Searching For Perfect Sequences

Jason Grout*

July 11, 2002

Contents

*grout@math.byu.edu

1 Introduction

In this paper, we implement an algorithm for exhaustively searching for perfect sequences of arbitrary lengths with elements that are arbitrary roots of unity. This algorithm was developed by the research group headed by Idris Mercer at the MSRI Computational Number Theory Workshop in June 2002.

To calculate perfect sequences of length n composed of mth roots of unity, run the program

`./msri` m n `<starting sequence>`

In order to search the entire search space, specify `<starting sequence>` as `0 0 ...0`, where there are n zeros.

The perfect sequences will be printed out, preceded by an indication that they passed the autocorrelation test. Sequences that are "almost" perfect–that had an autocorrelation of zero for all but the last few shifts–are also printed out, preceded by an indication that they failed the autocorrelation test (and the shift they failed on as well). Also, the progress through the search space is indicated by outputing the most recent major sequence that was checked.

2 Perfect Sequences

We consider a sequence of length n composed of mth roots of unity. A perfect sequence is such a sequence in which all the acyclic autocorrelations are zero. If the sequence under consideration is denoted $[a_0, a_1, \ldots, a_{n-1}]$, then the acyclic autocorrelation is defined as

$$\gamma_k := \sum_{j=0}^{n-1} \overline{a_j}\, a_{j+k},$$

where the $j + k$ addition is taken mod n.

We use a technique that seems computationally efficient to test if a sequence composed of mth roots of unity is perfect. We then iterate exhaustively through all sequences of length n, testing each sequence.

3 Program

The general structure of the program is as follows.

2 ⟨* 2⟩≡
⟨*Header Files* 14c⟩

⟨*Global Variables* 3a⟩

⟨*Global Functions* 4b⟩

⟨*Main Program* 10⟩

We will first discuss the various variables in the program (the "notation"), and then discuss our algorithm for testing a perfect sequence. Following that will be the implementation of our iterating process, the main program, and various other miscellaneous items.

4 Variables

We store the sequence length under consideration in `SEQUENCE_LENGTH`. The number of roots of unity we are considering is stored in `MAX_VALUE` (e.g., if we are considering 6th roots of unity, then `MAX_VALUE` is 6).

3a ⟨*Global Variables* 3a⟩≡ (2) 3b▷

```
int SEQUENCE_LENGTH;
int MAX_VALUE;
```

Defines:
`MAX_VALUE`, used in chunks 6a, 9, 10, 12a, 13b, and 16.
`SEQUENCE_LENGTH`, used in chunks 4a, 7b, 9a, 10, and 12–16.

We store the sequence under consideration in `sequence`. This variable is updated to exhaustively iterate through all sequences of size n. The values in `sequence` represent the powers of a primitive root of unity. This allows us to treat the roots of unity as a cyclic additive group of integers of order `MAX_VALUE`. The current autocorrelation we are considering (the shifted dot product of the sequence) is stored in `gamma`. The variables are defined as pointers to integers to allow us to dynamically allocate an arbitrary size array of integers in the program.

3b ⟨*Global Variables* 3a⟩+≡ (2) ◁3a 3c▷

```
int *sequence;
int *gamma;
```

Defines:
`gamma`, used in chunks 7b, 9, 10, 13b, 15, and 16.
`sequence`, used in chunks 10 and 12–15.

In the algorithm, it will be necessary to "cancel out" items in the sequence. Accordingly, we have a sentinel value we can set an element of the sequence or shift with to indicate that the value has been canceled. This sentinel value is `CANCELED`. It is necessary, then, that `CANCELED` be some integer that will not be used in a normal sequence. Since we are always working mod n because we are dealing with elements of the cyclic group, we will always have nonnegative values in `sequence` and `gamma`. Therefore, a negative value will suffice for `CANCELED`.

3c ⟨*Global Variables* 3a⟩+≡ (2) ◁3b 5▷

```
#define CANCELED -1
```

Defines:
`CANCELED`, used in chunks 7b, 9, 15, and 16.

5 Checking for a Perfect Sequence

The idea behind checking for a perfect sequence is straightforward. We are working in the cyclic additive integer group of order MAX_VALUE. An autocorrelation is just the sum of all the elements of gamma. In order for gamma to sum to zero, we must have elements in gamma cancel out with each other. Therefore, to check to see if we have a sum of zero, we just check that all the elements cancel out. This is equivalent to checking to see if gamma is composed of full cosets.

5.1 Check gamma

Here we check to see if the elements of gamma sum to 0. To do this, we call our generic check routine, which checks all possible cancellations. After calling this, gammaElementsLeft is 0 if all the elements of gamma cancel.

