compute the dot product between the signal vector, i.e., the
418         # measured output, and the function vector, i.e., the boolean
419         # function intended to be implemented.
420         for my $row (keys %{$c1}) {
421             $ttMeanDotProduct
422                 += ($circuit{"$circuitId1"}{truthTable}{"$row"}{mean}
423                     * $circuit{"$circuitId1"}{truthTable}{"$row"}{truthTableOutput});
424         }
425
426         # The angle between vectors is not computable if the magnitude of
427         # one of the vectors is zero. We check this, and if so, compute
428         # the angle. Else, we label it as "Undefined."
429         if ($circuit{"$circuitId1"}{meanVectorMag} > 0 &&  $circuit{"$circuitId1"}{truthTableVectorMag} > 0) {
430             $cosTheta
431                 = $ttMeanDotProduct /
432                     ($circuit{"$circuitId1"}{meanVectorMag} * $circuit{"$circuitId1"}{truthTableVectorMag});
433
434             # We save this value, calling it "self theta" to indicate the
435             # angle between measured and intended vectors, to contrast
436             # angles with all other functions.
437             $selfTheta = rad2deg(acos($cosTheta));
438             $circuit{"$circuitId1"}{theta} = $selfTheta;
439         } else {
440             $theta = "Undefined";
441         }
442
443         # This completes the computation of the angle between intended and
444         # measured vectors. We now move to computing the angle between
445         # measured and all possible function vectors. We also determine
446         # the function with the minimum angle, and the rank of the
447         # intended function, when all functions are sorted in ascending
448         # order of their angle with the measured vector. We use this as a
449         # proxy metric of whether or not a circuit satisfies the desired
450         # behavior.
451
452         my $min = 1000.0;
453         my $minCircuit = "";
454         my $rank = 0;
455
456         # This inner loop computes the angle between all possible boolean
457         # functions and the measured vector.
458         for my $circuitId2 (keys %circuit) {
459
460             # We first compute this, similarly as above, with the full
```

```
461             # function.
462             $ttMeanDotProduct = 0.0;
463             my $c1 = $circuit{"$circuitId1"}{truthTable};
464             for my $row (keys %{$c1}) {
465                 $ttMeanDotProduct
466                     += ($circuit{"$circuitId1"}{truthTable}{"$row"}{mean}
467                         * $circuit{"$circuitId2"}{truthTable}{"$row"}{truthTableOutput});
468             }
469
470             # Only if the magnitudes are positive, can we compute the
471             # angle. Else, we mark it as "Undefined."
472             if (
473                     $circuit{"$circuitId1"}{meanVectorMag} > 0 &&
474                     $circuit{"$circuitId2"}{truthTableVectorMag} > 0
475                ) {
476                 $cosTheta
477                     = $ttMeanDotProduct
478                         / ($circuit{"$circuitId1"}{meanVectorMag} * $circuit{"$circuitId2"}{truthTableVectorMag});
479                 $theta = rad2deg(acos($cosTheta));
480             } else {
481                 $theta = "Undefined";
482             }
483
484
485             # For each circuit, we save the angle it makes with all possible
486             # boolean functions, whenever it is defined.
487             $circuit{$circuitId1}{distanceTo}{$circuitId2}{theta} = sprintf("%2.2f", $theta);
488
489             # For convenience, we save explicitly, the identifier of the
490             # function making the smallest angle with the current circuit.
491             if ($theta !~ /Undefined/) {
492                 if ($min > $theta) {
493                     $minCircuit = $circuitId2;
494                     #print "$minCircuit is closer: $min > $theta\n"; ### DEBUG
495                     $min = $theta;
496                 } else {
497                 }
498
499                 # We also count the number of functions that make a
500                 # smaller angle with this circuit than it's intended
501                 # function makes with its measured output. In other words,
502                 # this gives us the rank of the intended function.
503                 # Whenever this rank is not 0, we consider the circuit as
504                 # having "failed" or being "dysfunctional." This is a
505                 # rather strict criterion, as the intended function may
506                 # lose out to some other function even by a fraction of an
```

