Analyzing Iteration-Space Performance Tuning Using the Deriche Recursive Filtering Algorithm

By: Mary Glaser
Advisor: Dave Wonnacott
Haverford College
Department of Computer Science
April 30, 2018

Abstract

No matter how efficient or satisfactory an algorithm or program may be, we as computer scientists are always looking to improve our programs by means of speeding them up or making them more accurate. In this paper, we will focus on finding the best way to optimize programs by using iteration-space performance tuning methods. We hope to contribute towards the bigger picture of understanding whether it is possible to do iteration-space performance tuning by studying a version of the Deriche Recursive Filtering Algorithm in C, which applies edge detection and smoothing to 2D images. Among the many different existing types of optimization techniques, we will be focusing on classic optimization and iterator-based optimization. With Deriche, we will first compare runtime results of the original code in C with the performance of classic optimization techniques and PluTo (a specific type of iteration-space optimization). Then we will convert the Deriche algorithm to the language Chapel and apply more direct iterator-based optimization techniques to further study its performance.

Contents

1 Introduction ........................................ 2
  1.1 Iteration-Space Performance Tuning ............... 2
  1.2 Previous Studies of Chapel ........................ 3
  1.3 Motivation ...................................... 4
  1.4 Tools and Notation ................................ 4

2 Literature Review .................................. 5
  2.1 Optimization By Reducing the Amount of Work ........... 5
  2.2 Optimization by Changing the Order of Operations ........ 6
  2.3 Optimization Using PluTo .......................... 7
  2.4 Iterator-based Optimization ........................ 8
  2.5 Deriche Recursive Algorithm ....................... 8
  2.6 Using ISCC with the Deriche Algorithm ............... 8

3 Background ........................................ 11
  3.1 An Idea for Changing the Order of Operations ............ 11
  3.2 Checking Correctness using ISCC .................... 11
  3.3 Example of an Incorrect Transformation ............... 13

4 Results From Background Discussion ................... 14
  4.1 Manual Optimization ............................. 15
  4.2 PluTo Optimization ............................... 15
  4.3 Remarks ........................................ 15
1 Introduction

There are many known efficient algorithms in the world, but we as computer scientists are always looking to improve our programs, either by means of speeding them up or making them more accurate. In this paper, we explore the various different ways to improve a program. For our research, we will be focusing on applying iteration-space performance tuning techniques on a particular algorithm. In general, we hope to be able to separate out the parts of a program that make it fast from the parts that ensure its correctness. By doing this, it makes it much easier to attempt to speed up the program without having to worry about breaking it.

We will look at the problem of optimizing dense matrix codes by making them faster in several different approaches: classic and manual optimization, Pluto optimization, and iterator-based optimization with the language called Chapel. We will use the Deriche Recursive Filtering Algorithm, which applies edge detection and smoothing to 2D images, as an example benchmark. We will first compare the performance of the original C code in Deriche with those of transformed equivalent programs from classic optimization techniques and Pluto optimization. After this, we will convert the code into Chapel and see if we can use iterators to manipulate the iteration of the code and further improve its performance.

1.1 Iteration-Space Performance Tuning

The primary goal of iteration-space performance tuning is to improve the efficiency of an algorithm without changing the original meaning of the program. There are several different ways to do this; in this particular paper, we will be heavily focusing on the order of operations that are being computed and where they are stored in memory. After determining if it is possible to apply these optimizing techniques involving order of operations and memory storage, we will execute them and compare the new programs' performance with that of the original algorithm.
As Collard explains, a program transformation is one that changes the text of the program but does not change the resulting output [Col03]. Our aim is to transform the program so that for all cases, the output would be the same as the output that the original would have produced. Thus, we must be careful about which operations to change and how exactly we would reorder a series of operations. This is why it is extremely useful for us to find ways to separate out the parts of the program that make it fast from those that make it correct or equivalent to the original program. Again, the most important part of iteration-space performance tuning is that we are not changing the essence of the program [Col03]. Note that if we cannot depend on having an equivalent program, there is no point attempting to speed it up.

1.2 Previous Studies of Chapel

Previous research has already been done regarding iteration-space performance tuning. Bertolacci et. all applied diamond tiling techniques to Jacobi 1D and 2D benchmarks in an effort to speed up the program. They furthermore combined these techniques with running the Jacobi benchmarks in Chapel. They found that diamond tiling provided better, faster results, while using Chapel was comparable to the original benchmark, written in C [BOH +15].

Primordial Machine Vision Systems (PMVS) conducted an entire study of Chapel, exploring the uses of the language for image processing problems. They experimented with color conversion, Gabor filters, the k-means clustering algorithm, FAST corner detection, and the Random Sample Consensus algorithm. To do this, they used several types of data structures, parallel programming, generic programming, and iterators. After these experiments with image processing algorithms, they noticed that Chapel is significantly slower than C.

PMVS's research was primarily based on experimenting with Chapel and getting to know the language [Sys15]. As explained previously, what we ultimately aim to do is to see if we can express programs, specifically by using tools such as iterators and records, that will allow it to be easier for us to improve performance without having to worry about breaking the program. So, we are much more interested in, for example, the specific usage of iterators, than PMVS was. PMVS was more focused on understanding how Chapel as a language compares to other languages, for the very specific application of image processing, rather than how using Chapel iterators compare to other optimization techniques for a program.

Furthermore, at the time of PMVS's experiments in October of 2015, the latest Chapel version available was Chapel 1.12. Since then, new versions of Chapel have been released (the latest release of Chapel is version 1.16) that might have been able to run these particular algorithms faster [Sys15]. We are more interested in the best way to speed up a program, and we will use a benchmark that is related to image processing. Unlike PMVS, we will focus more on the use of iterators in our experiments. Thus, we cannot directly compare our performance results to those of PMVS, both because our research questions are slightly different and because we are using a newer version of Chapel that is specifically much faster for iterators. PMVS also reported a number of reliability problems that they attributed to the compiler [Sys15]. While we did not experience such problems, we recognize that our experiments were done on a much more limited scale, so we can neither confirm nor deny that these problems exist, especially since we used a more modern version of Chapel.

Our aim in this paper is to test another benchmark, namely the Deriche Recursive Filtering Algorithm, to contribute to the research on iteration space transformations and performance-tuning. After several of these benchmarks are studied in Chapel, the overall goal is to demonstrate or disprove the claim that it is possible for programmers to perform iteration-space tuning without having to rewrite their own code completely. The ideal situation would be to have a level of rewriting the code that falls between manually rewriting the entire program oneself, and using PluTo, which automatically rewrites the entire code in the best way it decides is possible. We want to reach a middle ground between these two extremes so that the programmer can control the level of automation when redesigning the code. Essentially, we want to be able to rewrite the code without having to worry about mistakenly changing the output of the algorithm, which would thus render it an incorrect transformation. We will see later that the usage of iterators in Chapel will be extremely helpful for this exact purpose.

Furthermore, if there is a benchmark that we already know works, rewriting it is not only time consuming but also most likely frustrating. By changing a program that already works, we run into the risk of the program crashing much more often than desired, which leads to spending more time
debugging the code. If this entire process could be shortened by sharing iterators and making use of others’ performance findings, programmers could find faster algorithms for these benchmarks in less time. Thus, we as a society could make more and more discoveries of efficient benchmarks and algorithms, leading to more successful technology being produced more frequently. If, by the end of this paper, we find that it is worth pursuing using Chapel for iterator-based performance tuning, we will look forward to comparing Deriche in C with Deriche in Chapel in order to contribute to the ongoing research of iteration-space transformations.

1.3 Motivation

As previously discussed, we always want as much efficiency and speed as possible in our programs; taking less time to run a program means less battery time and less power required to run it. It is worth studying this problem because it would be extremely convenient for programmers to be able to share iterators with each other. It would also promote a reliable way to explore and alter fast algorithms without having to worry about breaking code that is known to work already. For this reason, rewriting programs in Chapel rather than rewriting them in the original language is an appropriate approach. Furthermore, Chapel supports iterators better than C does, and although there are other languages that appropriately support iterators, such as Python, these languages are not known to be very fast (see more discussion of this in Section 2.4).

With regards to the Deriche algorithm and edge detection specifically, I personally find 2D image processing intriguing, and recognize that there are several applications for edge detection in everyday life. These include face recognition, vehicle detection, traffic queue in motion, or even medical purposes [GRK13]. One could imagine edge detection is useful for sports; when relaying back videos of a track-and-field contest to determine which of several runners crossed the finish line first, the edge detection algorithm might be useful if it’s too difficult to tell by the naked eye. A more vital application of the algorithm would be when a person needs to have retinal blood vessels extracted [GRK13]. In this case, it is extremely important to know exactly where and what shape the blood vessels are, so that they can be safely removed without damaging other parts of the eye.

1.4 Tools and Notation

One of the tools we will be using is called ISCC, which stands for Integer Set Counting Calculator. ISCC is used for doing calculations on possibly infinite sets on integer tuples, and is useful for determining if a certain transformation of an algorithm is legal. In other words, when we try to optimize a program, we want to make it better in terms of memory storage, usage of variables, or the order of operations being computed. For programs in its domain, ISCC allows us to analyze our newly written programs so that we can know if our new version is equivalent to the original program and, where it matters, performs the same computations in the correct order.

ISCC uses the following notation (modeled off examples found in Dave Wonnacott’s ISCC example [Won17]):

1. Tuples are represented by square brackets around a set of integers, so \([1]\) denotes a tuple containing the integer 1, and \{ \[1\] \} is the set containing that 1. Each number inside the tuple represents the statement number in the program. Loops in this case count as one statement, though statements inside the same loop are their own separate statements.

2. We set variables like this: \(x := \{ [1] \};\)
   Then, the variable \(x\) is defined as the first statement in the program.

3. Inside the set, we can have 'such that' statements. For example, consider the following code:

   \[
   \begin{align*}
   0 & \text{ int } x = \ldots \\
   1 & \text{ int } y = \ldots \\
   2 & z \times = \ldots \\
   3 & y = \ldots \\
   \end{align*}
   \]

   In this case, \{ \[i\] : 0 \(<=\) \(i\) < 3 \} would denote the first three statements of the program, or lines 0, 1 and 2, since we would be looking at all statements of the program such that we were at the 0th, 1st or 2nd statement.
4. Now consider the following simple loop, which contains two statements and runs three times:

```java
for (int i = 0; i < 3; i++) {
    a = ... 
    b = ... 
}
```

With ISCC, we can denote the iterations of each of these two operations, where we set the variables a and b. Assuming the for loop is the first statement that appears in the program before any other executions, we denote the setting of a in line 1 to be:

```
first := \{ [1, i, 1] : 0 <= i < 3 \} (the first statement in the program, which will run up to 3 instances, and it’s the first statement in the for loop),
```

and the setting of b in line 2 as:

```
second := \{ [1, i, 2] : 0 <= i < 3 \} (same as first except it’s the second statement in the loop).
```

5. However, the number of times a loop executes might be a variable. From the previous example in number 4, if we changed the loop bound from a 3 to an n, we denote the iterations of the two statements as

```
first := [n] -> \{ [1, i, 1] : 0 <= i < n \} and 
second := [n] -> \{ [1, i, 1] : 0 <= i < n \}.
```

Note that we can add parameters that are more than just n. For example, we could write something like this (&& means 'and', similar to the syntax of many programming languages):

```
first := [m, n] -> \{ [1, i, 1] : i > m && i < n \}
```

2 Literature Review

In this section, we discuss a few methods of optimizing programs. A classic approach to optimizing code focuses solely on manipulating scalar variables and statements rather than arrays and iteration. Classic optimization includes optimization by reducing the amount of work, as explained in 2.1, and optimization by changing the order of the operations in the program, as described in 2.2. In Section 2.3, we introduce a tool called PluTo, which works with C code and attempts to make a program run faster by changing the order of iterations of loop nest(s) within the program. Finally, Section 2.4 discusses iterator-based optimization, which has to do with using iterators to manipulate the iterations of a given program.

After a discussion of optimization method types, we talk about the Deriche program itself and look at an example of using ISCC in a reaching definition analysis.

2.1 Optimization By Reducing the Amount of Work

As mentioned above, classic optimization works with reducing the number of computations a program does when it compiles, and manipulating the order of the execution of each line and iteration in the code so that it is as efficient as possible. As an example of reducing the amount of work, consider the following code, assuming that a, b, c and d are all valid integers:

```java
if (b == 0) {
    c = (a * 3) + 5;
    d = ((a * 3) + 5) / 2;
}
a = 20 - ((a * 3) + 5);
```

As we can see, in the worst case scenario where b is equal to 0, the computation of \((a \times 3) + 5\) is being computed 3 times (which is 6 operations, since \((a \times 3) + 5\) requires two operations). A much more efficient way to write this program would be to set a variable to this computation in the beginning, and then call upon the variable accordingly, shown below:
comp = (a * 3) + 5;
if (b == 0) {
    c = comp;
    d = comp / 2;
}
a = 20 - comp;

Now, comp is being set once, and the actual calculation only happens once in the program. So by writing this new program, we have saved 4 computations, in the case where b == 0. This is a very simple example of classic optimization and it is not difficult to ascertain that it is best to write using comp. One might wonder how one could not notice something like this. However, we must recognize that in long, complicated programs that have several different variables and calculations, finding these ways to make a program more efficient might not be as obvious. Furthermore, the difference is more significant for larger programs that use lots of data and memory.

2.2 Optimization by Changing the Order of Operations

The order of operations in a program is also extremely important when looking at how to speed up the program. As an example, consider a scenario where you are given the task of reading two different novels, and you want to have them read in as little time as possible. It would be extremely inefficient to read one page of the first novel, one page of the second novel, the next page of the first novel, and so on. This would not only likely cause confusion, because you would have to remember the context of where you left off every time you switched back and forth for each book, but it would also require that you picked up the first book, read the page, and put it down before picking up the second book and doing the same. Obviously, you would prefer to pick up the first book, read it the whole way through, put it down, and then pick up the second book, read it all, and put it down. That way, you wouldn’t waste time switching between books or remembering where you left off in each book.

A similar analysis can be done on programs and the order of operations. If we were to create a variable and then only use it after several other operations, it is a slower method than creating the variable right before immediately using it. This becomes increasingly important as we think about memory layout in a computer. Recall that there are several layers to memory in a computer, including registers, cache, main memory, and disk memory. Registers are where values are easily and directly accessed, and as we go further down the list, it takes longer and longer to access or store a value in memory. Furthermore, it is important to recognize that the longer it has been since we used a variable, the further down it will get pushed into memory. So, ideally, we would minimize the time taken between the usage of each value.