Note that this check will destroy the elements in gamma.

4a ⟨*checkGamma Variables* 4a⟩≡ (4b) 6b▷

```
int gammaElementsLeft=SEQUENCE_LENGTH;
```

Defines:
gammaElementsLeft, used in chunks 4b, 9b, and 16.
Uses SEQUENCE_LENGTH 3a.

4b ⟨*Global Functions* 4b⟩≡ (2)

```
int checkGamma(void)
{
  ⟨checkGamma Variables 4a⟩

  ⟨Check Condition 7b⟩

  /* The Hardcoded Check Condition is much faster,
     but needs to be modified depending on what roots
     of unity we are considering. */
  /*        <Hardcoded Check Condition>>*/

  /* If we have canceled all the elements, then we are successful.
     If there are elements left to cancel, then we failed. */
  if(gammaElementsLeft == 0)
    return 0;
  else
    return -1;
}
```

Defines:
checkGamma, used in chunk 13a.
Uses gammaElementsLeft 4a.

5.2 Generating Cosets

In order to know what we can cancel, we must first determine the cosets of the cyclic group. In the cyclic group, the cosets are generated by the divisors of `MAX_VALUE`. We store these divisors in the array `divisors`. The indices in `gamma` of the elements of a particular coset will be stored in the array `cosetElements`.

The value of `MAX_DIVISORS` sets an upper limit on the number of divisors or elements of a coset. This can be set arbitrarily large (but a larger value will slow the program down a bit.

5 ⟨*Global Variables* 3a⟩+≡ (2) ◁3c

```
#define MAX_DIVISORS 10

int NUM_DIVISORS;
int divisors[MAX_DIVISORS];
int cosetElements[MAX_DIVISORS];
```

Defines:
`cosetElements`, used in chunk 9.
`divisors`, used in chunks 6a, 7b, and 9.
`MAX_DIVISORS`, used in chunks 6a and 9a.
`NUM_DIVISORS`, used in chunks 6a and 7b.

To populate divisors, we divide MAX_VALUE by integers from MAX_VALUE/2 down to 1. We store an integer k in the divisors array if $k \equiv 0 \bmod$ MAX_VALUE. Since a larger k will have a smaller coset, we store the large values of k first. It seems that this will be more efficient for the cancellation step, because in general it will be faster to look for a coset having fewer elements. The last divisor stored is always 1, which corresponds to the whole cyclic group.

If we reach the end of the divisors array, we stop to avoid an overflow error. If this error comes up in running the program, it can be alleviated by increasing MAX_DIVISORS.

6a ⟨*Populate Divisor Array* 6a⟩≡ (10)

```
NUM_DIVISORS=0;
for(i=MAX_VALUE/2+1;i>0;i--)
{
  if(MAX_VALUE%i==0)
    {
      divisors[NUM_DIVISORS]=i;
      NUM_DIVISORS++;

      if(NUM_DIVISORS==MAX_DIVISORS)
        {
          fprintf(stderr,
                  "Error: Cannot store all divisors"\
                  "of %d (there are at least %d of them)\n",
                  MAX_VALUE, NUM_DIVISORS);
          assert(NUM_DIVISORS<MAX_DIVISORS);

        }
    }
}
```

Uses divisors 5, i 7a, MAX_DIVISORS 5, MAX_VALUE 3a, and NUM_DIVISORS 5.

5.3 Finding Cosets in gamma

The variable currentDivisor holds the index of the divisor in divisors that is generating the coset under consideration. The variable cosetElementsLeft keeps track of how many elements in a particular coset we still need to find to have a complete coset to cancel.

6b ⟨*checkGamma Variables* 4a⟩+≡ (4b) ◁4a 7a▷

```
int currentDivisor;
int cosetElementsLeft;
```

Defines:
cosetElementsLeft, used in chunks 7b and 9a.
currentDivisor, used in chunks 7b and 9.

For every element of a given shift `gamma`, we want to "cancel out" a full coset containing that element. To do this, we iterate through all of the elements of `gamma`. For each `gamma[i]`, if it has not already been canceled out (and hence is already a member of a coset), we search for a coset in `gamma` containing `gamma[i]`. A full coset is indicated by `cosetElementsLeft` equaling 0 (which means that all coset elements were found in the uncancelled elements of `gamma`). If we find a full coset, then we cancel out each element in the coset by setting it to `CANCELED`.

When we are done, either we have canceled out a coset or not. If we have not canceled out a coset, then `currentDivisor` will equal `NUM_DIVISORS`. In this case, we cannot cancel `gamma[i]`, and so we return with error.