```
507             # angle for the circuit to be considered dysfunctional.
508             if ($theta < $selfTheta) {
509                 $rank++;
510             }
511         }
512
513
514
515         # Next, we perform a similar computation of angles, but this
516         # time between intended boolean function, and all other
517         # boolean functions. Note, that this does not use the measured
518         # values and is intended to provide a baseline angular
519         # difference between this and other functions. Note that this
520         # value is a fundamental value arising from boolean logic and
521         # algebra.
522
523         # We perform this computation for all eight output values.
524         $ttTTDotProduct = 0.0;
525         my $c1 = $circuit{"$circuitId1"}{truthTable};
526         for my $row (keys %{$c1}) {
527             $ttTTDotProduct
528                 += (
529                     $circuit{"$circuitId1"}{truthTable}{"$row"}{truthTableOutput}
530                         * $circuit{"$circuitId2"}{truthTable}{"$row"}{truthTableOutput}
531                    );
532         }
533         if (
534             $circuit{"$circuitId1"}{truthTableVectorMag} > 0 &&
535             $circuit{"$circuitId2"}{truthTableVectorMag} > 0
536            ) {
537             $TTCosTheta
538                 = sprintf(
539                     "%2.5f",
540                     $ttTTDotProduct /
541                     ($circuit{"$circuitId1"}{truthTableVectorMag} * $circuit{"$circuitId2"}{truthTableVectorMag})
542                  );
543             $TTTheta = rad2deg(acos($TTCosTheta));
544         } else {
545             $TTTheta = "Undefined";
546         }
547
548
549         # Again, we save this value for each boolean function, against
550         # all other boolean functions.
551         $circuit{$circuitId1}{distanceTo}{$circuitId2}{truthTableTheta} = sprintf("%2.2f", $TTTheta);
552
```

```
553
554         }
555
556         # We mark circuits whose intended functions do not have the
557         # minimum angle as dysfunctional.
558         if (!($circuit{$minCircuit}{hexTruthTable} eq $circuit{"$circuitId1"}{hexTruthTable})) {
559             $dysfunc++;
560             $circuit{"$circuitId1"}{dysfunctional} = "Yes";
561         } else {
562             $circuit{"$circuitId1"}{dysfunctional} = "No";
563         }
564
565         # We also save the succinct descriptor, angle, and identifier of the boolean function
566         # that has the smallest angle with the intended function.
567         $circuit{"$circuitId1"}{closestTruthTable} = $circuit{$minCircuit}{hexTruthTable};
568         $circuit{"$circuitId1"}{thetaClosest} = $min;
569         $circuit{"$circuitId1"}{closestCircuitId} = $minCircuit;
570         $circuit{"$circuitId1"}{intendedTruthTableRank} = $rank;
571
572     }
573
574     # We also compute a measure of circuit quality similar to one defined
575     # in the literature (see Nielsen et al. Science 2016). Known as "dynamic
576     # range," we define this as the difference between the minimum
577     # measured output where the intended boolean output is 1, and the
578     # maximum measured outwhere where the intended boolean function is 0.
579     # Thus, it's a "worst case" difference in the range of measured output
580     # values.
581     computeDynamicRange();
582
583     $count = keys %circuit;
584
585     # We can persist the data collated and computed into files. We save
586     # fields as needed in comma-separated form to file.
587     exportAsCSV();
588
589     # We can compute and save histograms to summarize circuits.
590     saveAngleHistogram();
591
592
593     # We save the entire hash table, as is, to file.
594     # To minimize useless records, we remove unbuilt circuits.
595
596     for my $key (keys %circuit) {
597         if ($circuit{$key}{built} =~ /No/) {
598             delete $circuit{$key};
```

```
599     }
600 }
601
602 # We configure the JSON to print with readable formatting.
603 my $json = JSON->new;
604 $json = $json->pretty(1);
605 $json = $json->canonical(1);
606 my $exportJSON = $json->encode(\%circuit);
607
608 open(JS, ">JSON-$inFile.json") or die;
609 print JS $exportJSON;
610 close(JS);
611
612
613
614 # Done.
615 exit;
616
617 # Computes a histogram by angular distance between intended and
618 # measured outputs, across all circuits.
619 sub saveAngleHistogram {
620
621     my %histogram = ();
622     my $total = 0;
623
624     for my $key (keys %circuit) {
625         if ($circuit{$key}{built} =~ /Yes/) {
626             my $kk = ceil($circuit{$key}{theta});
627             if (exists $histogram{$kk}) {
628                 $histogram{$kk}{count}++;
629             } else {
630                 $histogram{$kk}{count} = 1;
631                 $histogram{$kk}{fraction} = 0.0;
632             }
633             $total++;
634         }
635     }
636
637     my $cumu = 0.0;
638     open(HIS, ">HIST-$inFile.dat") or die;
639     for my $key (sort {$a <=> $b} keys %histogram) {
640         my $frac = $histogram{$key}{count} / $total;
641         $cumu += $frac;
642         print HIS "$key $histogram{$key}{count} $cumu\n";
643     }
644     close(HIS);
```

