Theory and Implementation of Generating Arrays

Gabriel Lein
glein@marlboro.edu

December 12, 2005

1 Preamble

Latin squares have been used for centuries for purposes ranging from agriculture to research. Squares with more (or in some cases, fewer) conditions can be applied, and are essential to, other unique real world situations. But to do this, one must able to construct a square of necessary form and adequate dimension efficiently. Unfortunately, brute force methods quickly become infeasible due to the factorial growth of the problem. To combat this, a variety of methods have been constructed to find the more complex squares without trying all possible permutations. Our goals in this project were as follows:

- Explore and push the boundaries of the already known and implemented *directed terraces* using a C++ implementation.
- Demonstrate the theory for *generating arrays* and their feasibility as a method for effectively constructing Tuscan-k and Roman-k squares.
- Expand the directed terraces program to search for these generating arrays.

This paper assumes a casual knowledge of Latin squares and group theory, and some familiarity with the Unix environment, but should be readable to both non-mathematicians and non-computer scientists as well.

1.1 Definitions

Before moving into the workings of directed terraces, let's define the main terms we'll be using.

Definition 1.1 *For an $n \times n$ Italian square, take a $\{a, b\}$ pair to mean two distinct values of a row. If for $1 \leq m \leq k$, the symbol b appears at most once m places to the right of a, we call this square* ***Tuscan-k****. If the square is Latin, it is called* ***Roman-k****.*

Figure 1: A Roman-5 square of order 6

0	2	1	4	5	3
1	3	2	5	0	4
2	4	3	0	1	5
3	5	4	1	2	0
4	0	5	2	3	1
5	1	0	3	4	2

If $k = 1$ we will simply refer to the square as being Tuscan or Roman. If $k = n-1$ and the square is Italian, we call it **Florentine**. If it is Latin, we call it **Vatican**. Tuscan-k and Roman-k squares can be used to avoid bias in tests. For example, if we're rating wines, and wine α is particularly awful, then whatever wine is tasted immediately afterwards will probably get a higher rating than it would were it tasted after a better wine. If wine β always followed α in the test, it would leave an unresolved systematic error. Had we arranged the tests as the rows of a Tuscan square however, each wine would recieve the same bias adjustments and the systematic error would be eliminated.

An efficient method to find these squares is by finding *directed terraces.*

Definition 1.2 *Take a sequence* $\mathbf{a} = (a_1, a_2, a_3, ..., a_n)$ *containing all elements of* $\mathbb{Z}_n$*. Create from this a second sequence,* $\mathbf{b} = (b_1, b_2, b_3, \ldots, b_{n-1})$*, where* $b_i = a_{i+1} - a_i$*. If there are no repeats in* $\mathbf{b}$*, then* $\mathbf{a}$ *is called a* ***directed terrace****, and* $\mathbf{b}$ *is its* ***associated sequencing****.*

A valid associated sequencing cannot contain zero, since this would mean two terms of the terrace were identical.

Theorem 1.1 *If a directed terrace exists for a given order, then a Roman square exists for that order as well.*

Proof: If we take a directed terrace of order n as row 0 of our square, and generate row i, with $1 \leq i \leq n$, by adding i to each element of the terrace, each column contains all elements of $\mathbb{Z}_n$. Thus we have constructed a Latin square of order n that is also Roman. $\square$

Conversely, if we simply use any directed terrace of order n for each row (including, possibly, repeats), we can construct a Tuscan square of order n. Directed terraces are part of a more open class of sequences called *terraces*, that do not require each element of the associated sequencing to be unique.

To find Tuscan-k/Roman-k squares where $k > 1$, we most first generalize our definition of a directed terrace.

Definition 1.3 *Take a directed terrace* $\mathbf{a}$*, as defined in Definition 1.2 with associated sequencing* $\mathbf{b}^{(1)}$*. Define* $\mathbf{b}^{(m)}$ *as the sequence where* $b_i = a_{i+m} - a_i$*. If we treat all* $\mathbf{b}^{(m)}$ *for*

$1 \leq m \leq k$ *as our associated sequencings, and all* $\mathbf{b}^{(m)}$ *are valid up to k, then we have a* ***directed*** $\mathbf{T_k}$ ***terrace****.*

Example 1.1 *Assume we are working in* $\mathbb{Z}_{10}$.

$$\mathbf{a} = \{0, 3, 4, 6, 2, 7, 1, 9, 8, 5\}$$

is a T_2 *terrace with associated sequencing* $\mathbf{b}^{(1)}$ *such that*

$$\mathbf{b}^{(1)} = \{3, 1, 2, 6, 5, 1, 8, 9, 7\}$$

and associated sequencing $\mathbf{b}^{(2)}$ *where*

$$\mathbf{b}^{(1)} = \{4, 3, 8, 1, 6, 9, 7, 6\}$$

We're now ready to continue on and explore some properties of these directed terraces.