7a ⟨*checkGamma Variables* 4a⟩+≡ (4b) ◁6b 8▷

```
int i;
```

Defines:
`i`, used in chunks 6a, 7b, 9a, 10, and 13–16.

7b ⟨*Check Condition* 7b⟩≡ (4b)

```
for(i=0;i<SEQUENCE_LENGTH;i++)
{
  if(gamma[i]==CANCELED)
    continue;

  /* Loop through all divisors */
  for(currentDivisor=0;currentDivisor<NUM_DIVISORS;currentDivisor++)
  {
    /* Check to see if we have a full coset with this divisor */
    ⟨Find Coset 9a⟩;

    /* If we have a full coset, then cancel the elements and break */
    if(cosetElementsLeft==0)
      {
        ⟨Cancel Coset 9b⟩;
        break;
      }
  }

  /* If we have reached the end of the divisor list without canceling,
     then we must have failed */
  if(currentDivisor==NUM_DIVISORS)
    {
      return -1;
    }
}
```

Uses `CANCELED` 3c, `cosetElementsLeft` 6b, `currentDivisor` 6b, `divisors` 5, `gamma` 3b, `i` 7a, `NUM_DIVISORS` 5, and `SEQUENCE_LENGTH` 3a.

The `Find Coset` procedure searches for a full coset generated by adding multiples of `divisors[currentDivisor]` to `gamma[i]`.

To do this, we first calculate the number of elements in this coset. This is stored in `cosetElementsLeft`. We immediately subtract one from this because we already have `gamma[i]`.

For each remaining element of `gamma[j]` of `gamma`, we test to see if `gamma[j]` is an element of the current coset. To test this, we see if we can add a multiple k of `divisors[currentDivisor]` to `gamma[i]` and obtain `gamma[j]`. If we can, and if we have not already found this element of the coset yet, then we store `j` in `cosetElements[k]`. If we have found a new element of the coset, then we decrement `cosetElementsLeft` and break if that was the last element.

Notice that we are adding `MAX_VALUE` to `gamma[i]-gamma[j]`. This is because the `%` operator in C/C++ is not a true "mod" operator, but a "remainder" operator. When we are working with nonnegative values, however, these are equivalent operations. Since the elements of `gamma` are guaranteed by construction to be nonnegative and less than `MAX_VALUE`, and we are working mod `MAX_VALUE`, adding `MAX_VALUE` will ensure that we are dealing with nonnegative values and that we will not be changing the answer.

8 ⟨*checkGamma Variables* 4a⟩+≡ (4b) ◁7a

```
int j;
```

Defines:
`j`, used in chunks 9 and 16.

9a ⟨*Find Coset* 9a⟩≡ (7b)

```
cosetElementsLeft=MAX_VALUE/divisors[currentDivisor]-1;
cosetElements[0]=i;

bzero(cosetElements, sizeof(int)*MAX_DIVISORS);

/* Search through the rest of [[gamma]] for cancellation */
for(j=i+1;j<SEQUENCE_LENGTH;j++)
{
  if(gamma[j]==CANCELED)
    continue;

  if(gamma[i]!=gamma[j] &&
     ((gamma[i]-gamma[j]+MAX_VALUE)%(divisors[currentDivisor])==0))
    {
      /* We have a divisor */
      /* If we do not have one for this slot of the coset,
         then store this information in [[cosetElements]] */
      if(cosetElements[((gamma[i]-gamma[j]+MAX_VALUE)%MAX_VALUE)
                       /(divisors[currentDivisor])]==0)
        {
          cosetElements[((gamma[i]-gamma[j]+MAX_VALUE)%MAX_VALUE)
                        /(divisors[currentDivisor])]=j;
          cosetElementsLeft--;
          if(cosetElementsLeft==0)
            break;
        }
    }
}
```

Uses CANCELED 3c, cosetElements 5, cosetElementsLeft 6b, currentDivisor 6b, divisors 5, gamma 3b, i 7a, j 8, MAX_DIVISORS 5, MAX_VALUE 3a, and SEQUENCE_LENGTH 3a.

5.4 Canceling Cosets

Once we have found a full coset, we must cancel out each element of the coset in gamma. We do this by setting each element of the coset to CANCELED.

9b ⟨*Cancel Coset* 9b⟩≡ (7b)

```
for(j=0;j<MAX_VALUE/divisors[currentDivisor];j++)
{
  gamma[cosetElements[j] ]=CANCELED;
  gammaElementsLeft--;
  cosetElements[j]=0;
}
```

Uses CANCELED 3c, cosetElements 5, currentDivisor 6b, divisors 5, gamma 3b, gammaElementsLeft 4a, j 8, and MAX_VALUE 3a.