Thus, if we had a limited cache size and could only hold a very small number of variables at a time, we would care a lot about when we use particular variables. Another potential problem which often arises is when our program uses a large array, for which we have to be think carefully about how we use and store these arrays with a limited cache size. If we had an array that was bigger than cache, we would be forced to use larger parts of the memory system if we didn’t access each part of the array in a clever way. Since each layer of memory requires more time to access, this would make the program run slower overall. Even though a modern computer does in fact have lots of space in cache memory, we must remind ourselves that we are looking to speed up complex programs that use a large number of variables.

Now let us return to thinking about when we use a variable after declaring it for the first time. In the case of defining a variable and using it later on in the program, we would have to create the variable, store it in memory somewhere, and then later bring it back to use it. In the case where we immediately use a variable after defining it, we don’t have to worry about bringing it back from memory. Clearly, this method is much more desirable when thinking about efficiency and speed. Every operation we can save will count when trying to speed things up.

As an example, consider the following simple program, which initially defines four integer variables a, b, c and d. Let’s pretend that there are only three registers available to use at a time; thus, we cannot easily access four different variables at the same time without going back and forth from memory.

0 int a = 3;
1 int b = 8;
2 int c = 5;
3 int d = 10;
Notice how we set the integer \( c \) in line 2 before using it at line 6. In between lines 2 and 6, we will have to use all three registers to hold the values of \( a, b, \) and \( c \). Thus, once we set \( d \) in line 3, in the best case scenario that minimizes the number of steps taken, the program chooses to replace the register containing \( c \) to temporarily hold \( d \). In a worse case, it will replace the register containing the value of \( a \) or \( b \), which will then have to be brought back eventually for the computation in lines 4 and 5. Then for line 6, \( c \) will be brought back into an arbitrary register and will be updated accordingly.

To optimize this code, we can take advantage of the fact that \( c \) is not used in lines 4 and 5. We ask ourselves, why not set \( c \) right before its usage if we only have space for three registers? Take a look at the following code which moves the definition of \( c \) right before the usage of \( c \):

```
0 int a = 3;
1 int b = 8;
2 int d = 10;
3 b = a + 3;
4 d = (a * b) - 2;
5 int c = 5;
6 c = c * 4;
```

Now, the three registers will be taken up by \( a, b, \) and \( d \). In lines 3 and 4, there will be no need to move anything from memory storage since all the values being used are already held in the current registers. Note that we were only able to do this since \( c \) was not used in lines 3 or 4; otherwise, we would have had to define \( c \) earlier, and so moving the definition statement would have been impossible.

The optimization techniques discussed in Section 2.1 and 2.2 are examples of classic optimization. We will now discuss optimization using PluTo, and iterator-based optimization.

### 2.3 Optimization Using PluTo

PluTo was the first successful program written that performs iteration space transformation automatically. It uses the same mathematics model as ISCC does, called the polyhedral model. While ISCC lets us explore equations that involve integer set variables and check the dataflow of code in the polyhedral model, PluTo automatically uses the polyhedral model while trying to create a faster version of the program in question (and, of course, still producing an equivalent algorithm in the process). In other words, PluTo’s goal is to take in a C program and output a faster C program that performs the same job as the original program. To do this, it focuses on reorganizing the execution of iterations within the code.

PluTo has various settings, including tiling and parallelism (when applying both of these settings, it is best to apply tiling before applying parallelism). Since this is the essence of what we want to do in our analysis of Deriche, we will compare Pluto’s transformed program with the original code in C.

However, before doing so, there are certain limitations to PluTo that we must be aware of:

1. PluTo does not do anything to optimize the usage of variables in a program. Thus, it would not produce a reordering of the algorithm that would, for example, use less variables (which would be more effective for memory usage). This is a huge limitation; we can use the ideas from the example in Section 2.2 to understand that getting rid of even one variable could significantly speed up a program.

2. If there are expressions in the subscripts and loop bounds that are not affine, then ISCC cannot handle it and thinks that the transformation is invalid when there is a possibility that it might be valid. (Recall: An affine expression is a linear expression including a constant in the equation. In the affine expression \( Y = AX + B \), \( X \) and \( Y \) are vectors or numbers, \( A \) is a matrix or number, and \( B \) is a constant.)

3. Even if an expression is affine, it still might be too complicated to evaluate. This causes the complexity of PluTo’s algorithm to increase, and thus PluTo would run for an extremely long time.
4. Even if an expression is affine and not overly complex, PluTo just might not be able to find a way to make the program faster. If this happens, it either means that it is not possible to make the program faster, or that PluTo has failed to find a way to make it faster.

With these limitations in mind, we look at iterator-based optimization.

2.4 Iterator-based Optimization

Iterator-based optimization refers to changing the iterations of a program by using iterators. Before applying these techniques, it is important to figure out which language is best to rewrite the program. Our example of the Deriche Recursive Filtering Algorithm is written in C. We could rewrite it in C, but C is not known to have good iterator notation. Also, rewriting a program in the same original language can prove challenging in its own ways. It is much easier to use a different language and start from scratch, rather than taking an existing program and altering it. In the latter method, there is a bigger chance of breaking the program, which would most likely result in spending lots of unnecessary time figuring out why it broke.

Thus, we choose not to use C. We could use Python, which does have good iterator notation, but in comparison with other languages, Python is a slow language to use. So, we turn to a language called Chapel, has a fast compiler and also has clear iterator notation, allowing the optimizer to accurately do its job.

For our work, an iterator is a loop nest that can run and return multiple times. The most general version of an iterator is called a generator, which is able to return something from a program (in Chapel, these are \texttt{yield} statements), but can still continue to do work after it returns. This will be helpful for our analysis of the Deriche Recursive Filtering algorithm. As stated before, we will use iterators in Chapel to see if we can potentially speed up the performance of the algorithm.

Using iterators is particularly useful when we know that certain loops will not run for all instances. Iterators would allow us to skip the instances for which we know the loop would not run, and this would help the program run faster overall. Also, when we want to change the order of operations and commands in a program to make it faster, we are primarily focused on the iteration process of the program rather than the actual code, so experimenting with different iterators is more interesting than rewriting the code in a different way for our analysis.

2.5 Deriche Recursive Algorithm

The Deriche Recursive Filtering Algorithm applies edge detection and smoothing to 2D images. From the PolyBench code that we will be examining, the inputs of the algorithm are an image, several parameters for the size of the convolution, and several coefficients that are related to the filter the user desires. The single output is the processed image. The PolyBench implementation sets some of these input parameters as defaults. For an input size $n$, the algorithm has a total of $32n^2$ operations, which has order $O(n^2)$ complexity, and it uses $4n^2$ memory spaces, which also has order $O(n^2)$ complexity for memory. These numbers only consider the operations of the main algorithm and do not take into account more than one iteration for each loop in the algorithm [YP15].

2.6 Using ISCC with the Deriche Algorithm

To help us understand how we can connect this algorithm and optimization techniques, we will demonstrate an example using the Deriche PolyBench algorithm. We will use ISCC to analyze an array and transformation, similar to the example shown from [Won17] and Collard’s descriptions in chapter 5 [Col03]. Now, consider the following subset of the code from lines 9-19 in Appendix A.1 that defines and uses $y_{m1}$:

```c
for (i = 0; i < PB_W; i++) {
    ym1 = SCALAR_VAL(0.0);
    for (j = 0; j < PB_H; j++) {
        y1[i][j] = a1*imgIn[i][j] + a2*xm1 + b1*ym1 + b2*ym2;
        ym2 = ym1;
        ym1 = y1[i][j];
    }
}
```
When it comes to optimization techniques, we might be interested in how the definition of variables are being passed from one line to another. Consider the usage of the variable \( y_{m1} \). Notice how it is being defined as 0.0 in line 2, accessed in lines 4 and 5, and redefined as \( y_{1[i][j]} \) in line 6. We want to know which version or instance of \( y_{m1} \) is being called upon in line 5.

In the classic optimization approach, we only care about the possible values that \( y_{m1} \) could be. So, for line 5, the definition could possibly come from either the scalar value 0.0 in line 2, or from the setting of \( y_{m1} \) in line 6. We do not care for which instances each of these possibilities come from.

However, in the instance-wise approach, we want to know more specifically when each possibility happens and for what instances of the loop. In this case, \( y_{m1} \) comes from the setting of 0.0 when \( j = 0 \), and it comes from \( y_{m1} = y_{1[i][j]} \) when \( j > 0 \) and \( j < _PB_H \).

Also, notice how the definition of \( y_{m1} \) is run \(_PB_W\) times, right before each run through of the second for loop. Lines 4, 5 and 6, which use and define \( y_{m1} \), will be run a total of \(_PB_W \cdot _PB_H\) times. Let’s use ISCC to denote this in more detail.

\[
\begin{align*}
\text{line2} & := [\_PB_W] \rightarrow \{ [1, i, 0, 0, 0] : 0 <= i < \_PB_W \}; \\
\text{line4} & := [\_PB_W, \_PB_H] \rightarrow \{ [1, i, 1, j, 0] : 0 <= i < \_PB_W \&\& 0 <= j < \_PB_H \}; \\
\text{line5} & := [\_PB_W, \_PB_H] \rightarrow \{ [1, i, 1, j, 1] : 0 <= i < \_PB_W \&\& 0 <= j < \_PB_H \}; \\
\text{line6} & := [\_PB_W, \_PB_H] \rightarrow \{ [1, i, 1, j, 2] : 0 <= i < \_PB_W \&\& 0 <= j < \_PB_H \}; \\
\end{align*}
\]

Now let’s consider line 5, and find a way using ISCC to denote which definition of \( y_{m1} \) reaches the usage of \( y_{m1} \) in line 5. We know that it is possible for the definition in line 6 as well as the definition in line 2 to reach the usage in line 5, but we want to be more specific about these exact conditions. Similarly to the example in [Won17], for the definition in line 6 or in line 2 to reach the usage of \( y_{m1} \) in line 5, we need the following three conditions:

1. Both the definition and usage of \( y_{m1} \) need to be executed. So in this particular case, the definition must come from line 6 or line 2, and the usage must occur in line 5.
2. The definition and usage of \( y_{m1} \) must both use the same variable. Again, like in [Won17], this is easy, because we know we are using \( y_{m1} \) in both cases.
3. \( y_{m1} \) must be defined before it is used.

Now let’s put these words into ISCC. For the first point, we will denote all of the possible pairings from the definition to the usage in question. Looking first at line 6, we will apply the cross product of line 6 and line 5, since the definition from line 6 goes to line 5 to be used. Similarly, for the definition in line 2 to reach line 5, we use the cross product of those two lines. (Note here that we must put \texttt{unwrap} only to get rid of unnecessary parentheses and brackets that would be annoying to deal with later if we did not remove them.)

\[
\begin{align*}
\text{unwrap (line6 cross line5)}; \\
\text{unwrap (line2 cross line5)};
\end{align*}
\]

The results of the two lines above give the following:

\[
\begin{align*}
[_{PB_W}, _{PB_H}] & \rightarrow \{ [1, i, 1, j, 2] \rightarrow [1, i', 1, j', 1] : 0 <= i < _{PB_W} \&\& 0 <= j < _{PB_H} \}; \\
[_{PB_W}, _{PB_H}] & \rightarrow \{ [1, i, 0, 0, 0] \rightarrow [1, i', 1, j, 1] : 0 <= i < _{PB_W} \&\& 0 <= j < _{PB_H} \};
\end{align*}
\]

We do not need to do any more work to denote the second point, since we already know we are dealing with the same variable throughout this whole analysis.

For the third point, we will add a variable \texttt{FwdInTime} that forces the next statement number of any loop or instance to be greater than the previous operation. In other words, we disallow any order of the iterations that would be considered going back in time to a loop that should have happened earlier.

\[
\begin{align*}
\text{FwdInTime} & := [\_PB_W, \_PB_H] \rightarrow \{ [h, i, j, k, l] \rightarrow [h', i', j', k', l'] : \\
& (h < h' \text{ or } (h=h' \text{ and } i<i')) \text{ or } (h=h' \text{ and } i=i' \text{ and } j<j') \text{ or } (h=h' \text{ and } i=i' \text{ and } j=j' \text{ and } k<k') \text{ or } (h=h' \text{ and } i=i' \text{ and } j<j' \text{ and } k=k' \text{ and } l<l'))};
\end{align*}
\]
However, this alone does not protect against iterations that skip unnecessarily through iterations; for example, \([1, 3, 0, 0, 0] \rightarrow [1, 3, 1, 1, 1]\) should be illegal, since the usage of \(ym1\) in line 5 in this case should come from \([1, 3, 1, 0, 2]\) which is the setting of \(ym1\) in the previous inner loop iteration. In other words, the setting of \(ym1\) to 0.0 should not reach any usage of \(ym1\) when \(j > 0\).

So we will create a rule preventing these skips. We will create a few variables containing the set of these unwanted skips, which we will subtract or remove from the set of all statements reaching the usage of \(ym1\) in line 5. The variables will demonstrate two writes to \(ym1\) before it reads \(ym1\). There are three cases of these unwanted skips that we want to take care of:

1. Something like \([1, i, 0, 0, 0] \rightarrow [1, i, 1, 1, 1]\) should be illegal. This is the case we just described above.

2. Something like \([1, i, 1, j, 2] \rightarrow [1, i, 1, j + 2, 1]\) should be illegal. This involves writing to \(ym1\) in line 6, writing to it again in line 6 of the next iteration, and then reading from line 5 in the next iteration after that.

3. Something like \([1, i, 1, _PB_H - 1, 2] \rightarrow [1, i + 1, 1, 0, 1]\) should be illegal. This involves writing to \(ym1\) in line 6, then writing to it again in line 2 of a new outer loop iteration, then reading in line 5 for the first inner loop iteration of the new outer loop iteration.