2 Properties of T_k terraces and current knowledge boundaries

While Tuscan and Roman squares exist for odd orders, directed terraces do not. Clearly this means the absence of a T_k directed terrace is not enough to disprove the existence of a Roman-k/Tuscan-k square (see Figure 2).

Theorem 2.1 *There is no directed* T_k *terrace of odd order.*

Proof: Assume we have a directed terrace $\mathbf{a}$ and associated sequencing $\mathbf{b}$ in $\mathbb{Z}_n$ where n is odd. We can say that:

$$a_n = a_1 + b_1 + b_2 + b_3 + ... + b_{n-1}$$

Since $\mathbf{b}$ contains the non-zero elements of $\mathbf{a}$, we can rearrange the values so that

$$a_n = a_1 + (n-1) + 1 + (n-2) + 2 + ... + (n - (n/2+1)) + (n - (n/2))$$

$$a_n = a_1$$

Which contradicts our definition. Since a directed T_k terrace must also be T_1, no directed T_k terrace exists. □

Definition 2.1 *The following terrace construction was used implicitly by E. Lucas in [2], who gave credit to Walecki. It was also used explicitly by E. J. Williams in [4], and is thus called the LWW terrace. In* $\mathbb{Z}_{2r}$*, let:*

$$\mathbf{a} = (0, 1, 2r-1, 2, 2r-2, 3, ..., r+1, r)$$

Then $\mathbf{a}$*'s associated sequencing is:*

$$\mathbf{b} = (1, 2r-2, 3, 2r-4, 5, ..., 2, 2r-1)$$

It then follows that

Theorem 2.2 *There is a Roman square of every even order.*

Proof: By example 2.1 and theorem 1.1, we can construct a directed terrace of even order and use it to generate a Roman square. □

Definition 2.2 *Two directed T_k terraces with the same associated sequencing(s) are equivalent.*

Definition 2.2 stems from the fact that $\mathbb{Z}_n$ is a cyclic group. A directed T_k terrace starting with zero can generate one starting with x where $x \in \mathbb{Z}_n$ simply by adding x to all elements. This means if we find one terrace, we can extrapolate all terraces equivalent to it.

The follow is a list of known facts prior to our investigation [3]:

- When $n + 1$, there is a directed T_{n-1} terrace in $\mathbb{Z}_n$.
- There is a directed T_2 terrace in $\mathbb{Z}_n$ for all even n up to 50.
- There is no directed T_3 terrace in $\mathbb{Z}_8$, $\mathbb{Z}_{14}$, or $\mathbb{Z}_{20}$

$\mathbb{Z}_{24}$ and $\mathbb{Z}_{26}$ were generally unexplored beforehand, and $\mathbb{Z}_{26}$ largely still is.

3 Implementation of a directed T_k terrace search program in C++

Implementing a search for T_k terraces was certainly not new, but creating such a program was essential before taking the step into generating arrays. C++ was chosen not only because it is the language I am most familiar with, but because it is faster than most other high level languages. In a situation where computer speed is the greatest limiting factor, we want as much bang for our buck.

One downside of C/C++ is the static nature of arrays, and the inability to (easily) declare an arbitrary number of variables or arbitrarily named variable at runtime. These will be discussed more further on.

Figure 2: A Tuscan square of order 7

2	6	3	5	4	0	1
6	1	5	2	4	3	0
5	0	2	3	1	4	6
0	6	5	3	4	1	2
3	6	2	1	0	4	5
1	3	2	0	5	6	4
4	2	5	1	6	0	3

3.1 Using *squares3*

Squares3, the current version of the program, is designed to be easy to use and understand, but not to be “idiot-proof”. Checks are not put in place to catch obvious typos, meaning that entering mis-formatted or blatantly wrong data could cause the program to crash or behave erratically. Also, it is worth noting that since the program is designed to run for a long time, the user should be extra careful that they are looking for what they intend to. Failing to do so could waste days or even weeks of computing time. Squares3 was coded with execution on a Unix-based machine in mind, and has only been tested on such machines, but should compile and run on other systems as well.