6 Main Program

Here is the main program. We first set the values of MAX_VALUE and SEQUENCE_LENGTH from the command line. We then create the arrays sequence and gamma.

10 ⟨*Main Program* 10⟩≡ (2)

```
int main(int argc, char **argv)
{
  int i;
  int shift=1;
  int index;


  if(argc<3)
  {
    ⟨Print Usage 14b⟩;
    exit(-1);
  }


  MAX_VALUE = atoi(argv[1]);
  SEQUENCE_LENGTH = atoi(argv[2]);

  if(MAX_VALUE<3 || SEQUENCE_LENGTH<3)
  {
    ⟨Print Usage 14b⟩;
    exit(-1);
  }

  if(argc<(SEQUENCE_LENGTH+3))
  {
    ⟨Print Usage 14b⟩;
    exit(-1);
  }


  sequence = (int *)malloc(sizeof(int)*SEQUENCE_LENGTH);
  gamma = (int *)malloc(sizeof(int)*SEQUENCE_LENGTH);

  bzero(sequence, sizeof(int)*SEQUENCE_LENGTH);
  bzero(gamma, sizeof(int)*SEQUENCE_LENGTH);


  /* Populate the sequence */
```

```
⟨Populate Divisor Array 6a⟩

sequence[0]=sequence[1]=0;

/* Get starting sequence from command line */
for(i=0;i<SEQUENCE_LENGTH;i++)
{
  sequence[i]=atoi(argv[i+3]);
}

⟨Increment Sequence 12a⟩
{
  ⟨Check Shifts 13a⟩;

  /* If we get to here, then the sequence passes */
  printf("PASSED: ");

  ⟨Print sequence 14a⟩
  printf("\n");

  NEXT:
}
⟨Increment Sequence End 12b⟩

DONE:

free(sequence);
free(gamma);

exit(0);
}
```

Defines:
`main`, never used.
Uses `gamma` 3b, `i` 7a, `MAX_VALUE` 3a, `printf` 14b, `sequence` 3b, and `SEQUENCE_LENGTH` 3a.

6.1 Iterating Through Sequences

This is our incrementing procedure. It will increment the sequence in `sequence` to the next sequence in an odometer fashion. It will stop when we "roll over" the sequence back to $\{0, 0, \ldots, 0\}$.

To aid in debugging and seeing the program process the data, we print out the sequence every time the number halfway through the sequence rolls over.

From the nature of the problem, we may assume that all sequences start with a zero. This optimization is implemented in this incrementing routine.

12a ⟨*Increment Sequence* 12a⟩≡ (10)

```
while(1)
{
  sequence[SEQUENCE_LENGTH-1]++;
  sequence[SEQUENCE_LENGTH-1] %= MAX_VALUE;

  /* Check to see if we are wrapped around */
  index=SEQUENCE_LENGTH-1;
  while(sequence[index]==0)
  {
    index--;
    sequence[index]++;
    sequence[index] %= MAX_VALUE;
    if(sequence[index]!=0)
      break;

    if(index == (SEQUENCE_LENGTH/2))
      {
        printf("Finished sequences starting with ");

        ⟨Print sequence 14a⟩;

        printf("\n");
      }

    if(index ==1)
      {
        /* We are done. We have wrapped around the first index */
        goto DONE;
      }
  }
```

Uses `MAX_VALUE` 3a, `printf` 14b, `sequence` 3b, and `SEQUENCE_LENGTH` 3a.

Of course, we need to end the while loop that increments the sequence.

12b ⟨*Increment Sequence End* 12b⟩≡ (10)

```
}
```

6.2 Checking All Shifts

To check to make sure all the autocorrelations are zero, we only need to check the first half of the shifts. The other shifts are equivalent to shifts in the reverse direction. The variable `shift` runs through the shifts that we need to consider. The variable `gamma` is updated to reflect each value of `shift`. If the autocorrelation is not zero, and the shift is greater than a specific value, then we print out the sequence. Notice that if we fail at any particular shift, we move on to the next sequence. Thus, if we fail at a particular shift, it means we have passed all shifts less than that shift.

13a ⟨*Check Shifts* 13a⟩≡ (10)

```
for(shift=1;shift<(SEQUENCE_LENGTH/2+1);shift++)
{
  ⟨Update gamma 13b⟩;

  if(checkGamma())
  {
    /* Print out where we have failed */
    if(shift>5)
      {
        printf("FAILED: ");

        ⟨Print sequence 14a⟩
        printf(" at shift %d\n",shift);
      }

    /* Since we failed, go to the next sequence */
    goto NEXT;
  }
}
```

Uses `checkGamma` 4b, `printf` 14b, `sequence` 3b, and `SEQUENCE_LENGTH` 3a.