Thus we write the following (note that . is called a join operation, which is similar to the mathematical function composition in that it performs one action and then another action, and \(\ast\) means take the intersection, or 'and' in computer science terms):

\[
\text{Write2ToWrite6} := \text{unwrap} \left( \text{line2 cross line6} \right);
\text{Write6ToRead5} := \text{unwrap} \left( \text{line6 cros line5} \right);
\text{Write2Write6Read5} := \text{Write2ToWrite6} \ast \text{FwdInTime} \ast \text{Write6ToRead5} \ast \text{FwdInTime};
\]

\[
\text{Write6Write6Read5} := \text{Write6ToWrite6} \ast \text{FwdInTime} \ast \text{Write6ToRead5} \ast \text{FwdInTime};
\]

\[
\text{Write6Write2Read5} := \text{Write6ToWrite2} \ast \text{FwdInTime} \ast \text{Write2ToRead5} \ast \text{FwdInTime};
\]

We must then subtract these from our original reaching definitions of line 5 coming from line 2 and line 6, but before we do this we must intersect the original reaching definitions with \(\text{FwdInTime}\) as well:

\[
\text{reaching}_6_5 := \text{unwrap} \left( \text{line6 cross line5} \right) \ast \text{FwdInTime};
\text{reaching}_2_5 := \text{unwrap} \left( \text{line2 cross line5} \right) \ast \text{FwdInTime};
\]

Then for each reaching statement we will subtract the unwanted skips:

\[
\text{reaching}_6_5\text{valid} := \text{reaching}_6_5 - \text{Write6Write6Read5} - \text{Write6Write2Read5};
\text{reaching}_2_5\text{valid} := \text{reaching}_2_5 - \text{Write2Write6Read5};
\]

Finally, we union these two calculations together to get the final answer:

\[
\text{reachingFromBoth} := \text{reaching}_6_5\text{valid} + \text{reaching}_2_5\text{valid};
\text{reaching}_5 := \text{reaching}_6_5\text{valid} + \text{reaching}_2_5\text{valid};
\]

When we ask for \(\text{reaching}_5\) we get the following output:

\[
[_\text{PB}_W, _\text{PB}_H] \rightarrow \{ [1, i, 1, j, 2] \rightarrow [1, i' = i, 1, j' = 1 + j, 1] : \\
0 \leq i < _\text{PB}_W \text{ and } 0 \leq j < -2 + _\text{PB}_H; \\
[1, i, 0, j = 0, 0] \rightarrow [1, i' = i, 1, j' = 0, 1] : \\
_\text{PB}_H > 0 \text{ and } 0 \leq i < _\text{PB}_W \}
\]
The first object in the set describes going from the definition of \( ym1 \) in line 6 of iteration \([1, i, 1, j, 2]\) to the usage of line 5 in iteration \([1, i, 1, j + 1, 1]\). It also assumes that the definition of \( ym1 \) is not at the last loop of \( j \) so that we are able to stay in the same loop of \( i \). The second object in the set describes going from the definition of line 2 in iteration \([1, i, 0, 0, 0]\) to the usage of \( ym1 \) in iteration \([1, i, 1, 0, 1]\) (the first time through running through the nested for loop). This matches what we discussed earlier before doing an ISCC analysis.

Thus, we have our set of valid reaching definitions for the variable \( ym1 \) to reach line 5, which as we can see comes from the definition of \( ym1 \) in line 2 when we are in the first run-through of the nested for loop, and it comes from the definition in line 6 otherwise. If we were to rewrite this particular nested for loop, we would want our ISCC code for the reaching definition of \( ym1 \) to be the exact same. Then we would know that our transformation was equivalent to the original code.

3 Background

Currently, the overall goal is to determine whether it is worth trying to speed up the Deriche program. In order to do this, we will do several analyses. First we will run the original PolyBench C code for several different sized datasets and look at the graph of running time vs. total number of operations. If the graph seems linear, then it would not be very interesting to try to optimize the code on the computer we are testing the code on specifically for saving memory usage. If it’s not linear, then we say it is worth trying to speed it up by converting it to Chapel and changing the iteration of the code. Whether or not the original graph is linear, we still think it’s worth checking to see how PlutoTo attempts to make it faster.

3.1 An Idea for Changing the Order of Operations

After a closer look at the Deriche PolyBench code, we hypothesized that the statement on line 37 of Appendix A.1, which uses \( y2[i][j] \), can be placed right underneath line 27, which defines \( y2[i][j] \), and the program should run the same way. Similarly, we can also put line 68 right after line 58. This would get rid of the two for loops containing the usage of \( y2[i][j] \). This is beneficial because right now the original code must access every location in \( y2 \) twice, which takes more work than accessing it once, using it right away and never using it ever again. So we hope that this will make the program run faster since there will be less time spent accessing memory.

3.2 Checking Correctness using ISCC

First, we must make sure that this does not in fact change the program by using ISCC to help us. We will first analyze the original code, shown below (Note that we have removed statements that are not necessary in this particular analysis for simplicity. Also, we will only analyze one of the loops described since doing both would be essentially the same analysis.):

```c
for (i=0; i< _PB_W; i++) {
  for (j= _PB_H−1; j>=0; j−−) {
    y2[i][j] = a3*xp1 + a4*xp2 + b1*yp1 + b2*yp2;
  }
}
for (i=0; i< _PB_W; i++) {
  for (j=0; j< _PB_H; j++) {
    imgOut[i][j] = c1 * (y1[i][j] + y2[i][j]);
  }
}
```

To start off with our translation into ISCC, let’s create two variables, one where we write to the array \( y2 \) and another where we call for a memory space in that array:

```csharp
write_y2 := [_PB_W, _PB_H] -> { [i, 1, 1, j, 0] :
  0 <= i < _PB_W && 0 <= j <= _PB_H - 1 };
read_y2 := [_PB_W, _PB_H] -> { [2, i, 1, j, 0] :
  0 <= i < _PB_W && 0 <= j < _PB_H };
```

We will then create our \texttt{FwdInTime} variable, which will be the exact same as that of the example from Section 2.6, since the number of dimensions is the same.
In the example from Section 2.6, we subtracted out things we didn’t want from all possible reaching definitions of a variable. In this case, it is easier to intersect our set with things we know we want to be true. With this in mind, we know from looking at the two essentially identical for loops that the usage of a particular memory spot in $y_2$ must come from a previous definition where $i$ and $j$ were the exact same, since both use $y_2[i][j]$. So, let’s create a variable demonstrating this:

```plaintext
#adding in wanted cases where i=i' and j=j', to intersect later
same_memory := [_PB_W, _PB_H] -> { [x, i, y, j, z] -> [x', i', y', j', z'] :
  (i=i' and j=j') };
```

Finally, we will intersect our reaching definition set with $FwdInTime$ and $same_memory$. This will give us the iterations going from the write to the read of $y_2$ that do not go backwards to a previous iteration and that also only include cases with the same memory cell ($i$, $j$).

```plaintext
#all forward in time iterations from the write to the read of y2
WriteThenRead := (unwrap (write_y2 cross read_y2) * FwdInTime);

#intersecting with cases where i=i' and j=j'
reaching_read_y2 := WriteThenRead * same_memory;
```

Our variable $reaching_read_y2$ gives

```plaintext
[_PB_W, _PB_H] -> { [1, i, 1, j, 0] -> [2, i' = i, 1, j' = j, 0] :
  0 <= i < _PB_W && 0 <= j <= _PB_H - 1 };
```

Now let’s consider the following amendment to the code in question, which converts $y_2$ to a scalar rather than an array:

```plaintext
0 for (i=0; i<_PB_W; i++) {
  1 for (j=PB_H-1; j>=0; j--) {
  2 $y_2 = a_3*_{x}p1 + a_4*_{x}p2 + b_1*_{y}p1 + b_2*_{y}p2;
  3 imgOut[i][j] = c_1 * (y_1[i][j] + y_2);
  4 }
}
```

We will create the same variables $write_y2$ and $read_y2$, with the minor difference showing that the two statements are now within the same first statement of the program. Notice how the setup shows that $write_y2$ is the first statement of the inner loop (the third parameter is a 1), while $read_y2$ is the second statement of that same inner loop.

```plaintext
write_y2 := [_PB_W, _PB_H] -> { [1, i, 1, j, 0] :
  0 <= i < _PB_W && 0 <= j <= _PB_H - 1 };
read_y2 := [_PB_W, _PB_H] -> { [1, i, 1, j, 1] :
  0 <= i < _PB_W && 0 <= j < _PB_H };
```

We will also create the same $FwdInTime$ variable. To mirror the process from the original code, we will use the same $same_memory$ variable from before, but note that $same_memory$ will always be true since both write and read are in the same exact loop, and thus it would be impossible for $i$ and $j$ not to be the same for each write and read. Thus, intersecting $same_memory$ with other things will not affect our final result, but we will include it to parallel the ISCC analysis with the original code before our implemented transformation.

```plaintext
#adding in wanted cases where i=i' and j=j', to intersect later
same_memory := [_PB_W, _PB_H] -> { [x, i, y, j, z] -> [x', i', y', j', z'] :
  (i=i' and j=j') };
```

Now that the two statements are in the same loop, we have to remove the possibility of writing $y_2$ twice before reading it, similar to the example shown in Section 2.6. In other words, we have no rule preventing the read in line 3 of iteration $j$ in the inner loop coming from the write in line 2 from iteration $j-1$. This still counts as a write happening before a write by our $FwdInTime$ variable, but since all reads should come from the write in the same iteration of the inner loop, this situation is undesirable. It is also worth noting that this was not an issue in the original code.
Because the two statements were separated by two different loops, all the writes happen before all the reads, so it would be impossible to skip over a write using our rules.

Thus we will create variables to demonstrate what we have just discussed. The variable \texttt{WriteWrite} represents a write from one iteration to a write in the following iteration of the inner loop, and \texttt{WriteRead} represents going from a write to a read of \texttt{y2}. Finally, \texttt{WriteWriteRead} joins these together, creating our unwanted case that we will subtract from all reaching definitions in the end.

\begin{verbatim}
#write then write then read - will subtract this later
WriteWrite := (unwrap (write_y2 cross write_y2)) * FwdInTime;
WriteRead := (unwrap (write_y2 cross read_y2)) * FwdInTime;
WriteWriteRead := WriteWrite . WriteRead;
\end{verbatim}

Now we will compute all 'forward in time' iterations from a write of \texttt{y2} to a read of \texttt{y2}, and then intersect this result with \texttt{same_memory}.

\begin{verbatim}
#all forward in time iterations from the write to the read
WriteThenRead := (unwrap (write_y2 cross read_y2) * FwdInTime);

#intersecting with cases where i=i' and j=j' (will not change anything)
WriteReadSameMemory := WriteThenRead * same_memory;
\end{verbatim}

Finally, we subtract out \texttt{WriteWriteRead}, as discussed above:

\begin{verbatim}
#take out write write read cases
reaching_read_y2 := WriteReadSameMemory - WriteWriteRead;
\end{verbatim}

This time, \texttt{reaching_read_y2} gives us

\begin{verbatim}
[_PB_W, _PB_H] -> \{ [1, i, 1, j, 0] -> [1, i' = i, 1, j' = j, 1] :
  0 <= i < _PB_W and 0 <= j < _PB_H \}
\end{verbatim}

Note how this is the exact same result we got when running ISCC with the original C code, except for one minor difference. While \texttt{reaching_read_y2} for the analysis of the original code took statements from the first to the second statement of the program, \texttt{reaching_read_y2} for the transformed code takes statements from the first statement to the second statement in the inner loop. It is important to understand that these represent the same statements in each program and that they are simply labeled differently in ISCC since we altered their positions. Thus, we can see that there is no change of reaching definitions of the variable \texttt{y2} when transforming this piece of the code in this way. We can then conclude that making this change does not affect the correctness of the algorithm.

Notice also that we can also remove the entire variable of \texttt{y2} by doing the following:

\begin{verbatim}
0 for (i=0; i<_PB_W; i++) {
  1 for (j=PB_H-1; j>=0; j--) {
    2 imgOut[i][j] = c1 * (y1[i][j] + a3*xp1 + a4*xp2 + b1*yp1 + b2*yp2);
    3 }
  4 }
\end{verbatim}

When comparing running times with the original code, we will use both examples of moving statements and removing the usage of \texttt{y2} altogether.

### 3.3 Example of an Incorrect Transformation

Let us demonstrate how ISCC tells us when we have altered the code in such a way that the program no longer produces the same output as the original program. Consider the following change to the original loop we were studying in Section 3.2:

\begin{verbatim}
0 for (i=0; i<_PB_W; i++) {
  1 for (j=PB_H-1; j>=0; j--) {
    2 y2[i][j] = a3*xp1 + a4*xp2 + b1*yp1 + b2*yp2;
    3 imgOut[i][j] = c1 * (y1[i][j] + y2[i][PB_H-j]);
  4 }
  5 }
\end{verbatim}
If line 3 involved \( y_2[i][j] \), the transformation would be correct, but instead we have \( y_2[i][\_PB_H - j] \), which should produce an algorithm with a different result than the original Deriche program. Let’s see how ISCC handles this.

So, we create our variables `write_y2` and `read_y2`:

```plaintext
write_y2 := [\_PB_W, \_PB_H] -> { [1, i, 1, j, 0] : 
    0 <= i < \_PB_W && 0 <= j <= \_PB_H - 1 };
read_y2 := [\_PB_W, \_PB_H] -> { [1, i, 1, j, 1] : 
    0 <= i < \_PB_W && 0 <= j < \_PB_H };
```

Also, we create our `FwdInTime` variable. This time, our `same_memory` variable changes:

```plaintext
# adding in wanted cases where i = i' and j = \_PB_H - j, to intersect later
same_memory := [\_PB_W, \_PB_H] -> { [x, i, y, j, z] -> [x', i', y', j', z'] :
    (i = i' and j = \_PB_H - j) };
```

Our variables `WriteThenRead` and `reaching_read_y2` remain the same, referring to the variables we just created.

```plaintext
# all forward in time iterations from the write to the read of y2
WriteThenRead := (unwrap (write_y2 cross read_y2) * FwdInTime);

# intersecting with cases where i=i' and j=j'
reaching_read_y2 := WriteThenRead * same_memory;
```

This time, `reaching_read_y2` gives something entirely different from `reaching_read_y2` for the original code:

```plaintext
[\_PB_W, \_PB_H] -> { [1, i, 1, j, 0] -> [1, i = i, 1, j, 1] :
    2j = \_PB_H and 2j = \_PB_H and \_PB_H >= 2 and 0 <= i < \_PB_W;
    [1, i, 1, j, 0] -> [1, i = i, 1, j = \_PB_H - j, 1] :
    0 <= i < \_PB_W and 0 < j < \_PB_H and 2j < \_PB_H }
```

Since this is clearly not the same as what we get for the original code, we can conclude that this transformation is not equivalent to the snippet of code from the original Deriche program.

### 4 Results From Background Discussion

Table 1 contains selected data points showing the results of running the original Deriche C code with different size inputs for the image. The first column is the height and width of the dataset used, and the second column calculates the number of operations for the dataset. The equation used to calculate the number of operations `op`, for an image of height `\_PB_H` and width `\_PB_W`, is

\[ \text{op} = (18 \times \_PB_H) + (14 \times \_PB_W) + 16. \]

For each of the tables in this section, we ran each dataset type twice and took the average time it took to run in seconds. For these experiments, we used a computer with an Intel® Core™i5-6500 CPU, with a clock speed of 3.20GHz, 6MB of Level 3 cache, and 8GB of RAM. To measure these times, we used the provided and already built-in PolyBench timer, called `polybench_start_instruments`. As stated in the original files, the entire function, including the call and return, were timed [YP15].