Figure 3: A sample execution of squares3

```
 mdhcp5-248: /latinsquares gabe$ ./squares3
Enter length for the directed terrace:
10
And Tk value:
2
Made terrace array of length 10, and sequence array of length 17
Specify initial values?  [y/n]y
Specify initial values, separated by spaces:
0 1
01-1-1-1-1-1-1-1-1
1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
Terrace Found!
Terrace Found!
mdhcp5-248: /latinsquares gabe$ cat terracefile.txt
0 1 3 7 4 9 2 8 6 5
0 1 8 2 4 9 7 3 6 5
```

One “feature” that actually affects speed is the specification of starting values. Following from Definition 2.2, we know that by simply setting the first element, we can reduce our search time by a factor of $\frac{1}{n}$. Through some mathematical trickery, we can eliminate many options for consecutive elements, meaning we’ve drastically reduced our run-time merely by narrowing the playing field. (This is both hard to code and time-consuming by hand, and was decided not worth pursuing beyond two or three element places.)

As terraces are found, they are outputted to the file `terracefile.txt`, and the message “terrace found” is outputted to the terminal. The save file is specified internally, meaning it cannot be changed without modifying the code (this can be done easily by modifying the name of the “FILENAME” global variable at the top of the code). This also means that each time the code is run, it over-writes the file. To avoid this, the user is advised to rename `terracefile.txt` or move it to another directory after each execution. Because the program only opens the file while it’s actually writing to it, it’s safe to inspect the file while the program is running. For a sample execution, see figure 3. Future versions of squares3 may include dynamic filenames and an option to quit after finding a single terrace.

3.2 Underpinnings and development choices

At the core of squares3 is a simple backtrack permutation algorithm. The directed terrace is stored in an array of length n, and all k sequences are stored in a single array of length $n+(n-1)+(n-2)+...+(n-k)$, where different sections correspond to different k values. This is necessary since, as mentioned before, variables cannot be arbitrarily specified as needed at run-time, and the T_k value is specified by the user. Since the arrays are only "full" when we have a valid directed T_k terrace, we set the "unfilled" elements to -1. While not as elegant as some other methods, this is quick and caters to the language's needs. The original squares1 was designed to store the data in linked lists. While this would have been a lot easier to conceptualize and could simulate the effect of a dynamic-sized array, it would have been unable to efficiently deal with T_k's greater than 1, and been horrendously slow regardless. (All elements of an array are stored in one solid block of memory. To access one, we need merely add the proper amount to the array's memory address and access it directly. Linked lists on the other hand, are scattered throughout memory, using memory address pointers to connect to others. This means if we wanted to access the nth memory node in the list, we'd have to first access all $(n-1)$ nodes before it. Needless to say, this takes a lot of time.)

3.3 Results

The program was run concurrently on three machines: a Macintosh G4 tower, a Macintosh G5 tower, and a Saegar Pentium 4 laptop. Squares3 was compiled on each computer, using several versions of the gcc compiler with hardware optimization at its highest. The difference between the standard and the optimized compile was drastic on all machines, but most so using Apple's gcc version with G5 processor optimization. A task that took over a day when un-optimized completed in under two hours, so fast in fact, that we were obligated to investigate if it was still functioning fully.

Dividing up the task and specifying the first two elements at run-time, we were able to completely search $\mathbb{Z}_{24}$ for T_3 terraces in under two weeks (note that none of the computers were running the entire time). Unfortunately we found nothing. Discouraging, after so much work, but not entirely unexpected. We also did a few preliminary searches in $\mathbb{Z}_{26}$, and while we haven't found anything there yet, we haven't ruled it out either.

4 Generating Arrays

Creating a program to find directed terrace of course was only the first step. Before continuing into the next program however, we must define some more theory. *Generating arrays* were used by Matt Ollis in 2005 in an attempt to circumvented some of the short-

Figure 4: A 2×6 Italian generating array

$$\begin{array}{cccccc} (0,0) & (2,0) & (0,1) & (1,0) & (1,1) & (2,1) \\ (0,1) & (2,1) & (2,0) & (1,1) & (0,0) & (1,0) \end{array}$$

comings of directed terraces, such as their uselessness in looking for Tuscan and Roman squares of odd order (see Theorem 2.1).

4.1 Definitions and Theory

We are not limited to creating Latin and Italian squares solely out of $\mathbb{Z}_n$. Intuitively, we might use letters as our symbols (as is often done when representing mutually orthogonal Latin squares), with "addition" defined as a cycling through the alphabet. In fact, as long as we have a cyclic group of correct order, we can create a square of its elements where our previous definitions hold.