To update `gamma`, we subtract corresponding elements of the shifted sequence from the original sequence. Because we are working with the additive group of powers, this is equivalent to multiplying the roots of unity in the sequence with corresponding elements of the shifted, conjugated sequence. We also take the result mod `MAX_VALUE` to ensure the values of `gamma` are nonnegative and less than `MAX_VALUE`. (Notice we have to add `MAX_VALUE` to the value before taking the modulus because of the behavior of `%` in C.)

13b ⟨*Update* gamma 13b⟩≡ (13a)

```
for(i=0;i<SEQUENCE_LENGTH;i++)
{
  gamma[i] = ((sequence[i]-sequence[((i-shift)+SEQUENCE_LENGTH)%SEQUENCE_LENGTH])
                + MAX_VALUE) % MAX_VALUE;
}
```

Uses `gamma` 3b, `i` 7a, `MAX_VALUE` 3a, `sequence` 3b, and `SEQUENCE_LENGTH` 3a.

7 Printing

Here are the output functions.

7.1 Print a Sequence

Here is our generic print sequence function.

14a ⟨*Print* `sequence` 14a⟩≡ (10 12a 13a)

```
for(i=0;i<SEQUENCE_LENGTH;i++)
{
  printf("%d ", sequence[i]);
}
```

Uses `i` 7a, `printf` 14b, `sequence` 3b, and `SEQUENCE_LENGTH` 3a.

7.2 Program Usage Message

The usage message for the program.

14b ⟨*Print Usage* 14b⟩≡ (10)

```
printf("Usage: %s <Root of Unity> <Sequence Length> <Starting Sequence Values>\n\n",
       argv[0]);

printf("<Root Of Unity> and <Sequence Length> must each be greater than 2.\n");
printf("<Starting Sequence Values> is a space delimited listing \n"
       "of length <Sequence Length> of nonnegative integers \n"
       "less than <Root of Unity>.\n");
```

Defines:
`printf`, used in chunks 10 and 12–14.

8 Header Files

We have the standard header files.

14c ⟨*Header Files* 14c⟩≡ (2)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
```

9 Further Work

Further work on this program could include:

- Documenting rigorously the reasons behind some of our ideas and optimizations, including
  - Checking only half the shifts.
  - Normalizing the sequence so the starting element can always be assumed to be 0.
- Integrating in specialized functions to take care of small cases we are interested in, such as the quartic or sextic roots of unity sequences.
- Ending the sequence search space at a specified sequence.
- Modify the program to allow the sequence lengths and roots of unity below three.
- In some cases, we can assume that the first *two elements of the sequence are zero. We could modify the program to detect these cases and apply this optimization.*
- *Take advantage of the symmetry among the different shifts to narrow the search space quite a bit (e.g., perfect sequences are invariant under shifts).*

A Hardcoded Checks

Here is a hardcoded check for the quartic case. This seems about twice as fast as the general method on the quartic case.

15 ⟨*Hardcoded Check Condition* 15⟩≡

```
int fail=0;

/* We search through every element of the sequence */
for(i=0;i<SEQUENCE_LENGTH;i++)
{
  /* If we have yanked the element, then continue */
  if(gamma[i]==CANCELED)
    continue;

  ⟨Hardcoded Quartic Divisor Check 16⟩;

  /* If we have failed, return with error */
  if(fail)
    return -1;
}
```

Defines:
`fail`, used in chunk 16.
Uses `CANCELED` 3c, `gamma` 3b, `i` 7a, `sequence` 3b, and `SEQUENCE_LENGTH` 3a.

Here is the actual check.

16 ⟨*Hardcoded Quartic Divisor Check* 16⟩≡ (15)

```
fail=1;
for(j=i+1;j<SEQUENCE_LENGTH;j++)
{
  if(gamma[j] == CANCELED)
    continue;

  if(gamma[j] == ((gamma[i]+2)%MAX_VALUE))
    {
      gamma[i]=CANCELED;
      gamma[j]=CANCELED;
      gammaElementsLeft -= 2;
      fail=0;
      break;
    }
}
```

Uses CANCELED 3c, fail 15, gamma 3b, gammaElementsLeft 4a, i 7a, j 8, MAX_VALUE 3a, and SEQUENCE_LENGTH 3a.

Index

Identifiers

Code Chunks