<table>
<thead>
<tr>
<th>Dataset size (wxh)</th>
<th>Number of operations</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>2064</td>
<td>0.00018</td>
</tr>
<tr>
<td>192x198</td>
<td>5008</td>
<td>0.0011005</td>
</tr>
<tr>
<td>720x480</td>
<td>18736</td>
<td>0.013465</td>
</tr>
<tr>
<td>1680x1320</td>
<td>47296</td>
<td>0.0857565</td>
</tr>
<tr>
<td>4000x4000</td>
<td>128016</td>
<td>0.5879255</td>
</tr>
<tr>
<td>8000x6600</td>
<td>230816</td>
<td>2.0082745</td>
</tr>
<tr>
<td>9000x7000</td>
<td>252016</td>
<td>2.5415975</td>
</tr>
<tr>
<td>10000x10000</td>
<td>320016</td>
<td>3.7339505</td>
</tr>
</tbody>
</table>

Table 1: Results of running Deriche original PolyBench C code
If we graph the number of operations per time, we can see in Figure 1 that the graph is not linear. If we look closely at the points from 0 to about 10000 operations, the line seems relatively linear, but then after that the slope of the line changes significantly. This suggests that once the dataset gets large enough to make the computer do a total about 10000 operations for the program, the computer runs significantly slower and has a harder time keeping up with the original ratio of operations per second. From previous discussion, we understand that this suggests that it is worth trying to optimize the Deriche algorithm so that we can make this change in slope as little as possible if not remove it entirely.

![Graph of running time vs. number of operations for the original PolyBench Deriche C code](image)

**Figure 1:** Graph of running time vs. number of operations for the original PolyBench Deriche C code

### 4.1 Manual Optimization

In Section 3.1, we introduced some techniques to model possible iterator and data transformation classic optimization. In Section 3.2, we checked for the correctness of our transformation. The runtime results of our new program, with the same dataset sizes that we used to analyze the original code, can be found in Table 2. We tested both the first step of only moving statements without making $y_2$ a scalar, and then looked at the runtimes of both steps where we included moving statements and making $y_2$ a scalar.

The results from simply moving statements is not promising, but those from making $y_2$ a scalar are extremely interesting since for many dataset sizes, the running time decreases a small amount. This makes us believe that it would be worthwhile to attempt optimizing the Deriche Recursive Filtering algorithm using Chapel iterators.

### 4.2 PluTo Optimization

We ran PluTo once with regular default settings and once with the tiling setting. Again, both of these results for the runtime of several dataset sizes can be seen in Table 2.

It seems that PluTo has not successfully run the code any faster than the original C code. In fact, for every dataset, PluTo has performed slightly slower than the original. These prove interesting results, and indicate that it would be worthwhile trying to optimize the code using Chapel iterators.

### 4.3 Remarks

Table 2 puts together all of the running times for the select datasets shown from previous tables, and Figure 2 shows a graph directly comparing the various methods of optimization we have experimented with in this paper. It seems that making $y_2$ a scalar was the most successful in decreasing running time, but the difference in running time was not as much as we had hoped for. In any case, it was interesting to find that PluTo did a worse job than the original code when trying to optimize it, and also exciting nonetheless to see the running time decrease at least some ratio for our own optimization ideas.
<table>
<thead>
<tr>
<th>Dataset size (wxh)</th>
<th>Original</th>
<th>Combine For Loops</th>
<th>y2 Scalar</th>
<th>PluTo Normal</th>
<th>PluTo Tiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.00018</td>
<td>0.000215</td>
<td>0.00019</td>
<td>0.000196</td>
<td>0.000204</td>
</tr>
<tr>
<td>192x198</td>
<td>0.0011005</td>
<td>0.0010355</td>
<td>0.0009685</td>
<td>0.0011325</td>
<td>0.001151</td>
</tr>
<tr>
<td>720x480</td>
<td>0.013465</td>
<td>0.015573</td>
<td>0.0148365</td>
<td>0.014165</td>
<td>0.0157835</td>
</tr>
<tr>
<td>1680x1320</td>
<td>0.085765</td>
<td>0.094865</td>
<td>0.0834195</td>
<td>0.0862555</td>
<td>0.0873345</td>
</tr>
<tr>
<td>4000x4000</td>
<td>0.5879255</td>
<td>0.6515115</td>
<td>0.5760455</td>
<td>0.597593</td>
<td>0.5974995</td>
</tr>
<tr>
<td>8000x6600</td>
<td>2.0082745</td>
<td>2.219946</td>
<td>1.939622</td>
<td>2.3767915</td>
<td>2.063067</td>
</tr>
<tr>
<td>9000x7000</td>
<td>2.5415975</td>
<td>2.8977605</td>
<td>2.5640735</td>
<td>3.001914</td>
<td>2.593763</td>
</tr>
<tr>
<td>10000x10000</td>
<td>3.7339505</td>
<td>4.198566</td>
<td>3.6484805</td>
<td>5.190377</td>
<td>3.8323145</td>
</tr>
</tbody>
</table>

Table 2: All running time results

![All Methods](image)

Figure 2: All the optimization methods put into one graph

From these results, we believe that the Deriche Recursive Filtering algorithm could potentially use some memory locality optimization. Thus, we have confirmed that converting this program to Chapel and manipulating iterators would be worthwhile. As described previously, if we could prove that the Deriche algorithm in Chapel could run the program significantly faster than the original C code, we would be able to add Deriche to the small but hopefully growing list of algorithms in which you can do iteration-space performance tuning via iterators, and produce results superior to PluTo. So, we now turn to iterator-based optimization for the Deriche Algorithm.

5 Converting to Chapel

The code for Deriche, converted from C to Chapel, can be found under Appendix B.1. As we can see, the code is extremely similar to the original version in C. Below is the first nested for loop, where the first values of \( y_1 \) are being created. The only significant difference between the Chapel code and the original C code (before using iterators) is the syntax.

```chapel
for i in 0..w-1 {
  ym1 = 0.0;
  xm1 = 0.0;
  for j in 0..h-1 {
    y1[i,j] = a1*imgIn[i,j] + a2*xm1 + b1*ym1 + b2*ym2;
    xm1 = imgIn[i,j];
    ym2 = ym1;
    ym1 = y1[i,j];
  }
}
```
5.1 Applying Iterators to the Deriche Chapel Version

We now look at different iteration-based optimization techniques in hopes to improve the performance of the Deriche algorithm. We first create a simple version of the original code, only this time adding iterators. Since each for loop runs in a slightly different traversal order, we are unable to generalize the order of all the nested for loops and thus will use one iterator for every block of code. The following is an example of the first iterator used for the first nested for loop, along with the actual block of code that uses it:

```c
iter ij_forwards(w: int , h: int) : (int, int, int) {
    for i in 0..w-1 {
        yield (i, 0, -9999); // -9999 (j) will be ignored, since are not concerned with the j value here yet
        for j in 0..h-1 {
            yield (i, 1, j);
        }
    }

    for (i, statement, j) in ij_forwards(w, h) {
        if (statement == 0) {
            ym1 = 0.0;
            ym2 = 0.0;
            xm1 = 0.0;
        } else { // here, statement should be 1
            y1[i, j] = a1*imgIn[i, j] + a2*xm1 + b1*ym1 + b2*ym2;
            xm1 = imgIn[i, j];
            ym2 = ym1;
            ym1 = y1[i, j];
        }
    }
}
```

This translation as a whole can be found in Appendix B.2.

Notice that we put each iterator right above each block of code for which we want it to yield. We only do this for the first example using iterators, convenience and better understanding. In later examples, we will put all the iteration in one giant iterator, and follow with the code that uses that one iterator.

Also note that we have added a dimension in each iterator called `statement`. For the 1st, 2nd, 4th and 5th blocks of code that create arrays `y1` and `y2`, the variable `statement` allows us to distinguish between whether we are in a line of code within the first part of each for loop, where variables typically are being set to 0.0 (like in lines 1-3 in the code shown at the beginning of Section 5), or the completely nested part of the for loop, where either the array `y1` or the array `y2` is being created (like in lines 6-8 in the code shown at the beginning of Section 5).

This simple version of Deriche with iterators is a direct translation of the original order of iteration, so the runtime should not change. Our current end goal is to manipulate these iterators and change the order to help improve performance. Before we can do this, however, it will be easier to have a giant general array to keep track of where we are in any point of the program. Thus, we will have a yield statement for every line of code, which we can see in Appendix B.3. This setup allows us to put everything into one giant iterator for the entire program, which will make it possible to manipulate the order of iteration for the whole algorithm rather than just for each loop.

5.2 Using Iterators to Fuse Loops

If we take a closer look at the code as a whole (see the appendices), we may notice that there seem to be two main pieces to the algorithm. The first involves the first three nested for loops, and the second involves the next three nested for loops. Note how the first section takes each value of the original input array, `imgIn`, and creates two new arrays, `y1` and `y2`, based off the values of the original array.

In order to create `y1`, in the first nested for loop, the code traverses through the original image in a forwards motion for both the width and the height. On the other hand, to create `y2` it travels
Forwards for the width and backwards for the height, which we can see in the second nested for loop. Finally, in the third nested for loop, we combine each pixel of these two arrays to create \texttt{imgOut}.

The next three nested for loops follow an extremely similar process; this time, the 'input' array is \texttt{imgOut}, which we have just created from the previous three for loops. Then, \texttt{y1} and \texttt{y2} are used again to create transformed versions of our first created version of \texttt{imgOut}. Again, the traversal through \texttt{imgOut} varies for each different array in order to get a different transformation. In the final nested for loop of the Deriche code, we can see that the final version of \texttt{imgOut} is created, which serves as the output of the entire program. The diagram in Figure 3 illustrates this entire data flow. Note that the upper left dark green arrow corresponds to the block of code shown in Section 5.1, since it refers to the creation of the first values in the array \texttt{y1}.

With this whole process in mind, we speculated that it is possible to manipulate the iteration of this program so that there are two main loops, as expressed in the diagram in Figure 4. We hope that we can create \texttt{y1} and \texttt{y2} at the same time while traversing through \texttt{imgIn} and the first version of \texttt{imgOut}. Originally, there are two separate loops to write the values of each array \texttt{y1} and \texttt{y2}, and then a final loop to grab the values in each memory cell of those arrays and combine them in order to write to the memory cells of \texttt{imgOut}. This could cause the program to run very slowly, if either the cache is too small or the arrays are too large. If we could not fit all these arrays in cache at the same time, the program would have to take several more steps to relay information between memory and cache. Thus, there is a potential for extremely big run times for either small cache or large arrays.

We propose fusing these loops together, so that we write the same row or column (depending on which part of the program we are on) to \texttt{y1} and \texttt{y2} in the same sequence, before directly placing the transformed version of these values into the final \texttt{imgOut} array (see Figure 5, which articulates the first few steps of iterating over columns, for a visualization of this idea). This lets us forget the original values of \texttt{y1} and \texttt{y2} because we have already sent them to the required third and final array, and we do not need to save them for later. Thus, we hope to save lots of time by avoid having to go back into memory to grab the values we wrote before to bring them into cache. Below, we show the iterator to do this, but the entire version of the code can be found in Appendix B.5.

Note that the only thing we are changing from the first transformation using iterators is the iterator and not the main loop body. As previously discussed, since we already split up the iteration

![Figure 3: Data flow of the original Deriche algorithm](image)

![Figure 4: Data flow of the first version of our transformation of Deriche](image)
part and the computation part of the code, it is much easier to change just the iteration. Also, we can be confident that the original result of the main body will not have changed; so, if the code in the main body was computationally correct before, changing just the iteration using this iterator should not make it incorrect.

```c
0 iterator deriche_iterations(w: int, h: int): (int, int, int) {
1     for i in 0..w-1 {
2         for j in 0..h-1 {
3             yield (0, i, j);
4         }
5         for j in 0..h-1 by -1 {
6             yield (1, i, j);
7         }
8         for j in 0..h-1 {
9             yield (2, i, j);
10        }
11     }
12     for j in 0..h-1 {
13         for i in 0..w-1 {
14             yield (3, i, j);
15         }
16         for i in 0..w-1 by -1 {
17             yield (4, i, j);
18         }
19         for i in 0..w-1 {
20             yield (5, i, j);
21        }
22    }
23 }
```

Keep this idea of turning Deriche into two phases in mind, as we will later apply it to the ideas discussed in Section 5.3, which we now turn to.

### 5.3 A More Concise Way to Express Deriche

Another way that we can improve performance is by altering the semantics and logic of the code. If we look at the original Deriche algorithm, notice that, for all nested for loops except for those that record values into `imgOut`, the Deriche algorithm uses scalar variables as placeholders for the two most recent values that were written to either `y1` or `y2` (depending on which block of code we are in). All these lines of code can seem excessive and may detract from the main purpose of each block of code, which is to simply update the arrays `y1` or `y2`. Hence, we propose an idea for a class in which we do the work of saving the two most recent writes to the array "behind the scenes", thus removing from the main body the lines of code that are not directly related to the main action of the algorithm.
In order to do this, we create a class for these arrays that have the appropriate methods for what we need. This includes setter and getter methods, as well as several other procedures that will help us find the previous two values that were written to the array. Note that, since the iteration of each nested for loop is not the same, we need to write methods for obtaining values both above and below the current value being written, in both a row-wise and column-wise perspective. Hence we name these methods \texttt{jlower}, \texttt{jlowerlower}, \texttt{ilower}, \texttt{ilowerlower}, \texttt{jhigher}, and so on. In each of these methods, we directly call the array. For example, for \texttt{jlower}, if \(j\) is not equal to 0, then we return the array with subscripts \((i, j-1)\). If \(j\) is 0, then we return 0.0.

```c
0 proc jlower(i: int, j: int) {
1   if (j == 0) {
2     return 0.0;
3   }
4   else {
5     return Vals[i, j-1];
6   }
7 }
```

From this class, we can then write a much more logically concise version, which uses fewer statements overall. (Note that we have chosen long names such as \texttt{jlowerlower}, which may not necessarily be semantically concise, as we could have put something like \texttt{jll}, but we wanted to be as clear as possible in our description. In any case, we still are definitely being more concise logically this way.) Each block of code only has one line which is directly related to the focus of the algorithm. Although our new class seems extensive and has several different methods, we would argue that this makes the Deriche code much clearer, which is beneficial for someone unfamiliar with the algorithm who might be looking through the file. This class can be found under Appendix C.2.