```
645
646 }
647
648 # Computes a histogram of ranks across all circuits.
649 sub saveRankHistogram {
650
651     my %histogram = ();
652     my $total = 0;
653
654     for my $key (keys %circuit) {
655         if ($circuit{$key}{built} =~ /Yes/) {
656             my $kk = $circuit{$key}{intendedTruthTableRank};
657             if (exists $histogram{$kk}) {
658                 $histogram{$kk}{count}++;
659             } else {
660                 $histogram{$kk} = 1;
661                 $histogram{$kk}{fraction} = 0.0;
662             }
663             $total++;
664         }
665     }
666
667     my $cumu = 0.0;
668     open(HIS, ">HIST-$inFile.dat") or die;
669     for my $key (sort {$a <=> $b} keys %histogram) {
670         my $frac = $histogram{$key}{count} / $total;
671         $cumu += $frac;
672         print HIS "$key $histogram{$key}{count} $cumu\n";
673     }
674     close(HIS);
675
676 }
677
678 # Persists compiled data in comma-separated form to a file.
679 sub exportAsCSV {
680
681 my $circuitId1;
682
683     open(OUT, ">OUTPUT-from-$inFile") or die;
684
685
686     # We may want to print out column headings for each column.
687     #print OUT "CircuitId, Structure, HexGFP.MChTruthTable, GFPDynamicRange, MChDynamicRange,
688     #GFPCappedDynamicRange, MChCappedDynamicRange, Theta, IntendedTTRank,
689     #HexClosestGFP.MChTruthTable, ClosestTheta, Dysfunctional\n";
690
```

```
691     for my $circuitId1 (sort {$a <=> $b} keys %circuit) {
692
693         if ($circuit{$circuitId1}{built} =~ /Yes/) {
694
695             $structure = $circuit{$circuitId1}{structure};
696             $hexTT = $circuit{$circuitId1}{hexGfpTruthTable} . $circuit{$circuitId1}{hexMchTruthTable};
697             $GFPdr = $circuit{$circuitId1}{gfpDynamicRange};
698             $MChdr = $circuit{$circuitId1}{mchDynamicRange};
699             $GFPCappedDr = $circuit{$circuitId1}{gfpCappedDynamicRange};
700             $MChCappedDr = $circuit{$circuitId1}{mchCappedDynamicRange};
701             $theta = sprintf("%2.2f", $circuit{$circuitId1}{theta});
702             $intendedRank = $circuit{$circuitId1}{intendedTruthTableRank};
703             $closestTT
704                 = "0x"
705                   . $circuit{$circuit{$circuitId1}{closestCircuitId}}{hexGfpTruthTable}
706                   . $circuit{$circuit{$circuitId1}{closestCircuitId}}{hexMchTruthTable};
707             $closestTheta = sprintf("%2.2f", $circuit{$circuitId1}{thetaClosest});
708             $dysfunc = $circuit{$circuitId1}{dysfunctional};
709             $closestCircuitId = $circuit{$circuitId1}{closestCircuitId};
710
711             print OUT "$circuitId1 $structure $hexTT $GFPdr $MChdr $GFPCappedDr ";
712             print OUT "$MChCappedDr $theta $intendedRank $closestTT $closestTheta $dysfunc\n";
713         }
714
715     }
716     close(OUT);
717
718 }
719
720 # Computes worst case dynamic range of measured outputs for all
721 # circuits.
722 sub computeDynamicRange {
723
724 my $circuitId1;
725
726     # We compute dynamic range for all circuits. The dynamic range is
727     # the difference between:
728     #   highest measured output when the function output is defined
729     #     to be 0, and
730     #   the lowest measured output when the function output is defined
731     #     to be 1.
732     # NOTE that this may not be defined for certain functions where
733     # all outputs are 0, or 1.
734     # Since we perform this computation for all outputs together and
735     # by separating the two outputs, the dynamic range may not be
736     # defined where either of the outputs is either always 1 or 0.
```

```
737         # Also NOTE that we compute this range for capped and uncapped
738         # mean values.
739         for my $circuitId1 (keys %circuit) {
740
741             ## Is dynamic range defined for GFP?
742             if (
743                 $circuit{"$circuitId1"}{truthTableTotalOnesGfp} > 0 &&
744                 $circuit{"$circuitId1"}{truthTableTotalOnesGfp} < 4
745             ) {
746
747                 my $minOn = 1e6;
748                 my $maxOff = -1;
749                 my $cappedMinOn = 1e6;
750                 my $cappedMaxOff = -1;
751
752                 my $c1 = $circuit{"$circuitId1"}{truthTableGfp};
753                 for my $row (keys %{$c1}) {
754
755                     # We first deal with uncapped GFP output, and find
756                     # cases where the function output is 1, and the GFP
757                     # output is the lowest.
758                     if (
759                         $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{truthTableOutput} == 1 &&
760                         $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{uncappedMean} < $minOn
761                     ) {
762                         $minOn = $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{uncappedMean};
763                     }
764
765                     # Next, we find cases where the function output is 0, and the GFP
766                     # output is the highest.
767                     if (
768                         $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{truthTableOutput} == 0 &&
769                         $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{uncappedMean} > $maxOff
770                     ) {
771                         $maxOff = $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{uncappedMean};
772                     }
773
774
775                     ## We do the same for capped values of GFP.
776                     if (
777                         $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{truthTableOutput} == 1 &&
778                         $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{mean} < $cappedMinOn
779                     ) {
780                         $cappedMinOn = $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{mean};
781                     }
782
```