Definition 4.1 *Let n be an integer where $n = pq$ Let the pair (a, c) be an element of $\mathbb{Z}_p \times \mathbb{Z}_q$ (meaning $a \in \mathbb{Z}_p, c \in \mathbb{Z}_q$), and let addition be defined as mod p in the first co-ordinate and mod q in the second. Then an array of q rows and n columns is called a* $\mathbf{q} \times \mathbf{n}$ ***Italian generating array*** *if every symbol of $\mathbb{Z}_p \times \mathbb{Z}_q$ appears in each row [3].*

From this definition, it is easy to show that

Theorem 4.1 *If there exists a $q \times n$ Italian generating array A, we can generate an $n \times n$ Italian square.*

Proof: Assume we have a $q \times n$ generating array. For each row of A, generate p rows of the form

$$(a_1 + i, c_1) \quad (a_2 + i, c_2) \quad (a_3 + i, c_3) \quad \ldots \quad (a_n + i, c_n)$$

where $0 \leq i \leq p - 1$. We then have n rows of length n, which we can use as our Italian square. $\square$

It should be obvious that by making each column contain every element as well, we can make a Latin square, and the array would then be a *Latin generating array.* To generate the other rows of the array, we can just use the first row and cycle the value of the second co-ordinate mod q. To create the square from this, we take each row of the array and cycle the first co-ordinate mod q. The construction for a *Tuscan/Roman* generating array is a close analog to its directed T_k terrace counterpart, but with a bit of a twist due to the two co-ordinates.

Definition 4.2 *Assume we have an Italian generating array with elements of the form (a_i, c_i). Then for each pair of second co-ordinates (c_i, c_{i+m}), if $a_{i+m} - a_i$ for the corresponding first co-ordinates appears at most once in each row for $1 \leq m \leq k$, then the array is* ***Tuscan*-k***. If the array is also Latin, then it is* ***Roman*-k***.*

Figure 5: A $\mathbb{Z}_2 \times \mathbb{Z}_3$ Latin square generated from Figure 4 (that is also Roman)

$$\begin{array}{cccccc}
(0,0) & (2,0) & (0,1) & (1,0) & (1,1) & (2,1) \\
(1,0) & (0,0) & (1,1) & (2,0) & (2,1) & (0,1) \\
(2,0) & (1,0) & (2,1) & (0,0) & (0,1) & (1,1) \\
(0,1) & (2,1) & (2,0) & (1,1) & (0,0) & (1,0) \\
(1,1) & (0,1) & (0,0) & (2,1) & (1,0) & (2,0) \\
(2,1) & (1,1) & (1,0) & (0,1) & (2,0) & (0,0)
\end{array}$$

Example 4.1 *Using the generating array from Figure 4, we make a table of first coordinate differences for each value of m.*

(c_i, c_{i+m})	$m=1$	$m=2$	$m=3$	$m=4$	$m=5$
$(0,0)$	$2,1$	$2,1$	$1,2$		
$(0,1)$	$1,0,2$	$0,2$	2	$0,1$	2
$(1,0)$	$1,0,2$	$2,0$	1	$0,2$	1
$(1,1)$	$1,2$	$1,2$	$1,2$		

*There are no repeats in any cell of our m table up to $m = 5$, and the elements of all the array's columns are unique, so the array is **Roman-5**, or **Vatican**. If the array had been Roman-4 and not Roman-5, then some cell(s) in the $m = 5$ column would have a repeat (but the other columns would still be valid).*

For each m column, we should have a total of $2n - m$ entries. However, we have no guarantee as to the placement of those entries.

5 Implementation of the generating arrays program

Genarrays is designed to output Roman-k generating arrays. While the directed terraces program created much of the framework for this next task, several notions had to be thrown out due to the added complexity. Because of this complexity and time constraints on development, genarrays in its current form only searches for $2\times n$ arrays. This obviously limits its applications significantly, and makes looking for odd ordered squares, one of the motivators for its design, impossible. It is however, still capable of verifying the conceptual soundness of the method.

5.1 Using genarrays

Most of the user interface – and thus flaws – of squares3 is identical in genarrays. The user is still advised to check themself to avoid (accidentally) pursuing a nonsensical value. The user can specify as many elements of each row as they desire, however for now they can only specify sequencial values starting from the left. Future versions may include support for arbitrary specification, but this is largely unneccessary, since the user can search the whole range in (usually) very little time.

Once running, genarrays outputs the current associated sequencing (or "*m*") array to the console each time it modifies it. Found arrays are outputted to `arrayfile.txt`. As before, this file is overwritten each time genarrays is run.