To furthermore optimize this class, we can imagine using scalar variables in which the two previous writes were already saved before calling on them. So, we introduce two new scalar variables, \texttt{mostRecentWrite} and \texttt{previousWrite}, to represent the two most recent writes to the array (with \texttt{previousWrite} being the older write, and \texttt{mostRecentWrite} being the newer write). We then alter our set method to update these variables every time we set a new value within the array:

```c
0 proc set(i: int, j: int, value: real) {
1   previousWrite = mostRecentWrite;
2   mostRecentWrite = value;
3   Vals[i,j] = mostRecentWrite;
4 }
```

So, our \texttt{jlower} method looks like the following:

```c
0 proc jlower(i: int, j: int) {
1   return mostRecentWrite;
2 }
```

Note that, in a way, this makes our methods a bit redundant, since several different methods will now return the same exact scalar variables. For example, \texttt{jlower} and \texttt{jhigher} both return \texttt{mostRecentWrite}. The only real difference between these two methods is the initialization and the direction of the iteration when using the methods. But, we hope that it is both easier to understand and also better performance-wise.

Finally, we can apply the idea of iteration in two phases as described in Section 5.2. This not only gives us a more concise way of describing the algorithm in the principal file for Deriche, but also hopefully also improves the runtime of the algorithm. The code, specifically the iterator, for this two-phase idea can be found in Appendix B.7.

After running some tests, we questioned whether or not the \texttt{if} statements within our new class for the concise version of Deriche were hindering the speed of the code. In other words, our class needs to check whether or not we are in the very first iteration of the inner loop, because if we are, then we should be setting scalar variables to 0.0. However, we suspected that having the \texttt{if} statement itself may have been slowing down the overall speed of the program. Thus, we proposed a new method within our class called \texttt{resetScalars()}, shown below:

```c
0 proc resetScalars() {
1   mostRecentWrite = 0;
2   previousWrite = 0;
3 }
```
This method will get called at every `yield` statement in which there is a `-1` (which we will intentionally put in the appropriate places). We add these `yield` statements in the iterator between nested for loops, where normally we would be setting scalar variables to 0.0. Note that using `-1` is arbitrary, and we could have used any other number so long as it was consistent with what we asked for later when determining the `statement` number of the loop. The class we use for this particular example is listed in Appendix C.3, and the code using the class is in Appendix B.6.

We also speculated that our performance was so much worse than what we expected because of our usage of classes rather than records. Note that function calls require several steps that include making several different transitions to and from the stack in order to pass variables through from cache to memory or vice versa. Thus, we turn to something that Chapel has called records. A record in Chapel is similar to a class, but this time whenever a function is being called, the record allows the compiler to use function call inlining. Function call inlining is extremely helpful when we have a function that calls another function within its method. As we know, there is a much higher cost when we make a call to the stack opposed to when we only have to work with registers. Thus, when optimizing code, our goal in this regard is to reduce as much as possible the number of calls to the stack we make. At compile time, at every point in the program where there is a function call, function call inlining saves where to go to get the required elements from the call. This saves time from making a full function call to the stack, since the desired code is already known. Another benefit of using function call inlining is that it could potentially reveal other opportunities for optimization. For example, if there are identical copies of code between the outer function and the inner function, then we could use similar techniques as shown in the example from Section 2.1 to further optimize the code.

In order to do function call inlining, however, we must be aware that there are certain requirements to use it. First, we have to know which function is called, since if we didn’t, we obviously would not know where to go within the code. Second, we need to have access to that function’s body so that we can properly use that code through this process. Once we have these two things in place, then we know it is possible to do function call inlining.

It seems apparent from experiments that lots of time is being spent when we use function calls for classes. Thus, we decide to use records rather than classes, since we are saving ourselves from having to do several unnecessary extra steps. Now, using records, the code does not have to execute the instructions for a function call, such as call or return statements.

Also, as we briefly mentioned at the end of Section 5.2, we will apply the two phases idea on top of all the ideas we have discussed in this section. The final results of each of these performance tuning ideas will be shown later in Section 7.

### 5.4 Using 1-Dimensional Arrays

As a further experiment to using two phases from the concise record described in the previous section, we propose what we hope is the fastest version yet of the Deriche algorithm. Rather than using 2-Dimensional arrays like we have been for \( y_1 \) and \( y_2 \), we will use 1-Dimensional arrays.

It might seem difficult to understand how it is possible to do this when we must deal with information that requires input and output arrays with two dimensions, but if we consider the context from the previous section, it might seem more intuitive. Notice that in our version of Deriche that uses our special "derray" record, along with our iterators that allow us to implement everything in two big phases, we are only looking at one row or column (depending on which of the two big phases we are currently in) at a time. We have already explained that after we iterate through each row or column, we no longer need the data that was used for that row or column.

To help better visualize this, let us return back to Figure 5. Note how each column that we are working with for each iteration is no longer being used after we are done with that particular column; so we do not need to save any values from any given column as soon as we leave that column. So, we can see that we only need to hold one dimension of values at a time; that is, we only focus on one row or column at a time throughout the entire algorithm, given that we are implementing the version of fusing loops to create two phases of the code. Thus, \( y_1 \) and \( y_2 \) will be purely one-dimensional, which we hope will attribute to the fastest version of all our experiments with the Deriche algorithm.
6 Checking for Read/Write Correctness

All of these variants of the algorithm are of course prone to bugs, and it is in our best interest to check and make sure that we are still producing the same algorithm after altering the code. As described before, the goal is to produce a hopefully faster program that is also equivalent to the original program. Here, we describe a clever way to ensure that for each array we use, we are not overwriting anything or reading any entry that we shouldn’t be reading. This will help us check the correctness of the transformations we have been working with.

Thus, we will create a class for each type of array we use to help us check ourselves while we are reading and writing to array entries. In order to do these checks, we need to make sure that:

1. Every time we write to an array entry, we have not written to this particular entry in the past; and
2. Every time we read an array entry, we have already written to this particular array entry in the past.

So, our class will have functionality for this. Note that, specifically for \( y_1 \) and \( y_2 \) we will in fact be writing to each of these arrays more than once, but if we have another dimension such as statement that keeps track of which for loop we are in, we know that it will be a whole other dimension of the array in which we are reading and storing new values of \( y_1 \) and \( y_2 \). An example of this for our 5-dimensional arrays using complex iterators as described at the end of Section 5.1 can be found in Appendix C.1, and our class that allows logically concise programs is listed under Appendix C.2.

Note that ideally we would not have to invent this array to check for correctness if we had the polyhedral model for Chapel like we do for C. If we had the polyhedral model for Chapel, then we would expect that the Chapel compiler could look at the code and automatically decide the best transformation for what the programmer wants. Thus, as Chapel programmers, we hope for a version of Chapel that uses the polyhedral model to check the correctness of our transformation or iterator, rather than having to write an entire new class or record to manually check ourselves, which can be quite tedious.

7 Findings From Iterator-Based Optimization

After discussing several possible ways to optimize the Deriche algorithm using Chapel, we have provided the following table of running times (next to the running time for the original Deriche PolyBench benchmark written in C) after running each program just once. As a reminder, the computer we used for these experiments had an Intel® Core™ i5-6500 CPU, with a clock speed of 3.20GHz, 6MB of Level 3 cache, and 8GB of RAM. To measure these times, we used the Chapel class Timer combined with the functions start(), stop(), and elapsed(). Here, we only timed the Deriche calculation itself, not including the initialization of the variables, as is done in the PolyBench benchmark.

To allow for more space horizontally in the table, we corresponded numbers to the titles, excluding that of the original C code. The following is a key for those numbers:

1. Pure Chapel version, as discussed in the beginning of Section 5 and shown in Appendix B.1
2. Simple concise version, as introduced in Section 5.3
3. #2 with iterators, also discussed in Section 5.3 and shown in Appendix B.6
4. #3 with resettable scalars, as discussed later in Section 5.3 (class shown in Appendix C.3)
5. #4 in two phases, as discussed from Section 5.2 and shown in Appendix B.7
6. #5 using 1-Dimensional arrays, as discussed in Section 5.4 and shown in Appendix B.8
We have also included a graph, shown in Figure 6, of MFLOPS (floating point operations per second) by the input dataset size to give a visualization of how fast our computer was able to run for each of these optimizations. We have included datasets for the original C code, #1, #5 and #6 from the table. The most interesting result to look at here is the MFLOPS for the last optimization we discussed, which involved 1-Dimensional arrays. Clearly, the speed was significantly higher than any of the other optimizations, which we will discuss later.

For now, let us briefly ignore the results of the 1-Dimensional array experiment. As we can see, if we compare these results to those of Table 2 from Section 4.3, we will notice that for the smaller sized datasets, the performance is much better than the code in C. However, as the dataset sizes get larger, in general the running time increases much more than we would expect. In fact, for the larger datasets, the performance for all our optimizations in Chapel (with the exception of the 1-Dimensional Array example) is worse than the original C code.

Nonetheless, now turning to the 1-Dimensional Array experiment, we see that the performance is by far much better than the original C code, even in very large arrays. This is extremely exciting, especially when we look more closely at just how much faster the 1-Dimensional array example is compared to the code in C. This suggests that our intuition of using less memory was extremely helpful in our experimentation, since we got such positive results.

<table>
<thead>
<tr>
<th>Dataset size (wxh)</th>
<th>Original C</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.00018</td>
<td>0.000039</td>
<td>0.000057</td>
<td>0.000326</td>
<td>0.000205</td>
<td>0.000244</td>
<td>0.0000620</td>
</tr>
<tr>
<td>192x198</td>
<td>0.0011005</td>
<td>0.000309</td>
<td>0.000369</td>
<td>0.00285</td>
<td>0.00131</td>
<td>0.000533</td>
<td>0.000319</td>
</tr>
<tr>
<td>720x480</td>
<td>0.013465</td>
<td>0.012766</td>
<td>0.012645</td>
<td>0.0251</td>
<td>0.0228</td>
<td>0.0149</td>
<td>0.00501</td>
</tr>
<tr>
<td>1680x1320</td>
<td>0.0857565</td>
<td>0.087152</td>
<td>0.083661</td>
<td>0.0929</td>
<td>0.140</td>
<td>0.0874</td>
<td>0.0294</td>
</tr>
<tr>
<td>4000x400</td>
<td>0.5879255</td>
<td>0.683068</td>
<td>0.65393</td>
<td>0.814</td>
<td>1.36</td>
<td>0.808</td>
<td>0.241</td>
</tr>
<tr>
<td>8000x6600</td>
<td>2.0082745</td>
<td>2.125704</td>
<td>2.043784</td>
<td>2.39</td>
<td>3.52</td>
<td>2.21</td>
<td>0.737</td>
</tr>
<tr>
<td>9000x7000</td>
<td>2.5415975</td>
<td>2.546144</td>
<td>2.451868</td>
<td>2.87</td>
<td>4.25</td>
<td>2.69</td>
<td>0.930</td>
</tr>
</tbody>
</table>

Table 3: All running time results for Chapel code, in seconds
8 Conclusions

The main purpose of this paper was to first determine if it would be worth considering using Chapel to improve performance of the Deriche Recursive Filtering Algorithm. We ran some simple experiments on the C code and found that PluTo failed to improve performance, and our simple manual efforts to manually optimize the code seemed to be somewhat successful. Furthermore, after our own experimentation, we realized that PluTo would not be able to come up with the successful types of optimization that we found. Since PluTo does not look for transformations that have to do specifically with data flow and the data space, we had to use our own ideas to optimize in those ways. Thus, we turned to Chapel and translated the Deriche Algorithm, playing with iterators and records to see if we could improve performance.

It definitely was exciting to see that our final example of using 1-Dimensional Arrays improved speed so much. Obviously, we would have hoped to see better results from our other experiments. However, we are still satisfied with the fact that we were able to express these iteration-space transformations in Chapel, which will help continue the ongoing question of whether or not using Chapel and its iterators is a good method when trying to improve performance. Thus, we conclude that it would be interesting and worthwhile to analyze other PolyBench algorithms and see how they compare when considering iteration-space performance tuning.

References


A C Code

A.1 PolyBench Deriche C code

```c
A = (SCALAR_VAL(1.0) - EXP_FUN(-alpha)) * (SCALAR_VAL(1.0) - EXP_FUN(-alpha)) / (SCALAR_VAL(1.0) + SCALAR_VAL(2.0) * alpha + EXP_FUN(-alpha) - EXP_FUN(SCALAR_VAL(2.0) * alpha));

a1 = a5 = k;
a2 = a6 = k * EXP_FUN(-alpha);  // alpha = SCALAR_VAL(1.0);
a3 = a7 = k * EXP_FUN(-alpha);  // alpha = SCALAR_VAL(1.0);
a4 = a8 = -k * EXP_FUN(SCALAR_VAL(2.0) * alpha);
b1 = POW_FUN(SCALAR_VAL(2.0), -alpha);
b2 = -EXP_FUN(SCALAR_VAL(2.0) * alpha);
c1 = c2 = 1;

for (i = 0; i < PB_W; i++) {
    ym1 = SCALAR_VAL(0.0);
    ym2 = SCALAR_VAL(0.0);
    xm1 = SCALAR_VAL(0.0);
    for (j = 0; j < PB_H; j++) {
        y1[i][j] = a1 * imgIn[i][j] + a2 * xm1 + b1 * ym1 + b2 * ym2;
        xm1 = imgIn[i][j];
        ym2 = ym1;
        ym1 = y1[i][j];
    }
}

for (i = 0; i < PB_W; i++) {
    yp1 = SCALAR_VAL(0.0);
    yp2 = SCALAR_VAL(0.0);
    xp1 = SCALAR_VAL(0.0);
    for (j = PB_H - 1; j >= 0; j--) {
        y2[i][j] = a3 * xp1 + a4 * xp2 + b1 * yp1 + b2 * yp2;
        xp2 = xp1;
        xp1 = imgIn[i][j];
        yp2 = yp1;
        yp1 = y2[i][j];
    }
}

for (i = 0; i < PB_W; i++) {
    imgOut[i][j] = c1 * (y1[i][j] + y2[i][j]);
}

for (j = 0; j < PB_H; j++) {
    tm1 = SCALAR_VAL(0.0);
    ym1 = SCALAR_VAL(0.0);
    ym2 = SCALAR_VAL(0.0);
    for (i = 0; i < PB_W; i++) {
        y1[i][j] = a5 * imgOut[i][j] + a6 * tm1 + b1 * ym1 + b2 * ym2;
        tm1 = imgOut[i][j];
        ym2 = ym1;
        ym1 = y1[i][j];
    }
}

for (j = 0; j < PB_H; j++) {
    tp1 = SCALAR_VAL(0.0);
    tp2 = SCALAR_VAL(0.0);
    yp1 = SCALAR_VAL(0.0);
    yp2 = SCALAR_VAL(0.0);
    for (i = PB_W - 1; i >= 0; i--) {
        y2[i][j] = a7 * tp1 + a8 * tp2 + b1 * yp1 + b2 * yp2;
        tp2 = tp1;
        tp1 = imgOut[i][j];
        yp2 = yp1;
        yp1 = y2[i][j];
    }
}
```

for (i = 0; i < PB_W; i++) {
```
B Chapel Code of the Deriche Algorithm

B.1 Original Algorithm