```
783                 if (
784                     $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{truthTableOutput} == 0 &&
785                     $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{mean} > $cappedMaxOff
786                 ) {
787                     $cappedMaxOff = $circuit{"$circuitId1"}{truthTableGfp}{"$row"}{mean};
788                 }
789
790             }
791
792             $drGFP = $minOn - $maxOff;
793             $cappedDrGFP = $cappedMinOn - $cappedMaxOff;
794
795             $circuit{"$circuitId1"}{gfpDynamicRange} = $drGFP;
796             $circuit{"$circuitId1"}{gfpCappedDynamicRange} = $cappedDrGFP;
797
798         } else {
799             $circuit{"$circuitId1"}{gfpDynamicRange} = "Undefined";
800             $circuit{"$circuitId1"}{gfpCappedDynamicRange} = "Undefined";
801         }
802
803
804
805
806
807         # We repeat the above for mCherry.
808         #
809         if (
810             $circuit{"$circuitId1"}{truthTableTotalOnesMch} > 0 &&
811             $circuit{"$circuitId1"}{truthTableTotalOnesMch} < 4
812         ) {
813
814             my $minOn = 1e6;
815             my $maxOff = -1;
816             my $cappedMinOn = 1e6;
817             my $cappedMaxOff = -1;
818
819             my $c1 = $circuit{"$circuitId1"}{truthTableMch};
820             for my $row (keys %{$c1}) {
821
822                 if (
823                     $circuit{"$circuitId1"}{truthTableMch}{"$row"}{truthTableOutput} == 1 &&
824                     $circuit{"$circuitId1"}{truthTableMch}{"$row"}{uncappedMean} < $minOn
825                 ) {
826                     $minOn = $circuit{"$circuitId1"}{truthTableMch}{"$row"}{uncappedMean};
827                 }
828
```

```
                    if (
                        $circuit{"$circuitId1"}{truthTableMch}{"$row"}{truthTableOutput} == 0 &&
                        $circuit{"$circuitId1"}{truthTableMch}{"$row"}{uncappedMean} > $maxOff
                    ) {
                        $maxOff = $circuit{"$circuitId1"}{truthTableMch}{"$row"}{uncappedMean};
                    }


                    if (
                        $circuit{"$circuitId1"}{truthTableMch}{"$row"}{truthTableOutput} == 1 &&
                        $circuit{"$circuitId1"}{truthTableMch}{"$row"}{mean} < $cappedMinOn
                    ) {
                        $cappedMinOn = $circuit{"$circuitId1"}{truthTableMch}{"$row"}{mean};
                    }

                    if (
                        $circuit{"$circuitId1"}{truthTableMch}{"$row"}{truthTableOutput} == 0 &&
                        $circuit{"$circuitId1"}{truthTableMch}{"$row"}{mean} > $cappedMaxOff
                    ) {
                        $cappedMaxOff = $circuit{"$circuitId1"}{truthTableMch}{"$row"}{mean};
                    }

                }

                $drMCh = $minOn - $maxOff;
                $cappedDrMCh = $cappedMinOn - $cappedMaxOff;

                $circuit{"$circuitId1"}{mchDynamicRange} = $drMCh;
                $circuit{"$circuitId1"}{mchCappedDynamicRange} = $cappedDrMCh;

            } else {
                $circuit{"$circuitId1"}{mchDynamicRange} = "Undefined";
                $circuit{"$circuitId1"}{mchCappedDynamicRange} = "Undefined";
            }
        }
    }




# Adds 1 mod 2 to a vector of bits, to generate all boolean functions in lexicographic sequence.
# The *global* vector of bits is called bitVector and is assumed to be initially vec zero.

sub getNextTT {
```

```
875
876         my $i;
877         my $carry = 1;
878
879         # Begin at the end of the bitVector, i.e. at index $#bitVector, and go all the way down to index
880         # 0. Look for an index that's 0. If you find it, toggle it, and you've added 1. You're done.
881         # If you see a 1, toggle it to 0, remember carry, and continue.
882
883         for ($i=$#bitVector; $i>-1 and $carry == 1; $i--) {
884             if ($bitVector[$i] == 0) {
885                 $bitVector[$i] = 1;
886                 $carry = 0;
887             } else {
888                 $bitVector[$i] = 0;
889             }
890         }
891
892         # If, at the end, you're holding a carry, it means you have an overflow, which in turn means
893         # you're done enumerating all boolean functions.
894         return ($carry ? 0 : 1);
895     }
```