Figure 6: Abbreviated execution of genarrays

```
mdhcp5-248: /latinsquares gabe$ ./genarrays
Enter length of array:
4
And m value:
3
Made array of length 4, and sequence array of length 12
Specify initial values?  [y/n]y
Specify initial values for row 0, pairs separated by spaces and values
separated by commas:
0,0
Specify initial values for row 1, pairs separated by spaces and values
separated by commas:

Made initial array:
(0,0) (-1,-1) (-1,-1) (-1,-1)
(-1,-1) (-1,-1) (-1,-1) (-1,-1)
Made initial sequence:
(-1,-1,-1) (-1,-1,-1) (-1,-1,-1) (-1,-1,-1) (-1,-1,-1) (-1,-1,-1) (-1,-1,-1)
(-1,-1,-1) (-1,-1,-1) (-1,-1,-1) (-1,-1,-1) (-1,-1,-1)
⋮
mdhcp150: /latinsquares gabe$ cat arrayfile.txt
(0,0) (0,1) (1,1) (1,0)
(0,1) (1,0) (0,0) (1,1)

(0,0) (0,1) (1,1) (1,0)
(1,1) (0,0) (1,0) (0,1)

(0,0) (1,1) (0,1) (1,0)
(0,1) (0,0) (1,0) (1,1)

(0,0) (1,1) (0,1) (1,0)
(1,1) (1,0) (0,0) (0,1)
```

While checking the sequencing is helpful for debugging, it outputs a *lot* of text for large arrays. To avoid this, the user is advised to set genarrays to run in the background (but only **after** entering all data).

5.2 Implementation choices and obstacles

The biggest challenges in coding genarrays were coping with multiple rows, and using different data structures for the generating array and the sequence array. By limiting the rows to 2 for the time being, I was able to simplify my design process while still investigating greater row numbers. The program generates one row at a time, meaning it would take very little to extend this into a loop. The multiple rows also affects the sequencing array however, which is more complicated to modify.

Translating the concept of one big sequencing array to genarrays is fairly intuitive but messy. We're still dividing the array into logical "sections" for each m value (only with sections twice as long since we're checking two rows). Since we now have to keep track of the (c_i, c_j) pair as well, it was necessary to create a data structure for the sequencing array. The array is still one dimensional, but now contains pair data and a value. Storing data in a form closer to the table used in Example 4.1 would be possible, but slower and more memory intensive since we don't know how many entries per cell we'll have (and thus must allot potentially unnecessary space in each). By using the sequencing array, we needn't worry where in the array we place an element. This adds another dynamic however. Elements in the generating array now have an a and a c value, but don't have pair data, meaning we don't want to store them as the same data structure. Consequently, using elements of one array to set elements of the other is no longer so straight forward.

Most other modifications were negligible or not related to the actual workings of the algorithm. Future versions will undoubtedly include the option to look for non-Roman Tuscan arrays.

5.3 Results

Tests of genarrays have been regulated mostly to small, known values for verification, with only recent searches into new territory. Current results are promising. We have already shown that there are no Roman-3 generating arrays for $n = 10, 12$ that are not also Roman-4 (prior to this it was only known there were no T_3 terraces that were not T_4). In fact, we may still discover a Tuscan/Roman T_3 square of order 24, though there are still plenty of smaller ranges to investigate first. Genarrays has not yet been bench-marked against its predecessor. It's safe to assume though that due to the extra calculation, it will run slightly slower (while at the same time finding more solutions).

6 Conclusion

Squares3 has succeeded in its purpose to date. Further revisions can be made to enhance efficiency and usability, but the greatest limiting factor now is hardware. It took three computers (one being several years old) several days to tackle $\mathbb{Z}_{24}$, meaning it would take them upwards of a month or more to find all or any solutions for $\mathbb{Z}_{26}$. If however we used or developed some distributed computing software as a "shell", such as the BOINC project [1], and set a computer lab or the general populace to the problem, then $\mathbb{Z}_{26}$ and even $\mathbb{Z}_{28}$ would be calculable in very little time.

It would probably be fair to call genarrays a success at this point. We haven't yet found anything of particular merit but have demonstrated the feasibility and execution of a generating array finding program and raised the bar slightly for what is known. With just a little more work, we will have a tool for finding odd composite order arrays as well. Because of its promise, let us call genarrays a tentative victory, as future versions will put even more new results within our reach.

References

[1] http://boinc.berkeley.edu/.

[2] E. Lucas. *Récréations Mathmatiques, Tôme II.* Albert Blanchard, Paris, 1892.

[3] M. A. Ollis. Notes on italian squares and generating arrays. 2005.

[4] E. J. Williams. Experimental designs balanced for the estimation of residual effects of treatments. *Aust. J. Scient. Res. A*, 2:149–168, 1949.