```chapel
// width and height of input dataset will come from user
// NOTE: w corresponds to W which is a #defined constant in the C code, and
// also to w, which is initialized with W, and also to _PB_W,
// which is one of those selected by a macro to allow experiments in which
// the compiler does or does not know the loop bounds
config var w: int = read(int);
cfg var h: int = read(int);

// creating arrays
var imgIn: [0..w-1,0..h-1] real;  // w-1, h-1 because in Chapel everything is
                                  // inclusive
var imgOut: [0..w-1,0..h-1] real;
var y1: [0..w-1,0..h-1] real;
var y2: [0..w-1,0..h-1] real;

// initialize alpha
const alpha: real = 0.25;

// initializing imgIn array
for i in 0..w-1 {
    for j in 0..h-1 {
        imgIn[i,j] = ((313*i+991*j)%65536) / 65535.0;
    }
}

// initialize and start timer
use Time;
var t: Timer;
t.start();

const k: real = (1.0-Math.e**(-alpha))*(1.0-Math.e**(-alpha))/(1.0+2.0*
    Math.e**-(alpha-Math.e**(2.0*alpha)));
const a1: real = k;
const a5: real = k;
const a2: real = k*Math.e**(-alpha)*(alpha-1.0);
const a6: real = k*Math.e**(-alpha)*(alpha-1.0);
const a3: real = k*Math.e**(-alpha)*(alpha+1.0);
const a7: real = k*Math.e**(-alpha)*(alpha+1.0);
const a4: real = -k*Math.e**(-2.0*alpha);
const a8: real = -k*Math.e**(-2.0*alpha);
const b1: real = 2.0*(-alpha);
const b2: real = -Math.e**(-2.0*alpha);
const c1: real = 1;
const c2: real = 1;

var ym1: real;
var ym2: real;
var xm1: real;
var yp1: real;
var yp2: real;
var xp1: real;
var xp2: real;
var tm1: real;
var tp1: real;
var tp2: real;

for i in 0..w-1 {
    ym1 = 0.0;
    ym2 = 0.0;
    xm1 = 0.0;
    for j in 0..h-1 {
        y1[i,j] = a1*imgIn[i,j] + a2*xm1 + b1*ym1 + b2*ym2;
        xm1 = imgIn[i,j];
        ym2 = ym1;
    }
}
```

ym1 = y1[i, j];
}

for i in 0..w-1 {
  yp1 = 0.0;
  yp2 = 0.0;
  xp1 = 0.0;
  xp2 = 0.0;
  for j in 0..h-1 by -1 {
    y2[i, j] = a3*xp1 + a4*xp2 + b1*yp1 + b2*yp2;
    xp2 = xp1;
    xp1 = imgIn[i, j];
    yp2 = yp1;
    yp1 = y2[i, j];
  }
}

for i in 0..w-1 {
  for j in 0..h-1 {
    imgOut[i, j] = c1 * (y1[i, j] + y2[i, j]);
  }
}

for j in 0..h-1 {
  tm1 = 0.0;
  ym1 = 0.0;
  for i in 0..w-1 {
    y1[i, j] = a5*imgOut[i, j] + a6*tm1 + b1*ym1 + b2*ym2;
    tm1 = imgOut[i, j];
    ym2 = ym1;
    ym1 = y1[i, j];
  }
}

for j in 0..h-1 {
  tp1 = 0.0;
  tp2 = 0.0;
  yp1 = 0.0;
  yp2 = 0.0;
  for i in 0..w-1 by -1 {
    y2[i, j] = a7*tp1 + a8*tp2 + b1*yp1 + b2*yp2;
    tp2 = tp1;
    tp1 = imgOut[i, j];
    yp2 = yp1;
    yp1 = y2[i, j];
  }
}

for i in 0..w-1 {
  for j in 0..h-1 {
    imgOut[i, j] = c2*(y1[i, j] + y2[i, j]);
  }
}

// stop timer
stop();
write(t.elapsed());

B.2 Using Simple Iterators

// NOTE: all of the initialization of variables stays the same as in
// Appendix B.1

iter ij_forwards(i: int, h: int): (int, int, int) {
  for i in 0..w-1 {
    yield (i, 0, -9996); // -9996 (j) should not matter, we shouldn't be
    looking at it
    for j in 0..h-1 {
      yield (i, 1, j);
    }
  }

for (i, statement, j) in ij_forwards(w, h) {
    if (statement == 0) {
        ym1 = 0.0;
        ym2 = 0.0;
        xml = 0.0;
    } else { // here, statement should be 1
        y1[i, j] = a1*imgIn[i, j] + a2*xml + b1*yml + b2*ym2;
        xml = imgIn[i, j];
        ym2 = ym1;
        yml = y1[i, j];
    }
}

iter i_forwards_j_backwards(w: int, h: int): (int, int, int) {
    for i in 0..w-1 {
        yield (i, 0, -9999);
        for j in 0..h-1 by -1 {
            yield (i, 1, j);
        }
    }
}

for (i, statement, j) in i_forwards_j_backwards(w, h) {
    if (statement == 1) {
        imgOut[i, j] = c1 * (y1[i, j] + y2[i, j]);
    }
}

iter ji_forwards(w: int, h: int): (int, int, int) {
    for j in 0..h-1 {
        yield (j, 0, -9998);
        for i in 0..w-1 {
            yield (j, 1, i);
        }
    }
}

for (j, statement, i) in ji_forwards(w, h) {
    if (statement == 0) {
        tml = 0.0;
        yml = 0.0;
        ym2 = 0.0;
    } else { // statement should be 1
        y1[i, j] = a5*imgOut[i, j] + a6*tml + b1*yml + b2*ym2;
        tml = imgOut[i, j];
        ym2 = yml;
        yml = y1[i, j];
    }
}

iter j_forwards_i_backwards(w: int, h: int): (int, int, int) {
    for j in 0..h-1 {
        yield (j, 0, -9999);
        for i in 0..w-1 {
            yield (j, 1, i);
        }
    }
}
```plaintext
yield (j, 0, -9997);
for i in 0..w-1 by -1 {
    yield (j, 1, i);
}
}
}
}
for (i, statement, j) in j_forwards_i_backwards(w, h) {
    if (statement == 0) {
        tp1 = 0.0;
        tp2 = 0.0;
        yp1 = 0.0;
        yp2 = 0.0;
    } else { // statement should be 1
        y2[i, j] = a7*tp1 + a8*tp2 + b1*yp1 + b2*yp2;
        tp2 = tp1;
        tp1 = imgOut[i, j];
        yp2 = yp1;
        yp1 = y2[i, j];
    }
}
for (i, statement, j) in ij_forwards(w, h) {
    if (statement == 1) {
        imgOut[i, j] = c2*(y1[i, j] + y2[i, j]);
    }
}

B.3 Using Complex Iterators

//NOTE: all of the initialization of variables stays the same as in Appendix B.1, except for y1, y2, and those that used to be scalars, which are now all 5-dimensional arrays, as seen below
var y1: [0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
var y2: [0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;

var yp1: [0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
var yp2: [0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;

var tm1: [0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
var tp1: [0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
var imgOutTemp: [0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
```

```
18 iter deriche_iterations(w: int, h: int): (int, int, int, int, int) {
19     for i in 0..w-1 {
20         yield (0, i, 0, 0, 0);
21         yield (0, i, 1, 0, 0);
22         yield (0, i, 2, 0, 0);
23         for j in 0..h-1 {
24             yield (0, i, 3, j, 0);
25             yield (0, i, 3, j, 1);
26             yield (0, i, 3, j, 2);
27             yield (0, i, 3, j, 3);
28         }
29     }
30     for i in 0..w-1 {
31         yield (1, i, 0, 0, 0);
32         yield (1, i, 1, 0, 0);
33         yield (1, i, 2, 0, 0);
34         yield (1, i, 3, 0, 0);
35         for j in 0..h-1 by -1 {
36             yield (1, i, 4, j, 0);
37             yield (1, i, 4, j, 1);
38             yield (1, i, 4, j, 2);
39             yield (1, i, 4, j, 3);
```

29
yield (1, i, 4, j, 4); }

for i in 0..w-1 {
    for j in 0..h-1 {
        yield (2, i, 0, j, 0); }

    for j in 0..h-1 {
        yield (3, 0, 0, j, 0);
        yield (3, 0, 1, j, 0);
        yield (3, 0, 2, j, 0);
        for i in 0..w-1 {
            yield (3, i, 3, j, 0);
            yield (3, i, 3, j, 1);
            yield (3, i, 3, j, 2);
            yield (3, i, 3, j, 3);
        }
    }

    for j in 0..h-1 {
        yield (4, 0, 0, j, 0);
        yield (4, 0, 1, j, 0);
        yield (4, 0, 2, j, 0);
        yield (4, 0, 3, j, 0);
        for i in 0..w-1 by -1 {
            yield (4, i, 4, j, 0);
            yield (4, i, 4, j, 1);
            yield (4, i, 4, j, 2);
            yield (4, i, 4, j, 3);
            yield (4, i, 4, j, 4);
        }
    }

    for i in 0..w-1 {
        for j in 0..h-1 {
            yield (5, i, 0, j, 0); }

    }

for (outer, i, statement, j, inner) in deriche_iterations(w, h) {

    if (outer == 0) {
        if (statement == 0) {
            ym[outer, i, statement, j, inner] = 0.0;
        }

        else if (statement == 1) {
            ym2[outer, i, statement, j, inner] = 0.0;
        }

        else if (statement == 2) {
            xml[outer, i, statement, j, inner] = 0.0;
        }

        else if (statement == 3) {
            if (inner == 0) {
                yl[outer, i, statement, j, inner] = a1*imgIn[i, j] + a2*xm[outer, i, (if j == 0 then 2 else 3), (if j == 0 then 0 else j-1), (if j == 0 then 0 else 1)] + b1*ym[outer, i, (if j == 0 then 0 else 3), (if j == 0 then 0 else 1), (if j == 0 then 0 else 3)] + b2*ym2[outer, i, (if j == 0 then 0 else 3), (if j == 0 then 0 else j-1), (if j == 0 then 0 else 2)];
            }

            else if (inner == 1) {
                xml[outer, i, statement, j, inner] = imgIn[i, j];
            }

            else if (inner == 2) {
                ym2[outer, i, statement, j, inner] = ym[outer, i, (if j == 0 then 0 else 3), (if j == 0 then 0 else j-1), (if j == 0 then 0 else 3)];
            }

            else if (inner == 3) {
                ym[outer, i, statement, j, inner] = yl[outer, i, statement, j, 0];
            }

}}
else if (outer == 1) {
    if (statement == 0) {
        yp1[outer, i, statement, j, inner] = 0.0;
    }
    else if (statement == 1) {
        yp2[outer, i, statement, j, inner] = 0.0;
    }
    else if (statement == 2) {
        xp1[outer, i, statement, j, inner] = 0.0;
    }
    else if (statement == 3) {
        xp2[outer, i, statement, j, inner] = 0.0;
    }
}
else if (outer == 2) {
    y2[outer, i, statement, j, inner] = a3*xp1[outer, i, (if j == h-1 then 2 else 4), (if j == h-1 then 0 else j+1), (if j == h-1 then 0 else 2)] + a4*xp2[outer, i, (if j == h-1 then 3 else 4), (if j == h-1 then 0 else j+1), (if j == h-1 then 0 else 2)] + b1*yp1[outer, i, (if j == h-1 then 0 else j+1), (if j == h-1 then 0 else j+1), (if j == h-1 then 0 else 4)] + b2*yp2[outer, i, (if j == h-1 then 1 else 4), (if j == h-1 then 0 else j+1), (if j == h-1 then 0 else 4)];
}
else if (outer == 3) {
    tml[outer, i, statement, j, inner] = 0.0;
}
else if (statement == 0) {
    ym1[outer, i, statement, j, inner] = 0.0;
}
else if (statement == 1) {
    ym2[outer, i, statement, j, inner] = 0.0;
}
else if (statement == 2) {
    ym[outer, i, statement, j, inner] = 0.0;
}
else if (statement == 3) {
    if (inner == 0) {
        yl[outer, i, statement, j, inner] = a5*imgOutTemp[2, i, 0, j, 0] + a6*tml[outer, (if i == 0 then 0 else i-1), (if i == 0 then 0 else 3), j, (if i == 0 then 0 else 1)] + b1*ym1[outer, (if i == 0 then 0 else i-1), (if i == 0 then 0 else 3), j, (if i == 0 then 0 else 3)] + b2*ym2[outer, (if i == 0 then 0 else i-1), (if i == 0 then 2 else 3), j, (if i == 0 then 0 else 2)];
    }
}
else if (inner == 1) {
    tml[outer, i, statement, j, inner] = imgOutTemp[2, i, 0, j, 0];
}
else if (inner == 2) {
    ym2[outer, i, statement, j, inner] = ym[outer, (if i == 0 then 0 else i-1), (if i == 0 then 1 else 3), j, (if i == 0 then 0 else 3)];
}
else if (inner == 3) {

ym1[outer, i, statement, j, inner] = y1[outer, i, statement, j, 0];
}
}

else if (outer == 4) {
    if (statement == 0) {
        tp1[outer, i, statement, j, inner] = 0.0;
    }
    else if (statement == 1) {
        tp2[outer, i, statement, j, inner] = 0.0;
    }
    else if (statement == 2) {
        yp1[outer, i, statement, j, inner] = 0.0;
    }
    else if (statement == 3) {
        yp2[outer, i, statement, j, inner] = 0.0;
    }
    else if (statement == 4) {
        if (inner == 0) {
            y2[outer, i, statement, j, inner] = a7*tp1[outer, (if i == w−1 then 0 else i+1), (if i == w−1 then 0 else 4), j, (if i == w−1 then 0 else 2)]
                + a8*tp2[outer, (if i == w−1 then 0 else 1), (if i == w−1 then 1 else 4), j, (if i == w−1 then 0 else i+1), (if i == w−1 then 2 else 4), j, (if i == w−1 then 0 else 4)]
                + b1*yp1[outer, (if i == w−1 then 0 else 4), j, (if i == w−1 then 3 else 4)]
                + b2*yp2[outer, (if i == w−1 then 0 else 1), (if i == w−1 then 0 else i+1), (if i == w−1 then 0 else 3)];
        }
        else if (inner == 1) {
            tp2[outer, i, statement, j, inner] = tp1[outer, (if i == w−1 then 0 else i+1), (if i == w−1 then 0 else 4), j, (if i == w−1 then 0 else 2)];
        }
        else if (inner == 2) {
            tp1[outer, i, statement, j, inner] = imgOutTemp[2, i, 0, j, 0];
        }
        else if (inner == 3) {
            yp2[outer, i, statement, j, inner] = yp1[outer, (if i == w−1 then 0 else i+1), (if i == w−1 then 2 else 4), j, (if i == w−1 then 0 else 4)];
        }
        else if (inner == 4) {
            yp1[outer, i, statement, j, inner] = y2[outer, i, statement, j, 0];
        }
    }
}
else if (outer == 5) {
    imgOut[i, j] = c2*(y1[3, i, 3, j, 0] + y2[4, i, 4, j, 0]);
}

B.4 Deriche Without Scalars, Using Iterators

//NOTE: all of the initialization of variables stays the same as in Appendix B.1

iter deriche_iterations(w: int, h: int): (int, int, int) {
    for i in 0..w−1 {
        for j in 0..h−1 {
            yield (0, i, j);
        }
    }
    for i in 0..w−1 {
        for j in 0..h−1 by −1 {
            yield (1, i, j);
        }
    }
    for i in 0..w−1 {
        for j in 0..h−1 {
            yield (2, i, j);
        }
    }
    for j in 0..h−1 {
        for i in 0..w−1 {
            yield (3, i, j);
        }
    }
}
B.5 Fusing Loops

//NOTE: The following applies the idea of two phases to the 'Complex Iterators' version of Deriche, which uses arrays with five dimensions
//Again, most of the initialization of variables are the same as in Appendix B.1, except for the ones mentioned below

var y1 = new darray5(w, h);
var y2 = new darray5(w, h);

darray ym1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray ym2[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray ym3[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray yp1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray yp2[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray yp3[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray xp1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray xp2[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray xp3[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray tm1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray tp1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray tp2[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray imgOutTemp[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;

darray ym = new darray5(w, h);
darray yp = new darray5(w, h);
darray xp = new darray5(w, h);
darray tp = new darray5(w, h);

for i in 0..w-1 {
    yield (0, i, 0, 0);
}

darray ym1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray ym2[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray ym3[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray yp1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray yp2[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray yp3[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray xp1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray xp2[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray xp3[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray tm1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray tp1[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray tp2[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;
darray imgOutTemp[0..5, 0..w-1, 0..4, 0..h-1, 0..4] real;

return ym;

//Again, most of the initialization of variables are the same as in Appendix B.1, except for the ones mentioned below

for (statement, i, j) in deriche_iterations(w, h) {
    if (statement == 0) {
        y1[statement, i, j] = a1*imgIn[i,j] + a2*(if j == 0 then 0.0 else imgIn[i,j-1]) + b1*(if j == 0 then 0.0 else y1[statement, i, j-1]) + b2*(if j == 0 then 0.0 else (if j == 0 || j == 1 then 0.0 else y1[statement, i, j-2]));
    }
    else if (statement == 1) {
        y2[statement, i, j] = a3*(if j == h-1 then 0.0 else imgIn[i,j+1]) + a4*(if j == h-1 || j == h-2 then 0.0 else imgIn[i,j+2]) + b1*(if j == h-1 then 0.0 else y2[statement, i, j+1]) + b2*(if j == h-1 || j == h-2 then 0.0 else y2[statement, i, j+2]);
    }
    else if (statement == 2) {
        imgOutTemp[statement, i, j] = c1*(y1[0,i,j] + y2[1,i,j]);
    }
    else if (statement == 3) {
        y1[statement, i, j] = a5*imgOut[i,j] + a6*(if i == 0 then 0.0 else imgOut[i-1,j]) + b1*(if i == 0 then 0.0 else y1[statement, i-1,j]) + b2*(if i == 0 || i == 1 then 0.0 else y1[statement, i-2,j]);
    }
    else if (statement == 4) {
        y2[statement, i, j] = a7*(if i == w-1 then 0.0 else imgOut[i+1,j]) + a8*(if i == w-1 || i == w-2 then 0.0 else imgOut[i+2,j]) + b1*(if i == w-1 then 0.0 else y2[statement, i+1,j]) + b2*(if i == w-1 || i == w-2 then 0.0 else y2[statement, i+2,j]);
    }
    else if (statement == 5) {
        imgOut[i,j] = c2*(y1[3,i,j] + y2[4,i,j]);
    }
}

for (statement, i, j) in deriche_iterations(w, h) {
    if (statement == 0) {
        y1[statement, i, j] = a1*imgIn[i,j] + a2*(if j == 0 then 0.0 else imgIn[i,j-1]) + b1*(if j == 0 then 0.0 else y1[statement, i, j-1]) + b2*(if j == 0 then 0.0 else (if j == 0 || j == 1 then 0.0 else y1[statement, i, j-2]));
    }
    else if (statement == 1) {
        y2[statement, i, j] = a3*(if j == h-1 then 0.0 else imgIn[i,j+1]) + a4*(if j == h-1 || j == h-2 then 0.0 else imgIn[i,j+2]) + b1*(if j == h-1 then 0.0 else y2[statement, i, j+1]) + b2*(if j == h-1 || j == h-2 then 0.0 else y2[statement, i, j+2]);
    }
    else if (statement == 2) {
        imgOutTemp[statement, i, j] = c1*(y1[0,i,j] + y2[1,i,j]);
    }
    else if (statement == 3) {
        y1[statement, i, j] = a5*imgOut[i,j] + a6*(if i == 0 then 0.0 else imgOut[i-1,j]) + b1*(if i == 0 then 0.0 else y1[statement, i-1,j]) + b2*(if i == 0 || i == 1 then 0.0 else y1[statement, i-2,j]);
    }
    else if (statement == 4) {
        y2[statement, i, j] = a7*(if i == w-1 then 0.0 else imgOut[i+1,j]) + a8*(if i == w-1 || i == w-2 then 0.0 else imgOut[i+2,j]) + b1*(if i == w-1 then 0.0 else y2[statement, i+1,j]) + b2*(if i == w-1 || i == w-2 then 0.0 else y2[statement, i+2,j]);
    }
    else if (statement == 5) {
        imgOut[i,j] = c2*(y1[3,i,j] + y2[4,i,j]);
    }
}
yield (0, i, 2, 0, 0);
for j in 0..h-1 {
    yield (0, i, 3, j, 0);
    yield (0, i, 3, j, 1);
    yield (0, i, 3, j, 2);
    yield (0, i, 3, j, 3);
}
for j in 0..h-1 by -1 {
    yield (1, i, 0, 0, 0);
    yield (1, i, 1, 0, 0);
    yield (1, i, 2, 0, 0);
    yield (1, i, 3, 0, 0);
    for j in 0..h-1 by -1 {
        yield (1, i, 4, j, 0);
        yield (1, i, 4, j, 1);
        yield (1, i, 4, j, 2);
        yield (1, i, 4, j, 3);
        yield (1, i, 4, j, 4);
    }
    for j in 0..h-1 {
        yield (2, i, 0, j, 0);
    }
}
for j in 0..h-1 {
    yield (3, 0, 0, j, 0);
    yield (3, 0, 1, j, 0);
    yield (3, 0, 2, j, 0);
    for i in 0..w-1 {
        yield (3, i, 3, j, 0);
        yield (3, i, 3, j, 1);
        yield (3, i, 3, j, 2);
        yield (3, i, 3, j, 3);
    }
    yield (4, 0, 0, j, 0);
    yield (4, 0, 1, j, 0);
    yield (4, 0, 2, j, 0);
    yield (4, 0, 3, j, 0);
    for i in 0..w-1 by -1 {
        yield (4, i, 4, j, 0);
        yield (4, i, 4, j, 1);
        yield (4, i, 4, j, 2);
        yield (4, i, 4, j, 3);
        yield (4, i, 4, j, 4);
    }
    for i in 0..w-1 {
        yield (5, i, 0, j, 0);
    }
}
for (outer, i, statement, j, inner) in deriche_iterations(w, h) {
    if (outer == 0) {
        if (statement == 0) {
            ym1[outer, i, statement, j, inner] = 0.0;
        } else if (statement == 1) {
            ym2[outer, i, statement, j, inner] = 0.0;
        } else if (statement == 2) {
            xm1[outer, i, statement, j, inner] = 0.0;
        } else if (statement == 3) {
            if (inner == 0) {
                y1[outer, i, statement, j, inner] = a1*imgIn[i,j] + a2*xm1[outer, i, (if j == 0 then 2 else 3), (if j == 0 then 0 else j-1), (if j == 0 then 0 else 1)] + b1*ym1[outer, i, (if j == 0 then 0 else 3), (if j == 0 then 0 else j-1), (if j == 0 then 0 else 3), (if j == 0 then 0 else j-1), (if j == 0 then 0 else j-1), (if j == 0 then 0 else 2)];
            } else if (inner == 1) {
                xm1[outer, i, statement, j, inner] = imgIn[i,j];
            }
        }
    }
else if (inner == 2) {
    ym2[outer, i, statement, j, inner] = ym1[outer, i, (if j == 0 then 0 else 3), (if j == 0 then 0 else j-1), (if j == 0 then 0 else 3)];
} else if (inner == 3) {
    ym1[outer, i, statement, j, inner] = y1[outer, i, statement, j, 0];
}

else if (outer == 1) {
    if (statement == 0) {
        yp1[outer, i, statement, j, inner] = 0.0;
    } else if (statement == 1) {
        yp2[outer, i, statement, j, inner] = 0.0;
    } else if (statement == 2) {
        xp1[outer, i, statement, j, inner] = 0.0;
    } else if (statement == 3) {
        xp2[outer, i, statement, j, inner] = 0.0;
    } else if (statement == 4) {
        if (inner == 0) {
            y2[outer, i, statement, j, inner] = a3*xp1[outer, i, (if j == h−1 then 2 else 4), (if j == h−1 then 0 else j+1), (if j == h−1 then 0 else 2)] + a4*xp2[outer, i, (if j == h−1 then 3 else 4), (if j == h−1 then 0 else j+1), (if j == h−1 then 0 else 4)] + b1*yp1[outer, i, (if j == h−1 then 0 else 1), (if j == h−1 then 0 else 4)] + b2*yp2[outer, i, (if j == h−1 then 1 else 4), (if j == h−1 then 0 else j+1), (if j == h−1 then 0 else 3)];
        } else if (inner == 1) {
            xp2[outer, i, statement, j, inner] = xp1[outer, i, (if j == h−1 then 2 else 4), (if j == h−1 then 0 else j+1), (if j == h−1 then 0 else 2)];
        } else if (inner == 2) {
            xp1[outer, i, statement, j, inner] = imgIn[i, j];
        } else if (inner == 3) {
            yp2[outer, i, statement, j, inner] = yp1[outer, i, (if j == h−1 then 0 else j+1), (if j == h−1 then 0 else 4)];
        } else if (inner == 4) {
            yp1[outer, i, statement, j, inner] = y2[outer, i, statement, j, 0];
        }
    }
}

else if (outer == 2) {
    imgOutTemp[outer, i, statement, j, inner] = c1 * (y1[0, i, 3, j, 0] + y2[1, i, 4, j, 0]);
}

else if (outer == 3) {
    if (statement == 0) {
        tmp[outer, i, statement, j, inner] = 0.0;
    } else if (statement == 1) {
        ym1[outer, i, statement, j, inner] = 0.0;
    } else if (statement == 2) {
        ym2[outer, i, statement, j, inner] = 0.0;
    } else if (statement == 3) {
        if (inner == 0) {
            y1[outer, i, statement, j, inner] = a5*imgOutTemp[2, i, 0, j, 0] + a6*tmp[outer, (if i == 0 then 0 else i-1), (if i == 0 then 0 else 3), j, (if i == 0 then 0 else 1)] + b1*ym1[outer, (if i == 0 then 0 else i-1), (if i == 0 then 0 else 3)] + b2*ym2[outer, (if i == 0 then 0 else i-1), (if i == 0 then 2 else 3), j, (if i == 0 else i-1)];
        }
    }
}
then 0 else 2]);
} else if (inner == 1) {
} else if (inner == 2) {
ym2[out, i, statement, j, inner] = ym1[out, (if i == 0 then 0 else 1), (if i == 0 then 1 else 3), j, (if i == 0 then 0 else 3)];
} else if (inner == 3) {
ym1[out, i, statement, j, inner] = y1[out, i, statement, j, 0];
}
}
else if (outer == 4) {
if (statement == 0) {
    tp1[out, i, statement, j, inner] = 0.0;
} else if (statement == 1) {
    tp2[out, i, statement, j, inner] = 0.0;
} else if (statement == 2) {
    yp1[out, i, statement, j, inner] = 0.0;
} else if (statement == 3) {
    yp2[out, i, statement, j, inner] = 0.0;
} else if (statement == 4) {
    if (inner == 0) {
        y2[out, i, statement, j, inner] = a7*tp1[out, (if i == w-1 then 0 else i+1), (if i == w-1 then 0 else 4)], j, (if i == w-1 then 0 else 2)] + a8*tp2[out, (if i == w-1 then 0 else i+1), (if i == w-1 then 1 else 4)], j, (if i == w-1 then 0 else i+1), (if i == w-1 then 2 else 4)], j, (if i == w-1 then 0 else 4)] + b1*yp1[out, (if i == w-1 then 0 else i+1), (if i == w-1 then 3 else 4)], j, (if i == w-1 then 0 else 3)];
    } else if (inner == 1) {
    tp2[out, i, statement, j, inner] = tp1[out, (if i == w-1 then 0 else i+1), (if i == w-1 then 0 else 4)], j, (if i == w-1 then 0 else 2)];
    } else if (inner == 2) {
    tp1[out, i, statement, j, inner] = imgOutTemp[2, i, 0, j, 0];
    } else if (inner == 3) {
    yp2[out, i, statement, j, inner] = yp1[out, (if i == w-1 then 0 else i+1), (if i == w-1 then 2 else 4)], j, (if i == w-1 then 0 else 4)];
    } else if (inner == 4) {
    yp1[out, i, statement, j, inner] = y2[out, i, statement, j, 0];
    }
} else if (outer == 5) {
    imgOut[i, j] = c2*(y1[3, i, 3, j, 0] + y2[4, i, 4, j, 0]);
}
}

B.6 Concise Deriche

//NOTE: all of the initialization of variables stays the same as in Appendix B.1, except for y1 and y2, which are of class derrayConcise2, as shown below
//Also note that the iterator is the same as in Appendix B.4, but we will put it here anyway for convenience
var y1 = new derrayConcise2(w, h);
var y2 = new derrayConcise2(w, h);

iter deriche_iterations(w: int, h: int): (int, int, int) {
    for i in 0..w-1 {
        for j in 0..h-1 {

    }
yield (0, i, j);
}
}
}

for i in 0..w-1 {
    for j in 0..h-1 by -1 {
        yield (1, i, j);
    }
}

for i in 0..w-1 {
    for j in 0..h-1 {
        yield (2, i, j);
    }
}

for j in 0..h-1 {
    for i in 0..w-1 {
        yield (3, i, j);
    }
}

for i in 0..w-1 {
    for j in 0..h-1 by -1 {
        yield (4, i, j);
    }
}

for j in 0..h-1 {
    for i in 0..w-1 {
        yield (5, i, j);
    }
}

/
*
Note: in the following, y1.get(i, j) reads element i, j, and y1.set(i, j) sets it and y1.prev(i, j) reads element i, j-1 or returns 0 if it exists, i.e. if j>0, etc.
*/

for (statement, i, j) in deriche_iterations(w, h) {
    if (statement == 0) {
        y1.set(i, j, a1*imgIn.get(i, j) + a2*imgIn.lower(i, j) + b1*y1.lower(i, j) + b2*y1.lowerlower(i, j));
    }
    else if (statement == 1) {
        y2.set(i, j, a3*imgIn.h(i, j) + a4*imgIn.h.h(i, j) + b1*y2.h(i, j) + b2*y2.h.h(i, j));
    }
    else if (statement == 2) {
        imgOut.set(i, j, c1*(y1.get(i, j) + y2.get(i, j)));
    }
    else if (statement == 3) {
        y1.set(i, j, a5*imgOut.get(i, j) + a6*imgOut.lower(i, j) + b1*y1.lower(i, j) + b2*y1.lowerlower(i, j));
    }
    else if (statement == 4) {
        y2.set(i, j, a7*imgOut.h(i, j) + a8*imgOut.h.h(i, j) + b1*y2.h(i, j) + b2*y2.h.h(i, j));
    }
    else if (statement == 5) {
        imgOut.set(i, j, c2*(y1.get(i, j) + y2.get(i, j)));
    }
}

B.7 Concise Deriche, Two Phases

//NOTE: everything here, including the code body, is the exact same as in Appendix B.6, except for the iterator, which we will put here

iter deriche_iterations(w: int, h: int): (int, int, int) {
    for i in 0..w-1 {
        for j in 0..h-1 {
            yield (0, i, j);
        }
    }
    for j in 0..h-1 by -1 {

B.8 Concise Deriche, Two Phases, One Dimension

//NOTE: the code body is the exact same as in Appendix B.6

//Here, we use one dimensional arrays
var y1c = new derrayConcise2Clever1Col(w, h);
var y2c = new derrayConcise2Clever1Col(w, h);
var y1r = new derrayConcise2Clever1Row(w, h);
var y2r = new derrayConcise2Clever1Row(w, h);

//same usual initialization for other variables
//iterator is the same as in Appendix B7

for (statement, i, j) in deriche_iterations(w, h) {
  if (statement == 0) { y1c.set(i, j, a1*imgIn.get(i, j) + a2*imgIn.lower(i, j) + b1*y1c.lower(i, j) + b2*y1c.lowerlower(i, j)); }
  else if (statement == 1) { y2c.set(i, j, a3*imgIn.higher(i, j) + a4*imgIn.higherhigher(i, j) + b1*y2c.higher(i, j) + b2*y2c.higherhigher(i, j)); }
  else if (statement == 2) { imgMid.set(i, j, c1*(y1c.get(i, j) + y2c.get(i, j))); }
  else if (statement == 3) { y1r.set(i, j, a5*imgMid.get(i, j) + a6*imgMid.lower(i, j) + b1*y1r.lower(i, j) + b2*y1r.lowerlower(i, j)); }
  else if (statement == 4) { y2r.set(i, j, a7*imgMid.higher(i, j) + a8*imgMid.higherhigher(i, j) + b1*y2r.higher(i, j) + b2*y2r.higherhigher(i, j)); }
  else if (statement == 5) { imgOut[i, j] = c2*(y1r.get(i, j) + y2r.get(i, j)); }
}

C External Classes

C.1 5-D Arrays for Deriche With Complex Iterators

class derray5 {
  const W: int;
  const H: int;
  const dom: domain(5);
  var Vals: [dom] real;
  proc derray5(width: int, height: int){ W = width;
    H = height;
    dom = {0..5, W–1,0..4,0..H–1,0..4};
  }
}
class carefulDerray5: derray5 {
    var Written: [dom] bool; // relies on this initializing to all false
    proc carefulDerray5(width: int, height: int) {
        W = width;
        H = height;
        dom = {0..5, 0..W−1, 0..4, 0..H−1, 0..4};
    }
    proc this(outer: int, i: int, statement: int, j: int, inner: int): real { // this one is used for reading from vals
        assert (Written[outer, i, statement, j, inner], "passed", (outer, i, statement, j, inner), "but this value has not been written to");
        return Vals[outer, i, statement, j, inner];
    }
    proc this(outer: int, i: int, statement: int, j: int, inner: int): real { // this one is used for writing to the vals
        assert (!Written[outer, i, statement, j, inner], "passed", (outer, i, statement, j, inner), "but this value has already been written to");
        Written[outer, i, statement, j, inner] = true;
        return Vals[outer, i, statement, j, inner];
    }
}

C.2 2-D Arrays for Concise Deriche

class derrayConcise2 {
    const W: int;
    const H: int;
    const dom: domain(2);
    var Vals: [dom] real;
    proc derrayConcise2(width: int, height: int) {
        W = width;
        H = height;
        dom = {0..W−1, 0..H−1};
    }
    proc this(i: int, j: int) ref: real {
        return Vals[i, j];
    }
    proc set(i: int, j: int, value: real) {
        Vals[i, j] = value;
    }
    proc get(i: int, j: int) {
        return Vals[i, j];
    }
    proc jlower(i: int, j: int) {
        if (j == 0) {
            return 0.0;
        }
        else {
            return Vals[i, j–1];
        }
    }
    proc jlowerlower(i: int, j: int) {
        if (j == 0 || j == 1) {
            return 0.0;
        }
        else {
            //...
return Vals[i, j - 2];
}

proc jhigher(i: int, j: int) {
if (j == H-1) {
return 0.0;
} else {
return Vals[i, j + 1];
}
}

proc jhigherhigher(i: int, j: int) {
if (j == H-1 || j == H-2) {
return 0.0;
} else {
return Vals[i, j + 2];
}
}

proc ilower(i: int, j: int) {
if (i == 0) {
return 0.0;
} else {
return Vals[i-1, j];
}
}

proc ilowerlower(i: int, j: int) {
if (i == 0 || i == 1) {
return 0.0;
} else {
return Vals[i-2, j];
}
}

proc ihigher(i: int, j: int) {
if (i == W-1) {
return 0.0;
} else {
return Vals[i+1, j];
}
}

proc ihigherhigher(i: int, j: int) {
if (i == W-1 || i == W-2) {
return 0.0;
} else {
return Vals[i+2, j];
}
}

C.3 2-D Arrays for Concise Deriche with Resets and Scalars

class derrayConcise2Cleverer {
const W: int;
const H: int;
const dom: domain(2);
var Vals: [dom] real;
var mostRecentWrite: real;
var previousWrite: real;

proc derrayConcise2Cleverer(width: int, height: int) {
W = width;
H = height;
dom = {0..W-1,0..H-1};
}
proc set (i: int, j: int, value: real) {
    previousWrite = mostRecentWrite;
    mostRecentWrite = value;
    Vals[i,j] = mostRecentWrite;
}

proc get (i: int, j: int) {
    return Vals[i,j];
}

proc resetScalars () {
    mostRecentWrite = 0;
    previousWrite = 0;
}

proc jlower (i: int, j: int) {
    return mostRecentWrite; // Vals[i, j-1];
}

proc jlowerlower (i: int, j: int) {
    return previousWrite; // Vals[i, j-2];
}

proc jhigher (i: int, j: int) {
    return mostRecentWrite; // Vals[i, j+1];
}

proc jhigherhigher (i: int, j: int) {
    return previousWrite; // Vals[i, j+2];
}

proc ilower (i: int, j: int) {
    return mostRecentWrite; // Vals[i-1, j];
}

proc ilowerlower (i: int, j: int) {
    return previousWrite; // Vals[i-2, j];
}

proc ilowerlower (i: int, j: int) {
    return previousWrite; // Vals[i-2, j];
}

proc ilowerlower (i: int, j: int) {
    return previousWrite; // Vals[i-2, j];
}

proc ilowerlower (i: int, j: int) {
    return previousWrite; // Vals[i-2, j];
}

proc ilowerlower (i: int, j: int) {
    return previousWrite; // Vals[i-2, j];
}

C.4 1-D Arrays

record derrayConcise2Clever1Row {
    const W: int;
    const H: int;
    const dom: domain(1);
    var Vals: [dom] real;
    var mostRecentWrite: real;
    var previousWrite: real;
}

proc derrayConcise2Clever1Row (width: int, height: int) {
    W = width;
    H = height;
    dom = {0..W-1};
}

proc set (i: int, j: int, value: real) {
    previousWrite = mostRecentWrite;
    mostRecentWrite = value;
    Vals[i] = mostRecentWrite;
}

proc get (i: int, j: int) {
    return Vals[i];
}
```
proc ilower(i: int, j: int) {
  if (i == 0) {
    return 0.0;
  } else {
    return mostRecentWrite; // Vals[i-1];
  }
}

proc ilowerr(i: int, j: int) {
  if (i == 0 || i == 1) {
    return 0.0;
  } else {
    return previousWrite; // Vals[i-2];
  }
}

proc higher(i: int, j: int) {
  if (i == W-1) {
    return 0.0;
  } else {
    return mostRecentWrite; // Vals[i+1];
  }
}

proc higher(i: int, j: int) {
  if (i == W-1 || i == W-2) {
    return 0.0;
  } else {
    return previousWrite; // Vals[i+2];
  }
}

proc jlower(i: int, j: int) {
  if (j == 0) {
    return 0.0;
  } else {
    return mostRecentWrite; // Vals[i-1];
  }
}
```

```
else {
    return mostRecentWrite; // Vals[j-1];
}

proc jlowerlower(i: int, j: int) {
    if (j == 0 || j == 1) {
        return 0.0;
    } else {
        return previousWrite; // Vals[j-2];
    }
}

proc jhigher(i: int, j: int) {
    if (j == H-1) {
        return 0.0;
    } else {
        return mostRecentWrite; // Vals[j+1];
    }
}

proc jhigherhigher(i: int, j: int) {
    if (j == H-1 || j == H-2) {
        return 0.0;
    } else {
        return previousWrite; // Vals[j+2];
    }
}

C.5 1-D Careful Arrays to check correctness

// Note: Since these are inherited from derrayConcise2CleverRow, etc., these
must be classes. But since in Chapel we cannot inherit from a record,
to use these particular classes we would have to change
derrayConcise2CleverRow back to classes. Although it would be slower,
this is okay, since our main focus using these classes is to check to
make sure we are correct rather than be fast.

class derrayCareful2Clever1Row : derrayConcise2Clever1Row {
    const W: int;
    const H: int;
    const dom: domain(1);
    var Vals: [dom] real;
    var mostRecentWrite: real;
    var previousWrite: real;
    var mrWi: int;
    var pWi: int;

    proc derrayCareful2Clever1Row(width: int, height: int) {
        W = width;
        H = height;
        dom = {0..W-1};
    }

    proc set(i: int, j: int, value: real) {
        previousWrite = mostRecentWrite;
        mostRecentWrite = value;
        Vals[i] = mostRecentWrite;
        // if being careful, record the i values of mrWi and pWi (the two scalars)
        pWi = mrWi;
        mrWi = i;
    }

    proc get(i: int, j: int) {
        return Vals[i];
    }

    proc ilower(i: int, j: int) {
        if (i == 0) {

31}
// if being careful, assert that i-1 is the i value of mRW
assert (mRW_i == i - 1);
return mostRecentWrite; // Vals[i-1];
}
}

proc lowerlower(i: int, j: int) {
if (i == 0 || i == 1) {
    return 0.0;
}
else {
    // if being careful, assert that i-2 is the i value of pW
    assert (pW_i == i - 2);
    return previousWrite; // Vals[i-2];
}
}

proc higher(i: int, j: int) {
if (i == W - 1) {
    return 0.0;
}
else {
    // if being careful, assert that i+1 is the i value of mRW
    assert (mRW_i == i + 1);
    return mostRecentWrite; // Vals[i+1];
}
}

proc higherhigher(i: int, j: int) {
if (i == W - 1 || i == W - 2) {
    return 0.0;
}
else {
    // if being careful, assert that i+2 is the i value of pW
    assert (pW_i == i + 2);
    return previousWrite; // Vals[i+2];
}
}

class derrayCareful2Clever1Col : derrayConcise2Clever1Col {
const W: int;
const H: int;
const dom: domain(1);
var Vals: [dom] real;
var mostRecentWrite: real;
var previousWrite: real;
// if being careful, we'll need fields to record the i and j values of
// mRW and pW (the two scalars)
var mRWj: int;
var pWj: int;
}

proc derrayCareful2Clever1Col(width: int, height: int) {
W = width;
H = height;
dom = {0..H-1};
}

proc set(i: int, j: int, value: real) {
previousWrite = mostRecentWrite;
mostRecentWrite = value;
Vals[j] = mostRecentWrite;
// if being careful, record the j values of mRW and pW (the two scalars)
pWj = mRWj;
mRWj = j;
}

proc get(i: int, j: int) {
return Vals[j];
}
proc jlower (i: int, j: int) {
    if (j == 0) {
        return 0.0;
    } else {
        // if being careful, assert that j-1 is the j value of mRW
        assert (mRWj == j-1);
        return mostRecentWrite; // Vals[j-1];
    }
}

proc jlowerlower (i: int, j: int) {
    if (j == 0 || j == 1) {
        return 0.0;
    } else {
        // if being careful, assert that j-2 is the j value of pW
        assert (pWj == j-2);
        return previousWrite; // Vals[j-2];
    }
}

proc jhigher (i: int, j: int) {
    if (j == H-1) {
        return 0.0;
    } else {
        // if being careful, assert that j+1 is the j value of mRW
        assert (mRWj == j+1);
        return mostRecentWrite; // Vals[j+1];
    }
}

proc jhigherhigher (i: int, j: int) {
    if (j == H-1 || j == H-2) {
        return 0.0;
    } else {
        // if being careful, assert that j+2 is the j value of pW
        assert (pWj == j+2);
        return previousWrite; // Vals[j+2];
    }
}